Exploration exploitation in Go: UCT for Monte-Carlo Go

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

(**)Center of Applied Mathematics
Ecole Polytechnique, Palaiseau, France
yizao.wang@polytechnique.edu

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 modifications of UCT for Go application, among which efficient memory management, parametrization, ordering of non-visited nodes and parallelization. MoGo is now a top-level Computer-Go program on 9×9 Go board.

1 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 [3]. 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 Goban ranges from 9×9 to 19×19 (against 8×8 for the Chess board); the number of potential moves is a few hundreds against a few dozens 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 [9]) failed to provide good enough Go strategies.

Recent progress has been done regarding the evaluation of Go positions, based on Monte-Carlo approaches [4] (more on this in section 4.1). 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 [10], to enforce the exploration *versus* exploitation (EvE) search strategy required for a good Go player.

Another EvE setting from Game Theory, the multi-armed bandit problem, is thus considered in this paper. The multi-armed bandit problem models the gambler, choosing the next machine to play based on the past selections and rewards, in order to maximize the total reward [2]. The UCB1 algorithm proposed by Auer et al. in the multi-armed bandit frame-

work [1] was recently extended to tree-structured search space by Kocsys et al. (algorithm UCT) [7].

The MoGo player presented in this paper combines the UCT algorithm with the Monte-Carlo evaluation function. The improvements in the Monte-Carlo evaluation function are not presented here. Instead, we focus on several algorithmic issues: dynamic tree structure [5], parametrization of UCT, parallelized implementation. 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 two tournaments (9x9 and 13x13) on the international Kiseido Go Server[2].

Our approach is based on the Monte-Carlo Go and multi-armed bandit problems. The interested reader can refer to the appendix respectively in Section 4.1 and 4.2. UCT, which applies multi-armed bandit techniques to minimax tree search, is presented for completeness in appendix Section 4.3. We there briefly discuss some reasons of the efficiency of UCT in Go. Section 2 describes MoGo, focussing on our contribution about the implementation of UCT in large sized search spaces: an efficient memory management, solutions about better ordering of non-visited nodes, and a simple parallelization of UCT. Several significant experimental results are presented in appendix.

2 Main Work

In this section we present our program MoGo using UCT algorithm. Section 2.1 presents our application of UCT. Section 2.2 presents the modification on the exploring order of non-visited nodes. At last, Section 2.3 presents parallelization. We don't discuss the patterns used in the random simulations, focusing on the exploration-exploitation dilemna. All the results presented here are performed with classical random simulations, what we call "pure random mode". Much better results are obtained with patterns, but it is not the point here.

2.1 Application of UCT for Computer-Go

MoGo contains mainly two parts, namely the tree search part and the random simulation part. 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 [5] 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 1.

During each simulation, MoGo starts from the root of the tree that it saves in memory. At each node, MoGo selects one move according to the UCB1 formula [1] (see appendix 4.2 formula (1) for more details). 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.

[1] `http://cgos.boardspace.net/`

[2] http://www.weddslist.com/kgs/past/19/index.html

```
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;
```

Table 1: Pseudocode of UCT for MoGo

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 of Go. The nodes visited during this random simulation are not saved. The random simulation done, the score received, MoGo updates the value at each node of the tree visited by the sequence of moves before the random simulation part.

Remark 1 *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.*

2.2 Modification of exploring order for rarely-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, which are visited rarely, 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. This assumption is good when the number of options is low, but this is definitely not the case in Go. 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.

UCT does not give any solution to choose between moves that are not explored yet. We have set a first-play urgency (FPU) in the algorithm. For each move, we define its urgency by the value of formula named UCB1-TUNED in [1] (see appendix 4.2 formula (1) for more details) if the node has been visited. The FPU was by default in our first experiments (for unvisited nodes) set to 10000 for each legal move. Any node, after being visited at least once, has its urgency updated according to UCB1 formula. Thus, consistently with the UCB1 algorithm, the FPU 10000 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 give positive results. Smaller FPU improved the level of MoGo according to our experiment as shown in Table 4. This permits UCT to make more exploitations for deep nodes.

2.3 Parallelization

As UCT strongly improves when computation time increases, we made MoGo run on a multi-processor 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, but experimentaly, with the same number of simulations, there is very little difference between the two results[3]. Then, as UCT benefits from the computational power increase, the multithreaded UCT is efficient.

