Associating shallow and selective global tree search with Monte Carlo for 9x9 go

Bruno Bouzy
Université Paris 5, UFR de mathématiques et d'informatique, C.R.I.P.5,
45, rue des Saints-Pères 75270 Paris Cedex 06 France,
tél: (33) (0)1 44 55 35 58, fax: (33) (0)1 44 55 35 35,
email: bouzy@math-info.univ-paris5.fr,
http: www.math-info.univ-paris5.fr/∼bouzy/

Abstract

This paper explores the association of shallow and selective global tree search with Monte Carlo in 9x9 go. This exploration is based on Olga and Indigo, two experimental Monte Carlo programs. We provide a min-max algorithm that iteratively deepens the tree until one move at the root is proved to be superior to the other ones. At each iteration, random games are started at leaf nodes to compute mean values. The progressive pruning rule and the min-max rule are applied to non terminal nodes. We set up experiments demonstrating the relevance of this approach. Indigo used this algorithm at the 8th Computer Olympiad held in Graz.

1 Introduction

Knowledge and tree search are the two main approaches to computer go [8]. However, other approaches are worth considering. The Monte Carlo approach has been developed by Brügmann [10], and recently by Bouzy and Helmstetter [9]. While using very little go knowledge, Monte Carlo go programs have performed well on 9x9 boards. Furthermore, associating domain-dependent knowledge with Monte Carlo has been very effective too [6]. Therefore, the remaining question is to study the association of tree search with Monte Carlo. Because a strength of Monte Carlo consists in avoiding the breaking down of the whole game into sub-games, the tree search considered in this study is global. Moreover, because the association of knowledge and Monte Carlo partly relies on selectivity, the tree search is selective. Finally, because of the combinatorial explosion, the tree search is shallow and applied on 9x9 boards only. Thus, this paper aims to explore the association of shallow and selective global tree search with Monte Carlo in 9x9 go. This exploration is based on our current research with the go playing programs Indigo and Olga.

Section 2 describes the related work about Monte Carlo and tree search. Section 3 provides an example and the algorithm that associates tree search and Monte Carlo. Section 4 gathers the results of experiments proving the relevance of this approach. Before perspectives and conclusion, section 5 discusses some questions raised by this approach.

2 Related Work

This section first relates Monte Carlo works on games, and then the tree search works that have inspired our present work.

2.1 Monte Carlo and games

The term Monte Carlo has a very broad meaning, using the random function of the computer and averaging outcomes [14]. Simulated annealing is a refinement that includes a temperature which decreases during the simulation process [17]. Monte Carlo simulations have already been used in other games than go. Abramson has 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" [1]. In games containing either randomness or hidden information, the use of simulations has nothing surprising. Poki uses simulations at Poker [4], and Maven at Scrabble [22]. Tesauro and Galperin have tried "truncated rollouts" in Back-gammon by using a parallel approach [23]. In Go, the information is not hidden and randomness is absent, apparently yielding very little interest for simulations. However, ten years ago, Brügmann showed the adequacy of simulated annealing in Go, with his program Gobble [10]. Recently, Kaminski has performed Brügmann's experiment again with his program Vegos [16]. Last year, Bouzy and Helmstetter studied Monte Carlo Go programs, experimentally demonstrating their effectiveness on 9x9 boards [9]. Since then, Bouzy has successfully associated Monte Carlo and knowledge in his program [6], yielding Indigo2003.

2.2 Tree search works relative to our study

The works about tree search and games are very numerous. This subsection only mentions the works relative to our aim of integrating Monte Carlo and tree search. Because Monte Carlo averages samples of terminal position evaluations, we are most interested in studies assuming that the position evaluation is not a value but a set of values, such as a probability distribution or a sample of values.

Palay suggested the use of a back-up rule when the evaluation is a probability distribution [19]. It has been used in the Baum and Smith work about the Bayesian player [2], and applied to Othello. Berliner proposed the B* algorithm [3]. And Korf and Chickering described a general best-first min-max algorithm [18] that also inspired our work.

