Modifications of UCT and sequence-like simulations for Monte-Carlo Go

Yizao Wang
Center of Applied Mathematics
Ecole Polytechnique, Palaiseau, France
yizao.wang@polytechnique.edu

Sylvain Gelly
TAO (INRIA), LRI, UMR (CNRS - Univ. Paris-Sud)
University of Paris-Sud, Orsay, France
sylvain.gelly@lri.fr

***Abstract*— Algorithm UCB1 for multi-armed bandit problem has already been extended to Algorithm UCT which works for minimax tree search. We have developed a Monte-Carlo program, MoGo, which is the first computer Go program using UCT. We explain our modification of UCT for Go application and also the sequence-like random simulation with patterns which has improved significantly the performance of MoGo. UCT combined with pruning techniques for large Go board is discussed, as well as parallelization of UCT. MoGo is now a top-level Computer-Go program on 9×9 Go board.**

Keywords: Computer Go, Monte-Carlo Go, multi-armed bandit, UCT, sequence-like simulation, 3×3 patterns

I. Introduction

The history of Go stretches back some 4000 years and the game still enjoys a great popularity all over the world. Although its rules are simple (see http://www.gobase.org for a comprehensive introduction), its complexity has defeated the many attempts done to build a good Computer-Go player since the late 70's [1]. Presently, the best Computer-Go players are at the level of weak amateurs; Go is now considered one of the most difficult challenges for AI, replacing Chess in this role.

Go differs from Chess in many respects. First of all, the size and branching factor of the tree are significantly larger. Typically the Go board ranges from 9×9 to 19×19 (against 8×8 for the Chess board); the number of potential moves is a few hundred against a few dozen for Chess. Secondly, no efficient evaluation function approximating the minimax value of a position is available. For these reasons, the powerful alpha-beta search used by Computer-Chess players (see [2]) failed to provide good enough Go strategies.

Recent progress has been done regarding the evaluation of Go positions, based on Monte-Carlo approaches [3] (more on this in section II). However, this evaluation procedure has a limited precision; playing the move with highest score in each position does not end up in winning the game. Rather, it allows one to restrict the number of relevant candidate moves in each step. Still, the size of the (discrete) search space makes it hardly tractable to use some standard Reinforcement Learning approach [4], to enforce the exploration *versus* exploitation (EvE) search strategy required for a good Go player.

Another EvE setting originated from Game Theory, the multi-armed bandit problem, is thus considered in this paper. The multi-armed bandid problem models the gambler, choosing the next machine to play based on her past selections and rewards, in order to maximize the total reward [5]. The UCB1 algorithm proposed by Auer et al. in the multi-armed bandit framework [6] was recently extended to tree-structured search space by Kocsis et al. (algorithm UCT) [7].

The main contributions of the player we present (named MoGo) are: (i) modification of UCT algorithm for Go, (ii) original use of sequence-like simulations in Monte-Carlo evaluation function. Several algorithmic (dynamic tree structure [8], parallelized implementation) or heuristic (simple pruning heuristics) issues were also tackled. MoGo has reached a comparatively good Go level: MoGo has been ranked as the first Go program out of 142 on 9×9 Computer Go Server (CGOS[1]) since August 2006; and it won all the tournaments (9x9 and 13x13) on the international Kiseido Go Server[2] on October and November 2006.

This paper is organized as follows. Section II briefly introduces related work, assuming the reader's familiarity with the basics of Go. Section III describes MoGo, focussing on our contributions: the implementation of UCT in large sized search spaces, and the use of prior, pattern-based, knowledge to bias the Monte-Carlo evaluation. Experiment results are reported and discussed in Section IV. The paper concludes with some knowledge and computer-intensive perspectives for improving MoGo.

II. Previous Related Work

Our approach is based on the Monte-Carlo Go and multi-armed bandit problems, which we present respectively in Section II-A and II-B. UCT, which applies multi-armed bandit techniques to minimax tree search, is presented in Section II-C. We suppose minimax tree and alpha-beta search is well known for the reader.

A. Monte-Carlo Go

Monte-Carlo Go, first appeared in 1993 [3], has attracted more and more attention in the last years. Monte-Carlo Go has been surprisingly efficient, especially on 9×9 game; CrazyStone, developed by Rémi Coulom [8], a program using stochastic simulations with very little knowledge of Go, is the best known[3].

[1] `http://cgos.boardspace.net/`
[2] `http://www.weddslist.com/kgs/past/index.html`
[3] CrazyStone won the gold medal for the 9×9 Go game during the 11th Computer Olympiad at Turin 2006, beating several strong programs including GnuGo, Aya and GoIntellect.

Two principle methods in Monte-Carlo Go are also used in our program. First we evaluate Go board situations by simulating random games until the end of game, where the score could be calculated easily and precisely. Second we combine the Monte-Carlo evaluation with minimax tree search. We use the tree structure of CrazyStone [8] in our program.