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

3 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 of very limited search time. We have discussed the advantages of UCT relevant to Computer-Go. It is worth mentionning that since MoGo, 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 rarely explored moves. We proposed some methods adjusting the first-play urgency to solve this problem, and futher improvements are expected 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.

It worth mentionning that the multi-armed bandit problem we tackled is non-stationnary as adding child nodes changes the probability of winning for each move. Indeed, without child node, the probability of winning for each move is the probability of the random simulation. With a growing number of child nodes, the probability for a move depends on the predicted sequence in the tree. However, each attempt to deal with this non-stationnarity has given weaker results. We believe that the random simulations gives very noisy results, and the attempts for dealing with non-stationnarity increase the effect of noise.

Acknowledgments

We would like to thank Rémi Munos, Olivier Teytaud and 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] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.

[2] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert 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*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[3] Bruno Bouzy and Tristan Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[4] B. Bruegmann. Monte carlo go. 1993.

[5] Rémi 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.

[6] Akihiro Kishimoto and Martin Müller. A general solution to the graph history interaction problem. *Nineteenth National Conference on Artificial Intelligence (AAAI 2004), San jose, CA*, pages 644–649, 2004.

[7] L Kocsis and Cs Szepesvari. Bandit based monte-carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[8] L. Kocsis, Cs. Szepesvri, and J. Willemson. Improved monte-carlo search. working paper, 2006.

[9] Monty Newborn. *Computer Chess Comes of Age*. Springer-Verlag, 1996.

[10] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. MIT Press., Cambridge, MA, 1998.

4 Appendix

4.1 Monte Carlo Go

Monte Carlo Go, first appeared in 1993 [4], 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 [5], a program using stochastic simulations with very little knowledge of Go, is the best known[4].

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 [5] in our program.

Remark 2 *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 [6], we ignore this problem considering it not very serious, especially compared to other difficulties in Computer-Go.*

4.2 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. 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. [1], 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:

[4] 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.

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 [1]. 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:

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

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].

4.3 UCT: UCB1 for Tree Search

UCT is the extension of UCB1 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.

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.

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.

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. However, the theoretical analysis of UCT is in progress [8]. 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

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

[5]We will however say UCB1 for short.

with K constant and D the depth of the arm. 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.

5 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, as in the current tournament.

5.1 Dependence of Time

The performance of our program depends on the given time (equally the number of simulations) for each move. Table 2 shows its level improves as this number increases. The outstanding performance of MoGo on double-processors and quadri-processors also supports this claim.

Seconds per move	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
5	13/50	13/50	26%
20	103/250	106/250	41.8%
60	107/200	99/200	51.5%

Table 2: Pure random mode with different times.

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 3, UCB1-TUNED is almost optimal in this sense.

The second is the first-play urgency (FPU) as explained in 2.2. Some results are shown in Table 4. We believe that changing exploring order of non-visited nodes can bring further improvement. We finally use 1.1 as the initiate FPU for MoGo on CGOS.

Results on competitive events

Since July 2006, MoGo has been playing on 9×9 Computer Go Server (`http://cgos.boardspace.net/9x9.html`). It is ranked as the first program since August and won two tournaments (9x9 and 13x13) on Kgs (`http://www.weddslist.com/kgs/past/index.html`).

p	Winning Rate for Black Games	Winning rate for White Games	Total Winning Rate
0.05	1/50	2/50	3%
0.55	15/50	18/50	33%
0.80	17/50	20/50	37%
1.0	18/50	21/50	39%
1.1	40/100	40/100	40%
1.2	60/150	66/150	42%
1.5	15/50	13/50	28%
3.0	18/50	12/50	30%
6.0	11/50	9/50	20%

Table 3: Coefficient p decides the balance between exploration and exploitation. (Pure random Mode)

Simulations per move	FPU	Winning rate for Black Games	Winning rate for White Games	Total Winning Rate
70000	1.4	20/50	19/50	39%
70000	1.2	23/50	17/50	40%
70000	1.1	114/250	103/250	43.4%
70000	1.0	146/300	127/300	45.5%
70000	0.9	71/150	49/150	40%
70000	0.8	20/50	16/50	36%

Table 4: Influence of first-play urgency (FPU).