Buro's probCut algorithm uses the results of shallow tree searches to prune moves [12]. Junghanns surveyed all the alfa-beta works [15]. Rivest studied a back-up rule using a complex formula [20], using an exponent p. When $p = 0$, the formula yields the classical min-max back-up rule and when $p = \infty$, it gives the average back-up, that is a feature of Monte Carlo. Sadikov, Kononenko and Bratko have shown that evaluations containing errors introduce a bias in the min-max values of the tree. The bias varies in the search depth, but remains constant for two sibling nodes [21]. This would explain the success of tree search in practice, although, in theory, pathologies exist in game tree. Chen has experimentally shown the effect of selectivity during tree search in Go [13].

3 Our Work

This section describes our work based on go playing programs. First, it defines the names of the programs mentioned along the paper. Second, it gives an intuitive view of the requirements. Third, it uncovers the algorithm that we used for 9x9 go. Fourth, it shows the algorithm performing on an example. Finally, it highlights two enhancements to control the width of the tree.

3.1 The programs' names

Our work being based on experiments about go playing programs, let us start by clearly defining the programs' names used in this paper. First, Indigo is the generic name of the program we have been developing over the past years [5]. It regularly attends computer go competitions. Each year, we set up a new release of this program, and Indigo2002 corresponds to Indigo's release at the end of 2002. Indigo2002 was mainly based on knowledge and tree search [7], and not on Monte Carlo. Second, Olga is the name of the working release of Indigo. Each year, we work on Olga, and if our work turns out to be effective, then Olga's effective features are integrated into Indigo. In 2003, we processed our work in three stages. The first improvement tested in Olga was the Monte Carlo approach, thus, at the beginning of 2003, Olga was a very little knowledge Monte Carlo go program described in [9]. Since Monte Carlo worked well, we also integrated knowledge into Olga in a second stage, which was satisfactory and described in [6]. Furthermore, we added selective global tree search in Olga in a final stage. Thus, in this paper, Olga refers to a program containing Monte Carlo, knowledge and selective global tree search. As this paper aims at describing the selective global tree search aspect with different tree search parameters, say X and Y, it might be useful to write Olga($X = x$, $Y = y$) to refer to Olga using the particular values x and y. By the end of 2003, Olga was better than Indigo2002, so we copied Olga into Indigo2003 which is consequently a knowledge-based tree search and Monte Carlo go program too. Indigo2003 attended the computer go olympiad held in Graz. Finally, we also mention Oleg in this paper because it is the Monte Carlo go program containing very little knowledge which was developed by Bernard Helmstetter [9].

3.2 The requirement

On the one hand, Olga and Oleg with very little knowledge, described in [9], were performing a depth-one global search without any selectivity. The attempt of performing a depth-two search failed because the branching factor of 9x9 game (about 80 in the beginning of a game) was too high. On the other hand, Olga with knowledge, described in [6], was performing a depth-one global search with high selectivity adapted for 19x19 board. With such a selectivity, Olga at depth one plays very quickly on 9x9 boards. Thus, the idea of the current work consists in improving 9x9 Olga with a depth-n selective global tree search.

However, we should not overlook the idea of progressive pruning [9]. This is the back-up of average values on the evaluations of the random games that must drive the process. When many moves are equal, we want the process to use as much CPU time as needed to discriminate between them, while when one move is clearly superior to the other, we want the process to find it out quickly.

By pruning the bad moves quickly, the progressive pruning algorithm is very efficient at the beginning of the process. However, the longer the algorithm performs, the fewer moves it prunes. At the end of the process, when few moves remain, many random games may be necessary to separate those moves to keep the best one. When the separation requires too much CPU time at depth one, would it be relevant to expand these moves to depth-two in order to speed up the process ? Very often, looking one move ahead enables the player to observe a difference between the effect of moves. Thus, we need an algorithm that prefers to expand nodes one depth further to perform random games starting at this depth, than passively await for the end of the current depth process. This requirement remains valid at any depth. If some moves are nearly equal at a given depth, expanding them to the next depth is worthwhile.