Remark 1: We speak of a tree, in fact what we have is often an oriented graph. However, the terminology "tree" is widely used. As to the Graph History Interaction Problem (GHI) explained in [9], we ignore this problem considering it not very serious, especially compared to other difficulties in Computer-Go.

B. Bandit Problem

A K-armed bandit, is a simple machine learning problem based on an analogy with a traditional slot machine (one-armed bandit) but with more than one arm. When played, each arm provides a reward drawn from a distribution associated to that specific arm. The objective of the gambler is to maximize the collected reward sum through iterative plays[4]. It is classically assumed that the gambler has no initial knowledge about the arms, but through repeated trials, he can focus on the most rewarding arms.

The questions that arise in bandit problems are related to the problem of balancing reward maximization based on the knowledge already acquired and attempting new actions to further increase knowledge, which is known as the exploitation-exploration dilemma in reinforcement learning. Precisely, exploitation in bandit problems refers to select the current best arm according to the collected knowledge, while exploration refers to select the sub-optimal arms in order to gain more knowledge about them.

A K-armed bandit problem is defined by random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, where each i is the index of a gambling machine (i.e., the "arm" of a bandit). Successive plays of machine i yield rewards $X_{i,1}$,$X_{i,2}$,... which are independent and identically distributed according to a certain but unknown law with unknown expectation μ_i. Here independence holds also for rewards across machines; i.e., $X_{i,s}$ and $X_{j,t}$ are independent (probably not identically distributed) for each $1 \leq i < j \leq K$ and each $s,t \geq 1$. Algorithms choose the next machine to play depending on the obtained results of the previous plays. Let $T_i(n)$ be the number of times machine i has been played after the first n plays. Since the algorithm does not always make the best choice, its expected loss is studied. Then the regret after n plays is defined by

$$\mu^* n - \sum_{j=1}^{K} \mu_j E[T_j(n)] \quad \text{where} \quad \mu^* = \max_{1 \leq i \leq K} \mu_i$$

$E[\,]$ denotes expectation. In the work of Auer and Al. [6], a simple algorithm UCB1 is given, which ensures the optimal machine is played exponentially more often than any other machine uniformly when the rewards are in $[0,1]$. Note

$$\bar{X}_{i,s} = \frac{1}{s} \sum_{j=1}^{s} X_{i,j} \quad , \quad \bar{X}_i = \bar{X}_{i,T_i(n)} \,,$$

then we have:

Algorithm 1: Deterministic policy: UCB1

- Initialization: Play each machine once.
- Loop: Play machine j that maximizes $\bar{X}_j + \sqrt{\frac{2 \log n}{T_j(n)}}$, where n is the overall number of plays done so far.

One formula with better experimental results is suggested in [6]. Let

$$V_j(s) = \left(\frac{1}{s} \sum_{\gamma=1}^{s} X_{j,\gamma}^2 \right) - \bar{X}_{j,s}^2 + \sqrt{\frac{2 \log n}{s}}$$

be an estimated upper bound on the variance of machine j, then we have a new value to maximize:

$$\bar{X}_j + \sqrt{\frac{\log n}{T_j(n)} \min\{1/4, V_j(T_j(n))\}} \,. \qquad (1)$$

According to Auer and Al., the policy maximizing (1) named UCB1-TUNED, considering also the variance of the empirical value of each arms, performs substantially better than UCB1 in all his experiments. This corresponds to our early results and then we use always the policy UCB1-TUNED in our program[5].

C. UCT: UCB1 for Tree Search

UCT [7] is the extension of UCB1 [6] to minimax tree search. The idea is to consider each node as an independent bandit, with its child-nodes as independent arms. Instead of dealing with each node once iteratively, it plays sequences of bandits within limited time, each beginning from the root and ending at one leaf.

The algorithm UCT is defined in Table I[6]. The program continues playing one sequence each time, which is defined from line 1 to line 8. Line 9 to line 21 are the function using UCB1 for choosing one arm (one child-node in the UCT case). Line 15 ensures each arm be selected once before further exploration. Line 16 applies the formula of UCB1. After each sequence, the value of played arm of each bandit is updated[7] iteratively from the father-node of the leaf to the root by formula UCB1, described in function $updateValue$ from line 22 to line 29. Here the code deals with the minimax case. In general, the value of each node converges to the real max (min) value as the number of simulations increases.

In the problems of minimax tree search, what we are looking for is often the optimal branch at the root node. It is sometimes acceptable if one branch with a score near to the optimal one is found, especially when the depth of the tree is very large and the branching factor is big, like in Go, as it is often too difficult to find the optimal branch within short time.

[4]We will use "play an arm" when refering to general multi-armed problems, and "play a move" when refering to Go. In Go application, the "play" will not refer to a complete game but only one move.

[5]We will however say UCB1 for short.

[6]In order to be clear, the optimization is not discussed here.

[7]Here we use the original formula in Algorithm 1.

TABLE I

PSEUDOCODE OF UCT FOR MINIMAX TREE.

```
 1:  function playOneSequence(rootNode);
 2:    node[0] := rootNode; i = 0;
 3:    while(node[i] is not leaf) do
 4:      node[i+1] := descendByUCB1(node[i]);
 5:      i := i + 1;
 6:    end while ;
 7:    updateValue(node, -node[i].value);
 8:  end function;

 9:  function descendByUCB1(node)
10:    nb := 0;
11:    for i := 0 to node.childNode.size() - 1 do
12:      nb := nb + node.childNode[i].nb;
13:    end for;
14:    for i := 0 to node.childNode.size() - 1 do
15:      if node.childNode[i].nb = 0
           do v[i] := ∞;
16:      else v[i] := 1-node.childNode[i].value
         /node.childNode[i].nb
         +sqrt(2*log(nb)/(node.childNode[i].nb)
17:      end if;
18:    end for;
19:    index := argmax(v[j]);
20:    return node.childNode[index];
21:  end function;

22:  function updateValue(node,value)
23:    for i := node.size()-2 to 0 do
24:      node[i].value := node[i].value + value;
25:      node[i].nb := node[i].nb + 1;
26:      value := 1-value;
27:    end for;
28:  end function;
```

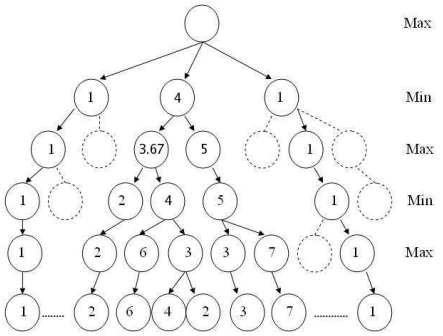

Fig. 1. UCT search. The shape of the tree enlarges asymmetrically. Only updated values ($node[i].value$) are shown for each visited nodes.

In this sense, UCT outperforms alpha-beta search. Indeed we can outlight three major advantages. First, it works in an anytime manner. We can stop at any moment the algorithm, and its performance can be somehow good. This is not the case of alpha-beta search. Figure 2 shows if we stop alpha-beta algorithm prematurely, some moves at first level has even not been explored. So the chosen move may be far from optimal. Of course iterative deepening can be used, and solve partially this problem. Still, the anytime property is stronger for UCT and it is easier to finely control time in UCT algorithm.

Second, UCT is robust as it automatically handles uncertainty in a smooth way. At each node, the computed value is the mean of the value for each child weighted by the frequency of visits. Then the value is a smoothed estimation of max, as the frequency of visits depends on the difference between the estimated values and the confidence of this estimates. Then, if one child-node has a much higher value than the others, and the estimate is good, this child-node will be explored much more often than the others, and then UCT selects most of the time the 'max' child node. However, if two child-nodes have a similar value, or a low confidence, then the value will be closer to an average.

Third, the tree grows in an asymmetric manner. It explores more deeply the good moves. What is more, this is achieved in an automatic manner. Figure 1 gives an example.

Figure 1 and Figure 2 compares clearly the explored tree of two algorithms within limited time. However, the theoretical analysis of UCT is in progress [10]. We just give some remarks on this aspect at the end of this section. It is obvious that the random variables involved in UCT are not identically distributed nor independent. This complicates the analysis of convergence. In fact we can define the bias for the arm i by:

$$\delta_{i,t} = \left| \mu_i^* - \frac{1}{t} \sum_{s=1}^{t} X_{i,s} \right| ,$$

where μ_i^* is the minimax value of this arm. It is clear that at leaf level $\delta_{i,t} = 0$. We can also prove that

$$\delta_{i,t} \leq K^D \frac{\log t}{t} ,$$

with K constant and D the depth of the arm (counted from the root down). This corresponds to the fact that the bias is amplified when passing from deep level to the root, which prevents the algorithm from finding quickly the optimal arm at the root node.

An advantage of UCT is that it adapts automatically to the 'real' depth. For each branch of the root, its 'real' depth is the depth from where $\delta_{i,t} = 0$ holds true. For these branches, the bias at the root is bounded by $K^d \frac{\log t}{t}$ with the real depth $d < D$. The values of these branches converging faster than the other, UCT spends more time on other interesting branches.

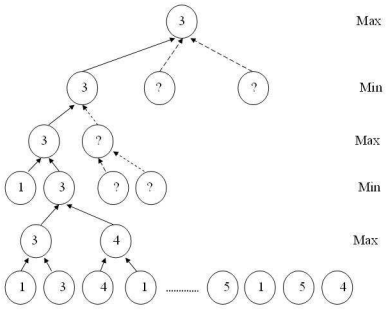

Fig. 2. Alpha-beta search with limited time. The nodes with '?' are not explored yet. This happens often during the large-sized tree search where entire search is impossible. Iterative deepening solves partially this problem.