The statistical evaluation of a node consists in the expectation of sampled evaluations, given that the sequence of moves from the root until this node has been played out. When considering two sibling leaf nodes, the two players have played the same number of moves, and their expected evaluation can be compared. When considering a parent node and its children, the parent node updates an expected value assuming that a given sequence of moves has been played out while the children have expectation values assuming one additional move has been played after the given sequence. Thus, the statistical evaluation of a parent node cannot be compared to its children's one. In addition, in the min-max context, the parent node must compute its min-max value with the statistical evaluations of its children. In this context, it is inappropriate to compare the min-max value of the parent node with its own statistical evaluation. Now, considering two sibling nodes, the first one being a leaf node with a statistical evaluation, and the second one being a parent node of leaf nodes with statistical evaluations, it is inappropriate to compare the statistical evaluation of the first node with the min-max value of the second one. This remark can be extended to the comparison of min-max values of sibling nodes whose sub-trees have different depths. In conclusion, we need an algorithm that compares sibling nodes whose sub-trees have the same depth.

Moreover, [21] has shown that min-max back-ups on evaluations containing errors introduce a bias in the min-max values of the tree. As the bias depends on the search depth, we need a fixed-depth searching algorithm. Besides, the greater the error in the evaluations, the greater the bias in the min-max value. Consequently we need enough random games to lower the evaluation errors. When the algorithm reaches the maximal depth, it has to perform a sufficient number of random games.

To sum up, we need an algorithm that prunes some moves at depth one, expand the remaining ones to depth-two, run random games starting at depth-two, prune some depth-two moves, and so on, increasing the search depth iteratively either until one move remains at the root or until the maximal depth is reached.

3.3 The algorithm

This subsection yields the algorithm answering the requirements. Before showing the algorithm, we define the two classes of interest: `Node` and `J_stat` (representing the statistical player). We do not mention the properties of a node not related to progressive pruning.

```
class Node {
  float mean;
  float mean_sd;
  int n_children;
  bool all_are_equal;
}
```

While the node is terminal, the slot `mean` contains the mean value of the sample of evaluations. When the node becomes interior to the tree, this slot contains the min-max value of the node. `mean_sd` is the standard deviation of the mean. Its value is used by the progressive pruning rule. The slot `n_children` contains the number of remaining moves in the progressive pruning meaning [9]. The slot `all_are_equal` indicates whether all remaining moves are "equal" or not in the progressive pruning meaning. This slot is used to check the end of the algorithm's internal loop.

```
class J_Stat {
  int   r_games_p_depth;
  int   depth;
  int   depth_max;
  int   width_p;
  int   width_m;
  Node  root;
}
```

r_games_p_depth is the current number of random games performed while processing a given depth. The slot depth is the current depth of the algorithm at which random games are started. The slot depth_max is the maximal depth at which the process performs the random games. root is the root node of the tree. width_p is the number of moves selected by the knowledge-based move generator. In other words, it is the maximal branching factor of the tree which is developed by the algorithm. While it does not appear explicitly in the pseudo-code below, it is a parameter of importance in the experiments. width_m enables the algorithm to control the tree growing. Its use will be explained in subsection 3.5. Other slots are useful but have been omitted for clarity.

```
int J_Stat.choose_move() {
  depth = 0;
  width_p = WIDTH_PLUS;  // 5, 7, 9 or 11
  width_m = WIDTH_MINUS; // 2, 3, 4 or 5
  int n = N_R_GAMES;     // 2500
  do {
    depth = depth + 1;
    generate_nodes(depth);
    n *= 1+root.n_children;
    process(depth, n);
  } while ((root.n_children>1) && (depth<depth_max));
  return root.best_move();
}
```

The function choose_move() is the solution we offer. It returns the move chosen by the algorithm. The function best_move() returns the best move of a node. The function generate_nodes(int d) generates the nodes at depth d. The function process(int depth, int max) is defined below.

```
void J_Stat.process(int depth, int max) {
  r_games_p_depth = 0;
  int max_r = width_p ;
  while (    (root.n_children>1)
          && (!root.all_are_equal)
          && (r_games_p_depth<max)
          && ((max_r>width_m)||(depth==depth_max))
        ) {
    perform_random_games(depth);
    for (int d=depth-1; d>=0; d--) {
      int r = update_remaining_moves(d);
      if (d==depth-1) max_r = r;                // (a)
      update_min_max_values(d);
    }
  }
  cut_nodes(depth, width_m);                    // (b)
}
```

In this subsection, we decided to explain the basic version of the algorithm only. The two lines containing a comment are not part of the basic version. They refer to enhancements for controlling the width of the tree, explained in subsection 3.5. The other lines explained below correspond to the basic version.

The function `perform_random_games(int d)` performs random games starting at depth `d`. After each random game, it updates `mean` and `mean_sd` of the current node. The function `update_remaining_moves(int d)` updates the remaining moves of nodes situated at depth `d` with the progressive pruning rule. It updates `n_children` and `all_are_equal` of the current node. The function `update_min_max_values(int d)` applies the min-max back-up rule to nodes situated at depth `d`. After each min-max back-up, it updates `mean` and `mean_sd`.

To sum up, the algorithm is similar to iterative deepening. It stops when there is only one remaining move left at the root, or when the maximal depth is reached. Each depth has its own specificity. At the root, the goal is to find out the best move. At maximal depth, the random games are started. At non maximal depth, the progressive pruning rule and the min-max rule are applied. In order to perform all these updates, the whole tree is, of course, stored in the computer's memory.

3.4 An example

This subsection shows how the algorithm works on a very simple example. At the beginning, the moves are generated from the root node, resulting in the tree on the left of Figure 1. The leaf nodes are drawn with a white circle, and other nodes are gray. For clarity, we use `width_p` = 4. Thus, the root is expanded with four children. Then, the function `process()` runs at depth one, and during this process two moves are pruned leading to the tree situated on the right of Figure 1. Let us suppose that the ending conditions are true for depth one.

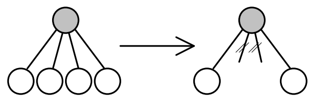

Figure 1: Performing depth 1.

Thus, the leaf nodes are expanded into the tree drawn on the left of Figure 2. Again, the random games are performed starting on depth-two nodes, and some moves are pruned leading to the tree located on the right of Figure 2. Supposing that the ending conditions are true for this depth, the leaf nodes are expanded once more to bring about the tree drawn on the left of Figure 3.

At depth three, the algorithm prunes other nodes, leaf nodes or interior nodes. For example, as shown by the tree drawn in the middle of Figure 3, it prunes a node situated at depth two, pointing out the possibility of pruning an

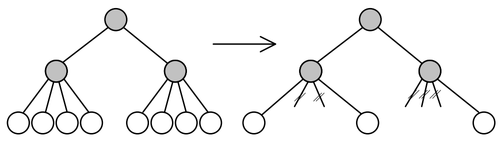

Figure 2: Performing depth 2.

interior node. Finally, a move situated at the root is pruned. This corresponds to the tree situated on the right of figure 3. Because one move is left at the root, the algorithm stops and returns this move.

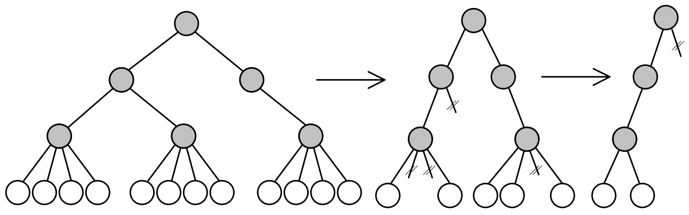

Figure 3: Performing depth 3.

3.5 Controlling the width of the tree

This subsection highlights the way in which the algorithm drastically controls the width of the tree. The instructions enabling the algorithm to control the width of the tree were indicated with a comment in subsection 3.3. The comment marked up with `(a)` apply to expanding nodes difficult to discriminate at the current depth. The comments marked up with `(b)` mention the node number limitation after processing a given depth. This last enhancement is useful when the loop ends up with either `r_games_p_depth` reaching `max` or `root.all_are_equal` being true. `max_r` is the maximum of `n_children` over all the nodes situated at depth - 1. `width_m` is a threshold that determines the maximal width of the tree from depth 0 to depth - 2. In the example shown in subsection 3.4, it is set up with the value 2. With the enhancement `(a)`, the algorithm does not perform random games if `max_r` reaches the threshold `width_m` and if the depth is not `depth_max`. Instead, the algorithm goes to the next depth. With the enhancement `(b)`, the processing of a given depth always terminates with less than `width_m` nodes at depth-1. Of course, the lines marked up with comments can be removed and the algorithm may work without them.