TABLE II

PSEUDOCODE OF UCT FOR MOGO

```
1:  function playOneSequenceInMoGo(rootNode)
2:    node[0] := rootNode; i := 0;
3:    do
4:      node[i+1] := descendByUCB1(node[i]); i := i + 1;
5:    while node[i] is not first visited;
6:    createNode(node[i]);
7:    node[i].value := getValueByMC(node[i]);
8:    updateValue(node,-node[i].value);
9:  end function;
```

III. MAIN WORK

In this section we present our program MoGo using UCT algorithm. Section III-A presents our application of UCT. Then, considering two important aspects for having a strong Monte-Carlo program: the quality of simulations (then the estimation of score) and the depth of the tree, we show in the two following sections our corresponding improvements. Section III-B presents the sequence-like random simulation with patterns. Section III-C presents ideas for tree search pruning on large Go board. Section III-D presents the modification on the exploring order of non-visited nodes. At last, Section III-E presents parallelization.

A. Application of UCT for Computer-Go

MoGo contains mainly two parts, namely the tree search part and the random simulation part, as shown in Figure 3. Each node of the tree represents a Go board situation, with child-nodes representing next situations after corresponding move.

The application of UCT for Computer-Go is based on the hypothesis that each Go board situation is a bandit problem, where each legal move is an arm with unknown reward but of a certain distribution. We suppose that there are only two kinds of arms, the winning ones and the losing ones. We set respectively reward 1 and 0. We ignore the case of draw, which is too rare in Go.

In the tree search part, we use a parsimonious version of UCT by introducing the same dynamic tree structure as in CrazyStone [8] in order to economize memory. The tree is then created incrementally by adding one node after each simulation as explained in the following. This is different from the one presented in [7], and is more efficient because less nodes are created during simulations. In other words, only nodes visited more than twice are saved, which economizes largely the memory and accelerates the simulations. The pseudocode is given in Table II. Again we do not talk about optimization.

During each simulation, MoGo starts from the root of the tree that it saves in the memory. At each node, MoGo selects one move according to the UCB1 formula 1. MoGo then descends to the selected child node and selects a new move (still according to UCB1) until such a node has not yet been created in the tree. This part corresponds to the code from line 1 to line 5. The tree search part ends by creating this new node (in fact one leaf) in the tree. This is finished by $createNode$. Then MoGo calls the random simulation part, the corresponding function $getValueByMC$ at line 7, to give a score of the Go board at this leaf.

In the random simulation part, one random game is played from the corresponding Go board till the end, where score is calculated quickly and precisely according to the rules. The nodes visited during this random simulation are not saved. The random simulation done, the score received, MoGo updates the value at each node passed by the sequence of moves of this simulation[8].

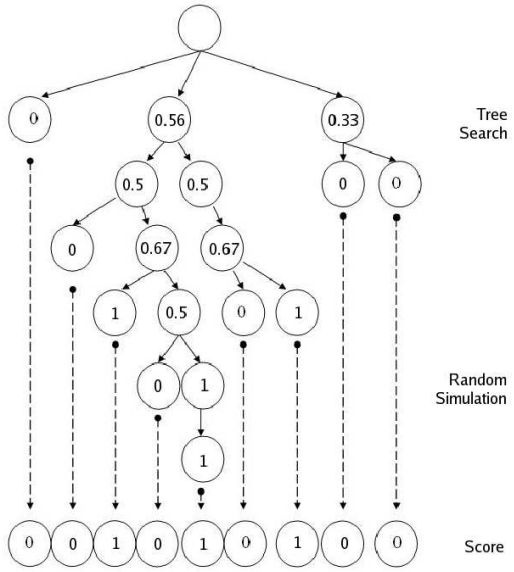

Fig. 3. MoGo contains the tree search part using UCT and the random simulation part giving scores. The numbers on the bottom correspond to the final score of the game (win/loss). The numbers in the nodes are the updated values of the nodes ($node[i].value$)

Remark 2: In the update of the score, we use the $0/1$ score instead of the territory score, since the former is much more robust. Then the real minimax value of each node should be either 0 or 1. In practice, however, UCT approximates each node by a weighted average value in $[0, 1]$. This value is usually considered as the probability of winning.

B. Improving simulation with domain-dependent knowledge

In this section we introduce our sequence-like random simulation with patterns. Its advantage is obvious compared with the classical random simulation, which we call pure random mode. In the following we talk about our improved random mode as well as our implementation of patterns.

In the random simulation part, it is very important to have clever simulations giving credible scores. Using some simple 3×3 patterns inspired by Indigo [11] (similar patterns can also be found in [12]), our random simulation is likely to have more meaningful sequences in random simulations than before, which has improved significantly the level of MoGo.

[8]It is possible to arrive at one end game situation during the tree search part. In this case, one score could be calculated immediately and there is no need to create the node nor to call the random simulation part.

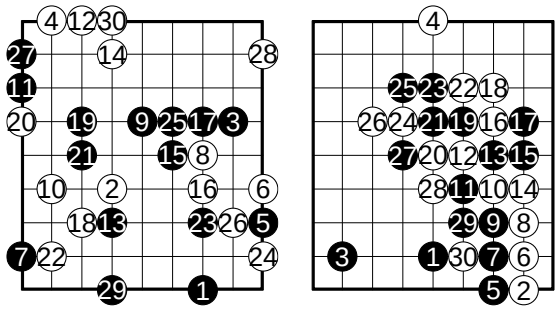

Fig. 4. Left: beginning of one random game simulated by pure random mode. Moves are sporadically played with little sense. Right: beginning of one random game simulated by the pattern-based random mode. From move 5 to move 29 one complicated sequence is generated.

Essentially, we use patterns to create meaningful sequences in simulations by finding local answers. The moves played are not necessarily globally better moves. It is not obvious that is more important to get better sequences rather that better moves to make the Monte-Carlo evaluation more accurate. However our experiments showed that the main improvement came from the use of local answers. If the same patterns are used to find interesting moves everywhere in the board instead of near the previous moves, the accuracy decreases. We believe that this claim is not obvious, and one of the main contribution of MoGo. We also don't use patterns for pruning in the tree. We have not investigated the more sophisticated patterns equipped by other programs like GnuGo.

In our pure random mode, legal moves are played on the Go board uniformly randomly, with few rules preventing the program from filling its own eyes. We also privilege the moves capturing some stones. On CGOS our first program using exactly this mode has achieved rank score 1647 ELO[9]. Currently the rank of MoGo is close to 2200 ELO.

Then, since we were not satisfied by the pure random simulations which gave meaningless games most of the time, local patterns are introduced in order to have some more reasonable moves during random simulations. Our patterns are defined as 3×3 intersections, centered on one free intersection, where one move is supposed to be played. Our patterns consist of several functions, testing if one move in such a local situation (3×3) is interesting. More precisely, we test if one move satisfies some classical forms in Go games, for example cut move, Hane move, etc.

Moreover, we look for interesting moves only around the last played move on the Go board. This is because that local interesting moves look more likely to be the answer moves of the last moves, and thus local sequence appears when several local interesting moves are tested and then played continuously in random simulations.

We describe briefly how the improved random mode generates moves. It first verifies whether the last played move is an Atari; if yes, and if the stones under Atari can be saved (in the sense that it can be saved by capturing stones or increasing liberties), it chooses one saving move randomly; otherwise it looks for interesting moves in the 8 positions around the last played move and plays one randomly if there is any; otherwise it looks for the moves capturing stones on the Go board, plays one if there is any. At last, if still no move is found, it plays one move randomly on the Go board. Surely, the code of MoGo is actually complicated in details, with many small functions equipped with hand-coded Go knowledges. However, we believe the main frame given here is the most essential to have sequence-like simulations.

Figure 4 shows the first 30 moves of two random games using different modes. Moves generated by the improved random mode are obviously much more meaningful.

[9]The ELO (`http://en.wikipedia.org/wiki/Elo_rating_system`) is a rating system where the probability of winning is related to the difference between the ranks.

We now give the detailed information on our patterns. The patterns are 3×3 intersections centered on an empty position, say p, where is supposed to play the next move. Each pattern is a boolean function, answering the question whether the next move playing on p is an interesting move. True is returned (when pattern is matched), if and only if the state of each position on the Go board is the same as the one on the corresponding position of the pattern, or there is a cross on the corresponding position (which means the situation of this position is ignored). Normally there is no constraint on the color of the next move (one move good for black is supposed to be also good for white). Some special cases are explained when mentioned. The symmetry, rotations and exchange of stone colors of patterns are considered. Moves are tested by patterns only if they are neither illegal moves nor self-Atari moves.

We have tried several patterns during the development of MoGo and implemented finally the ones shown in Figure 5, 6, 7 and 8, where the position with a square is where the next move is supposed to be played. We used hand-coded patterns in our implementation. However, it will be more interesting if this can be achieved by a learning system. Another approach using Bayesian generation can be found in Bouzy's work [13].

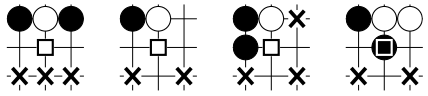

Fig. 5. Patterns for Hane. True is returned if any pattern is matched. In the right one, a square on a black stone means true is returned if and only if the eight positions around are matched and it is black to play.

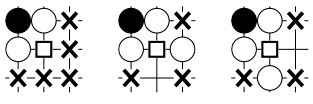

Fig. 6. Patterns for Cut1. The Cut1 Move Pattern consists of three patterns. True is returned when the first pattern is matched and the next two are not matched.