4 Experiments

In this section, we provide the results of the experiments carried out with this algorithm on 9x9 boards. An experiment is a set of confrontations. One confrontation consists in a match of 100 games between 2 programs, each program playing 50 games with Black. The result of such a confrontation is the mean score and a winning percentage. Given that the standard deviation of games played on 9x9 boards is roughly 15 points, 100 games enable our experiments to lower σ down to 1.5 point and to obtain a 95% confidence interval of which the radius equals 2σ, i.e., 3 points. We have used 2.4 GHz computers, and we mention the response time of each program. The variety of games is guaranteed by the random seed values that are different from one game to another. 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.

Subsection 4.1 highlights the relative strengths of programs using different depths. Subsection 4.2 underlines the relative strengths of programs using different widths. Then, subsection 4.3 shows the relative strengths of Olga against GNU Go 3.2 [11]. Finally, subsection 4.4 mentions the result of Indigo at the last 9x9 go competition held during the 8th Computer Olympiad.

4.1 Making the depth vary

This subsection contains the results of the experiment making `depth_max` vary. We set up different instances of Olga, each of them using their own value of `depth_max`. In the following, d and *Depth* refer to `depth_max` for short. For the same reason, in the following sections, $W+$ refers to `width_p`, and $W-$ to `width_m`.

In the first experiment, each instance of Olga uses $W+ = 7$, and $W- = 3$. Table 1 summarizes the results of the confrontations of Olga($Depth = d$) versus Olga($Depth = d-1$) and Olga($Depth = d-2$) for d ranging from 2 up to 5. The table mentions the mean score and the winning percentage assuming that the program of the column is the max player. The difference between Olga($Depth = 2$) and Olga($Depth = 1$) is clear as well as the difference between Olga($Depth = 3$) and Olga($Depth = 2$). These two confrontations experimentally prove the relevance of global tree search with Monte Carlo. For higher values of *Depth*, the upside is less significant, which confirms the fact that the returns diminish when the search depth increases. The results of depth five against depth four, and depth four against depth three are below the 3 points' threshold giving 95% confidence in the superiority of one program over the other one. More games should be performed to get a statistically significant conclusion. The winning percentages of Olga($d = 5$) over Olga($d = 4$) and over Olga($d = 3$) are the same. This is not a mistake, but it results from a too low number of games. A similar remark can be made about the winning percentages of Olga($d = 4$) over Olga($d = 3$) and over Olga($d = 2$).

d	2		3		4		5	
d-1	+7.7	61%	+7.6	63%	+1.8	54%	+1.0	52%
d-2			+12.8	70%	+5.9	54%	+4.2	52%

Table 1: Olga($Depth = d$) versus Olga($Depth = d-1$) and Olga($Depth = d-2$), $W+ = 7$, $W- = 3$.

In addition, table 2 summarizes the results of Olga($Depth = d$) versus Olga($Depth = d - 1$) and versus Olga($Depth = d - 2$) with $W+ = 9$ and $W- = 4$. The results of this experiment confirms the results of the first one. Going up to depth two from depth one, returns about 8 points and, going up to depth three from depth two returns about 7 points. But the returns diminish at depth three and even deeper. As in the previous table, many more results should be collected to conclude on the superiority of one program over the other one.

d	2		3		4		5	
d-1	+8.4	67%	+7.0	65%	+1.2	54%	+2.1	53%
d-2			+11.5	72%	+3.2	60%	+4.2	60%

Table 2: Olga($Depth = d$) versus Olga($Depth = d-1$) and Olga($Depth = d-2$), $W+ = 9$, $W- = 4$.