Remark 3: We believe that it is not always better to have more 'good' patterns in the random modes, meanwhile what

Fig. 7. Pattern for Cut2. True is returned when the 6 upper positions are matched and the 3 bottom positions are not white.

Fig. 8. Patterns for moves on the Go board side. True is returned if any pattern is matched. In the three right ones, a square on a black (resp. white) stone means true is returned if and only if the positions around are matched and it is black (resp. white) to play.

is more important is whether the random simulation can have some meaningful sequences often. This claim needs more experiments.
The Table III shows clearly how patterns improve the overall performance.

TABLE III

DIFFERENT MODES WITH 70000 RANDOM SIMULATIONS/MOVE IN 9X9.

Random mode	Win. Rate for B. Games	Win. rate for W. Games	Total Win. Rate
Pure	46% (250)	36% (250)	41.2% $\pm$ 2.2%
Sequence-like	77% (400)	82% (400)	80% $\pm$ 1.4%

C. UCT with pruning ideas

In this section we show our ideas (heuristics) to reduce the huge tree size, which makes MoGo relatively strong on large Go board. Thus we gain a larger local depth in the search tree by losing the global view of UCT. Obviously pruning heuristics may lead to a sub-optimal solution. First we define group by Go knowledge to reduce the branching factor in tree search. Then zone division is derived from group, which helps to have a more precise score. We use group and zone mode for 13×13 and 19×19 Go board. Figure 9 will give one example.

Remark 4: As we are not very experienced for Go-knowledge-based programming and we had little time working on it, we believe other programs like GnuGo and AyaGo, or Monte-Carlo programs have more clever pruning techniques. Some other techniques are mentioned in [14][15]. Due to the space limitation, we do not give the detailed pseudo code of this part. However, our experimental results of combining UCT with pruning techniques are already encouraging.

First we define one group as a set of strings and free intersections on a Go board according to certain Go knowledge, which gathers for example one big living group and its close enemies. We have implemented Common Fate Graph (CFG) [16] in our program to help the calculation of groups. The method starts from one string and recursively adds close empty intersections and strings close to these empty intersections until no more close strings are found within a distance controled by a parameter.

In group mode, in the tree search part we search only the moves in the group instead of all over the Go board. In random simulation part there is no more such restriction. Using groups, we reduce the branching factor to less than 50 at the opening period. Then, depth of MoGo's tree could be around 7-8 on large Go board. Table IV shows MoGo becomes competitive on 13×13 Go board by using group pruning technique. However, sophisticated pruning techniques are undoubtedly necessary to improve the level of Computer-Go programs.

TABLE IV

MOGO WITH 70000 SIMULATIONS PER MOVE, ON 13×13 GO BOARD, USING OR NOT THE GROUP MODE HEURISTIC AGAINST GNUGO 3.6 LEVEL 0 (GG 0) OR 8 (GG 8).

Opponents	Win. Rate for B. Games	Win. rate for W. Games	Total Win. Rate
No group vs GG 0	53.2%(216)	51.8% (216)	52% $\pm$ 2.4%
No group vs GG 8	24.2%(300)	30% (300)	27% $\pm$ 1.8%
group vs GG 0	67.5% (80)	61.2% (80)	64.3% $\pm$ 3.7%
group vs GG 8	51.9% (160)	60% (160)	56% $\pm$ 2.7%

As explained above, group mode limits the selection of moves in the tree search part. It has however no restriction on the random simulation. As the accuracy of the simulations becomes lower as the game length increases, we tried to generate the random moves only in a certain zone instead of on the whole Go board. The zones were defined using the groups presented above. However due to space limitations, and as the zones are no more used in the current MoGo player, we do not describe them further. Interesting future research directions could be to define properly zones to limit the simulations lengths.

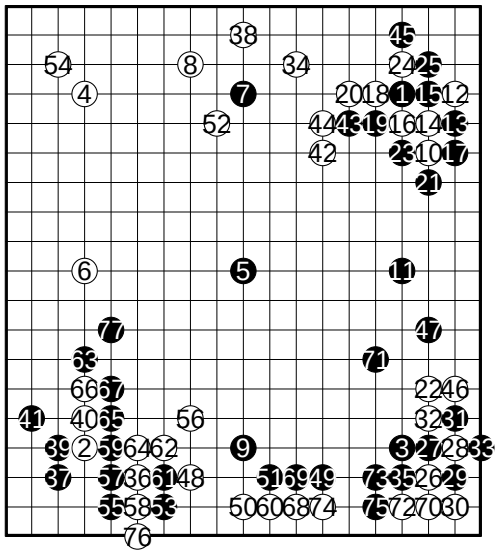

Fig. 9. The opening of one game between MoGo and Indigo in the 18th KGS Computer Go Tournament. MoGo (Black) was in advantage at the beginning of the game, however it lost the game at the end.

D. Modification of exploring order for non-visited nodes

UCT works very well when the node is frequently visited as the trade-off between exploration and exploitation is well handled by UCB1 formula. However, for the nodes far from the root, whose number of simulations is very small, UCT tends to be too much exploratory. This is due to the fact that all the possible moves in one position are supposed to be explored before using the UCB1 formula. Thus, the values associated to moves in deep nodes are not meaningful, since the child-nodes of these nodes are not all explored yet and, sometimes even worse, the visited ones are selected in fixed order. This can leads to bad predicted sequences.

Two modifications are made to have a better order.

First-play urgency: UCB1 algorithm begins by exploring each arm once, before using the formula (1). This can sometimes be unefficient especially if the number of trials is not large comparing to the number of arms. This is the case for numerous nodes in the tree (number of visits is small comparing to the number of moves). For example if an arm keeps returning 1 (win), there is no good reason to explore other arms. We have set a fixed constant named first-play urgency (FPU) in the algorithm. For each move, we name its urgency by the value of formula (1). The urgency value is set to the value of FPU (FPU is $+\infty$ by default) for each legal move before first visit (see line 15 in Table I). Any node, after being visited at least once, has its urgency updated according to UCB1 formula. We play the move with the highest urgency. Thus, FPU $= +\infty$ ensures the exploration of each move once before further exploitation of any previously visited move. On the other way, smaller FPU ensures earlier exploitations if the first simulations lead to an urgency larger than FPU (in this case the other unvisited nodes are not selected). This improved the level of MoGo according to our experiment as shown in Table VII.

Use information of the parents: One assumption that can be made in go game is that given a situation, good moves may sometimes still be good ones on the following move. When we encounter a new situation, instead of exploring each move m in any order, we can use the value estimation of m in an earlier position to choose a better order. We typically use the estimated values of the grandfather of the node. We believe this helps MoGo on the large Go board, however we do not have enough experiments to claim significant results.

E. Parallelization

As UCT scales well with time, we made MoGo run on a multi-processors machine with shared memory. The modifications to the algorithm are quite straightforward. All the processors share the same tree, and the access to the tree is locked by mutexes. As UCT is deterministic, all the threads could take exactly the same path in the tree, except for the leaf. The behavior of the multithreaded UCT as presented here is then different from the monothreaded UCT. Two experiments has then to be done. First, we can compare the level of MoGo using the monothreaded or the multithreaded algorithms while allowing the same number of simulations per move. All such experiments showed non significant differents in the play level[10]. Second, we can compare the level using the same time per move (the multithreaded version will then make more simulations per move). As UCT benefits from the computational power increase, the multithreaded UCT is efficient ($+100$ ELO on CGOS with 4 processors).

IV. Results

We list in this section several experiment results who reflect characteristics of the algorithm. All the tests are made by letting MoGo play against GnuGo 3.6 with default mode. Komi are set to 7.5 points. In the tables, the winning rates when MoGo plays black and white are given with the number of games played in each color (in parentheses). The number given after the $\pm$ is the standard deviation.

A. Dependence of Time

The performance of our program depends on the given time (equally the number of simulations) for each move. Table V shows its level improves as this number increases. The outstanding performance of MoGo on double-processors and quadri-processors also supports this claim.

TABLE V

Pure random mode with different times.

Seconds per move	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
5	26% $\pm$ 6% (50)	26% $\pm$ 6% (50)	26% $\pm$ 4.3%
20	41% $\pm$ 3% (250)	42% $\pm$ 3% (250)	41.8% $\pm$ 2.2%
60	53% $\pm$ 3.5% (200)	50% $\pm$ 3.5% (200)	51.5% $\pm$ 2.5%

B. Parametrized UCT

We parametrize the UCT implemented in our program by two new parameters, namely p and FPU. First we add one coefficient p to formula UCB1-TUNED (1), which by default is 1. This leads to the following formula: choose j that maximizes:

$$\bar{X}_j + p\sqrt{\frac{\log n}{T_j(n)}\min\{1/4, V_j(n_j)\}}$$

p decides the balance between exploration and exploitation. To be precise, the smaller the p is, the deeper the tree is explored. According to our experiment shown in Table VI, UCB1-TUNED is almost optimal in this sense.

The second is the first-play urgency (FPU) as explained in Section III-D. Some results are shown in Table VII. We believe that changing exploring order of non-visited nodes can bring further improvement.

C. Results On CGOS

MoGo is ranked as the first program on 9×9 Computer Go Server since August 2006.

[10]we had only access to a 4 processors computer, the behavior can be very different with many more processors.