4.2 Making the width vary

This subsection contains the results of the experiment making $W-$ vary. In this experiment, each instance of Olga uses $Depth = 3$ and $W+ = 1 + 2W-$. Table 3 summarizes the results of the confrontations of Olga($W- = w$) versus Olga($W- = w - 1$) and Olga($W- = w - 2$) for w ranging from 3 up to 5. The table mentions the mean score and the winning percentage assuming that the program of the column is the max player.

w	3		4		5	
w-1	+1.5	57%	+3.8	61%	+0.1	51%
w-2			+5.9	62%	+0.6	51%

Table 3: Olga($Width = d$) versus Olga($W- = w - 1$) and Olga($W- = w - 2$), with $Depth = 3$.

This table shows that going from $W- = 3$ up to $W- = 4$ is worth considering. About four points are gained on average. But, going up to $W- = 5$ does not seem satisfactory. We can explain this fact by the good move ordering of the knowledge move generator for the first ranked moves and the bad ordering for the moves ranked after the fifth position.

Table 4 yields the CPU time in minutes used by Olga($Depth$, $W-$) for playing one 9x9 game, for $Depth$ ranging from 1 up to 5 and $W-$ ranging from 2 up to 5.

This table shows that time increases with both $Depth$ and $W-$. Considering our computer olympiad program, Olga($Depth = 3$, $W- = 3$), is it better to increase $W-$ or $Depth$? Choosing between the two possibilities is not obvious. Increasing $Depth$ seems better than increasing $W-$ because no programming

	d=1	d=2	d=3	d=4	d=5	d=6
w=3	2	5	10	16	24	35
w=4	3	7	12	27	42	55
w=5	3	10	22	44	65	100
w=6	3	10	30	60	100	160

Table 4: CPU time used by Olga($Depth = d$, $W- = w$) on 2.4 Ghz computers.

effort has to be made to increase $Depth$, while knowledge based move generation has to be improved if we want to increase $W-$.

4.3 Playing against GNU Go 3.2

To measure the effect of the variation in the depth and the width of the tree, we have set up confrontations between Olga($Depth$, $W-$) and GNU Go 3.2. Table 5 shows the mean scores, and table 6 shows the winning percentages. The results are given from Olga's viewpoint. For each table, the last column (line respectively) indicates the mean of the previous columns (lines respectively).

	d=1	d=2	d=3	d=4	d=5	d=6	total
w=3	-5.4	-3.2	-5.7	-6.3	-6.5	-1.5	-4.8
w=4	-6.5	-8.7	-3.0	-5.0	-4.6	-3.7	-5.2
w=5	-8.1	-6.8	-4.1	-2.7	-2.9	-1.0	-4.3
w=6	-6.6	-5.1	-1.8	-0.8	-3.1		-3.5
total	-6.6	-6.0	-3.7	-3.7	-4.3	-2.1	-4.4

Table 5: Mean score obtained by Olga($Depth = d$, $W- = w$) against GNU Go 3.2.

	d=1	d=2	d=3	d=4	d=5	d=6	total
w=3	33	40	33	37	34	44	37
w=4	33	32	40	39	44	41	38
w=5	40	36	38	37	47	45	40
w=6	36	38	48	49	45		42
total	35	36	40	41	42	43	39

Table 6: Winning percentage obtained by Olga($Depth = d$, $W- = w$) against GNU Go 3.2.

First, the total line of table 6 shows a correlation between $Depth$ and the winning percentage. Second, the total column of table 6 also shows a correlation between $W-$ and the winning percentage. On average, Olga wins 39% of the games. Third, in terms of mean score obtained by Olga, the correlation still appears, but less clearly. On average, the mean score equals -4.4. We think that the correlation clearly exists within self-play, because other elements than tree search remain constant. The correlation observed within self-play diminishes against differently designed opponents such as GNU Go. Unlike GNU Go, Olga still lacks important elements such as a good life and death move generator or a good territory move generator. Consequently, to improve Olga, some effort should be made on such elements that are very different from global tree search and Monte Carlo. We did not compute the data for $w > 6$ and $d > 6$ because of the memory constraints: each internal node of the tree contains a whole board with its knowledge, and the whole tree is kept in the computer's memory. Hence, it is impossible to yield the results for $(w, d) = (6, 6)$.

4.4 9x9 competition at the 2003 Computer Olympiad

Indigo, a copy of Olga(d=3, w=3), has participated in the 9x9 go competition during the 8th Computer Olympiad in Graz, in November 2003. Indigo ranked 4th upon 10 programs with 11 wins and 7 losses, which was a reasonable result, demonstrating the relevance of this approach against other differently designed programs.

5 Discussion

In this section we mention the advantages of the approach in term of complexity. Then, we discuss the relevance of adding classical tree search enhancements within our statistical search. Finally, we mention the possibility of scaling the results up to 19x19.

5.1 Complexity of the approach

W being the width of the tree and $Depth$ the search depth, full-width and full-depth tree search algorithms have a time complexity in W^{Depth}. N being the number of random games per candidate move, depth-one Monte Carlo has a time complexity in $N \times W \times Depth$. The Monte Carlo and tree search approach developed in this work, has a time complexity in $N \times W_+ \times Depth \times W_-^{Depth_{max}-1}$ because it starts a depth-one Monte Carlo tree search at leaf nodes of a tree whose width equals W_-, and whose depth equals $Depth_{max} - 1$. Thus, on 9x9 boards, the time complexity can fit the computing power by adjusting $Depth_{max}$, W_+, and W_- appropriately, and the program using this hybrid approach is endowed with some global tactical ability. Besides, the space complexity should also be underlined. The computer's memory is mostly occupied by internal nodes containing a board with its domain-dependent knowledge: the matched patterns, and the global evaluation. The size of leaf nodes is not taken into account because they only contain statistical information. Furthermore, the memory size occupied by the running random game is in $Depth$, and it is not taken into account either. At a depth inferior to $Depth_{max} - 1$, the tree branching factor being equal or inferior to W_{-}, the space complexity of the algorithm is in $W_{-}^{Depth_{max}-1}$.

5.2 Classical tree search enhancements

Our algorithm uses the min-max back-up rule and approximately follows the idea of iterative deepening. So far, it has not used the transposition table principle. How could the transposition principle be integrated into our algorithm ? When two leaf nodes refer to the same position, it would be interesting to merge the two samples. However, this enhancement is not urgent because the depths remain very shallow at the moment, not greater than five.

Since ProbCut includes the idea of correlation of position evaluations situated at different depths [12], how to establish the link between our algorithm and ProbCut ? Before performing a deep search, ProbCut performs a shallow tree search to obtain rough alfa-beta values and prune moves with some confidence level. In our work, before performing the next depth search, the algorithm prunes moves by using the results of the current depth search. In this respect, at root node, our algorithm corresponds to a simple version of ProbCut.

5.3 Scaling up to 19x19

On 2.4 GHz computers, the algorithm performs well on 9x9 with $Depth_{max} = 3$ and $W+ = 7$ in about 10 minutes. The same algorithm plays on 19x19 with $Depth_{max} = 1$ and $W+ = 7$ in about 50 minutes. Knowing the time used by the algorithm to play a 19x19 game with $Depth_{max} = 3$ and $W+ = 7$, and knowing the number of points corresponding to the self-play improvement are worth considering. Actually, a ten-game test shows that a 19x19 game lasts about 2 hours for $Depth_{max} = 2$, and 6 hours for $Depth_{max} = 3$. A ten-point improvement is observed with $Depth_{max} = 2$, and a fifteen-point improvement with $Depth_{max} = 3$, which has nothing exceptional. Of course, this assessment is not statistically significant, and it must be carried out again with more games in a few years' time. In conclusion, on 19x19 boards, as described in [6], instead of increasing $Depth_{max}$, our current strategy consists in increasing $W+$.

6 Perspectives and conclusion

Various perspectives can be considered. First, to make it more general, we wish to apply our algorithm to another mind game. The game of Amazons may be a relevant choice. Besides, since most of the thinking time of the program is spent at the beginning of the game, we want to develop an opening book. Finally, as 5x5 Go was solved by [24], we also plan to apply this algorithm on small boards ranging from 5x5 up to 9x9.