TABLE VI

COEFFICIENT p DECIDES THE BALANCE BETWEEN EXPLORATION AND EXPLOITATION. (PURE RANDOM MODE)

p	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
0.05	2% ± 2% (50)	4% ± 2.5% (50)	3% ± 1.7%
0.55	30% ± 6.5% (50)	36% ± 6.5% (50)	33% ± 4.7%
0.80	33% ± 4.5% (100)	39% ± 5% (100)	36% ± 3.3%
1.0	40% ± 4% (150)	38% ± 4% (150)	39% ± 2.8%
1.1	39% ± 4% (150)	41% ± 4% (150)	40% ± 2.8%
1.2	40% ± 4% (150)	44% ± 4% (150)	42% ± 2.9%
1.5	30% ± 6.5% (50)	26% ± 6% (50)	28% ± 4.5%
3.0	36% ± 6.5% (50)	24% ± 6% (50)	30% ± 4.5%
6.0	22% ± 5.5% (50)	18% ± 5% (50)	20% ± 4%

TABLE VII

INFLUENCE OF FPU (70000 SIMULATIONS/MOVE).

FPU	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
1.4	37% ± 4.5% (100)	38% ± 5% (100)	37.5% ± 3.5%
1.2	46% ± 5% (100)	36% ± 5% (100)	41% ± 3.5%
1.1	45% ± 3% (250)	41% ± 3% (250)	43.4% ± 2.2%
1.0	49% ± 3% (300)	42% ± 3% (300)	45% ± 2%
0.9	47% ± 4% (150)	32% ± 4% (150)	40% ± 2.8%
0.8	40% ± 7% (50)	32% ± 6.5% (50)	36% ± 4.8%

V. CONCLUSION

The success of MoGo shows the efficiency of UCT compared to alpha-beta search in the sense that nodes are automatically studied with better order, especially in the case when search time is too limited. We have discussed the advantages of UCT relevant to Computer-Go. It is worthy to mention that a growing number of top level Go programs now use UCT.

We have discussed improvements that could be made to UCT algorithm. In particular, UCT does not help to choose a good ordering for non-visited moves, nor is it so effective for little explored moves. We proposed some methods adjusting the first-play urgency to solve this problem, and futher improvements are expected in this direction.

We have proposed the pattern-based (sequence-like) simulation which has improved significantly the level of MoGo. We implemented 3×3 patterns in random simulations in order to have more meaningful sequences. We believe that significant improvements can still be made in this direction, for example by using larger patterns, perhaps automatically generated ones. It is also possible to implement some sequence-forced patterns to improve the quality of simulations.

We have also shown the possibilities of combining UCT with pruning techniques in order to have a strong program on large Go board. Having had some encouraging results, we believe firmly further improvements in this direction.

A straightforward parallelization of UCT on shared-memory computer is made and has given some positive results. Parallelization on a cluster of computers can be interesting but the way to achieve that is yet to be found.

ACKNOWLEDGMENTS

We would like to thank Rémi Munos, Olivier Teytaud, Pierre-Arnaud Coquelin for the help during the development of MoGo. We specially thank Rémi Coulom for sharing his experiences of programming CrazyStone. We also appreciate the discussions on the Computer-Go mailing list.

REFERENCES

[1] B. Bouzy and T. Cazenave, "Computer go: An AI oriented survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.
[2] M. Newborn, *Computer Chess Comes of Age*. Springer-Verlag, 1996.
[3] B. Bruegmann, "Monte carlo go," 1993.
[4] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press., 1998.
[5] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "Gambling in a rigged casino: the adversarial multi-armed bandit problem," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 322–331.
[6] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2/3, pp. 235–256, 2002.
[7] L. Kocsis and C. Szepesvari, "Bandit based monte-carlo planning," in *15th European Conference on Machine Learning (ECML)*, 2006, pp. 282–293.
[8] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006, To appear.
[9] A. Kishimoto and M. Müller, "A general solution to the graph history interaction problem," *Nineteenth National Conference on Artificial Intelligence (AAAI 2004), San jose, CA*, pp. 644–649, 2004.
[10] L. Kocsis, C. Szepesvri, and J. Willemson, "Improved monte-carlo search," vol. working paper, 2006.
[11] B. Bouzy, "Associating domain-dependent knowledge and monte carlo approaches within a go program," *Information Sciences, Heuristic Search and Computer Game Playing IV, Edited by K. Chen*, no. 4, pp. 247–257, 2005.
[12] L. Ralaivola, L. Wu, and P. Baldi, "Svm and pattern-enriched common fate graphs for the game of go," *ESANN 2005*, pp. 485–490, 2005.
[13] B. Bouzy and G. Chaslot, "Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go," *In G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pp. 176–181, 2005.
[14] T. Cazenave, "Abstract proof search," *Computers and Games, Hamamatsu, 2000*, pp. 39–54, 2000.
[15] T. Cazenave and B. Helmstetter, "Combining tactical search and monte-carlo in the game of go," *IEEE CIG 2005*, pp. 171–175, 2005.
[16] T. Graepel, M. Goutrié, M. Krüger, and R. Herbrich, "Learning on graphs in the game of go," *Lecture Notes in Computer Science*, vol. 2130, pp. 347–352, 2001.