To sum up, we have issued the algorithm that Indigo used during the 9x9 Go competition at the last Computer Olympiad held in Graz in November

2003. This algorithm combines a shallow and selective global tree search with Monte Carlo. It illustrates the model developed by Abramson [1]. To our knowledge, this algorithm is new within the computer go community. It results from a work about Monte Carlo alone [9], and a work associating Monte Carlo and knowledge [6]. Following this line, the current work shows the association between tree search and Monte Carlo. As could be expected, we have observed an improvement when increasing the depth of the search, or when increasing the width of the tree. This improvement is clearer in the self-play context than against GNU Go. On today's computers, a tree search with $depth = 3$, $W+ = 7$, and $W- = 3$ offers the satisfactory compromise between time and level on 9x9 boards. However, depth-four and depth-five tree searches are possible, and furthermore they reach a better level. We believe that combining global tree search and Monte Carlo will strengthen go programs in the future.

References

[1] B. Abramson. Expected-outcome : a general model of static evaluation. *IEEE Transactions on PAMI*, 12:182–193, 1990.

[2] E. Baum and W. Smith. A bayesian approach to relevance in game-playing. *Artificial Intelligence*, 97:195–242, 1997.

[3] H. Berliner. The B* tree search algorithm: a best-first proof procedure. *Artificial Intelligence*, 12:23–40, 1979.

[4] D. Billings, A. Davidson, J. Schaeffer, and D. Szafron. The challenge of poker. *Artificial Intelligence*, 134:201–240, 2002.

[5] B. Bouzy. Indigo home page. www.math-info.univ-paris5.fr/~bouzy/INDIGO.html, 2002.

[6] B. Bouzy. Associating knowledge and Monte Carlo approaches within a go program. In *7th Joint Conference on Information Sciences*, pages 505–508, Raleigh, 2003.

[7] B. Bouzy. The move decision process of Indigo. *International Computer Game Association Journal*, 26(1):14–27, March 2003.

[8] B. Bouzy and T. Cazenave. Computer go: an AI oriented survey. *Artificial Intelligence*, 132:39–103, 2001.

[9] B. Bouzy and B. Helmstetter. Monte Carlo go developments. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, *10th Advances in Computer Games*, pages 159–174, Graz, 2003. Kluwer Academic Publishers.

[10] B. Brügmann. Monte Carlo go. www.joy.ne.jp/welcome/igs/Go/computer/-mcgo.tex.Z, 1993.

[11] D. Bump. GNU Go home page. www.gnu.org/software/gnugo/devel.html, 2003.

[12] M. Buro. Probcut: an effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[13] K. Chen. A study of decision error in selective game tree search. *Information Sciences*, 135:177–186, 2001.

[14] Fishman. *Monte-Carlo : Concepts, Algorithms, Applications*. Springer, 1996.

[15] A. Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21(1):14–32, March 1998.

[16] P. Kaminski. Vegos home page. www.ideanest.com/vegos/, 2003.

[17] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, May 1983.

[18] R. Korf and D. Chickering. Best-first search. *Artificial Intelligence*, 84:299–337, 1994.

[19] A.J. Palay. *Searching with probabilities*. Morgan Kaufman, 1985.

[20] R. Rivest. Game-tree searching by min-max approximation. *Artificial Intelligence*, 34(1):77–96, 1988.

[21] A. Sadikov, I. Bratko, and I. Kononenko. Search versus knowledge: an empirical study of minimax on KRK. In Ernst A. Heinz H. Jaap van den Herik, Hiroyuki Iida, editor, *10th Advances in Computer Games*, pages 33–44, Graz, 2003. Kluwer Academic Publishers.

[22] B. Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134:241–275, 2002.

[23] G. Tesauro and G. Galperin. On-line policy improvement using Monte Carlo search. In *Advances in Neural Information Processing Systems*, pages 1068–1074, Cambridge MA, 1996. MIT Press.

[24] E. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk. Solving go on small boards. *International Computer Game Association Journal*, 26(2):92–107, June 2003.