SuperGo: The Manual

Linpeng Tang, Qin Liu, Jiayuan Ma, Jiajun Shen

June 22, 2011

Abstract

1 Monte-Carlo Method

As the search space of Go is too large (the game tree is in the order of 10^{360} for alpha-beta tree search [2], one major alternative to using hand-coded knowledge and searches is the use of Monte-Carlo methods.

In the Monte-Carlo method, when we need to evaluate the current state of a Go board (say, we need to estimate whether Black is more likely to win than White), we simply simulate "random" plays on the board and see what happens when the simulation of the game ends. Alpha-beta tree search doesn't work well for computer Go, because on the one hand it is impossible to search the whole tree until it reaches the terminal state, and on the other hand if we only search for several steps the evaluation may be very inaccurate. In Go a good move may not seen to be useful until many steps later. The Monte-Carlo method, however, overcomes this difficulty because it simulates to the end of the game, when the state can be easily evaluated.

The Monte-Carlo method also has the advantage that it requires little hand-coded domain knowledge, and is easy to code. Compared to pattern matching, it also can make good use of the increasing power of modern computers. Actually, it is showed that with the growth of simulation times, the win-rate of a Monte-Carlo based computer Go program against a traditional program grows accordingly[4]. And it is proven theoretically that when the number of simulations approaches infinity, Monte-Carlo methods will eventually give a accurate evaluation of the board (as the exhaustive alpha-beta tree search)[8].

1.1 Monte-Carlo simulation

It is said in [3] that the design of a good Monte-Carlo simulator is a dark art. A good Monte-Carlo simulator should give an accurate evaluation of the current state of game as fast as possible. This often requires a large amount of simulations, so the simulator should be very fast (SuperGo can do about 7000 simulations in 8 seconds in one 2GHz core). The simulator should not be totally random, i.e. choose randomly from all the legal moves. Instead, we should think of the simulator as simulating what two simple-minded players will do in a game—they will capture stones when they can, save stones from being captured when they can, or do some simple pattern matching. Curiously, it is said in in [3] that a complicated simulator, even if given enough time, may not give a good result. For example, even if we use GNU Go as the simulator and simulate as many times as a simple simulator, the result of the evaluation may not be as accurate as the simple simulator. This is a curious property. It shows that the simulator must seek a balance between exploration and exploitation, between simplicity (randomness) and sophistication. SuperGo's simulator largely borrows the methods of Fuego[7], employing some greedy heuristics (like Nakade, Atari capture and Atari defense) and local pattern matching.

1.2 Upper Confidence Tree Search

UCT (Upper Confidence Tree) Search, originally proposed in [8], is a general framework for planning with Monte-Carlo simulations. It seeks a balance between exploration and exploitation. Consider the game tree in computer Go, every node typically has 80 children for a 13×13 board, and the game tree typically has a height of more than 100. Now suppose we have searched some part of the tree, what should we do next? There are two options, the first is to pick the best result we have met so far and keep expanding the path (exploitation); the second is to pick a new path and start over (exploration). Note there are problems with both approaches: if we only focus on the best result we have achieved so far, we may be trapped in a local minimum and miss other good results, and if we spread our searches among all the paths, since there are so many paths, we can not expand each path deep enough and the evaluation may be inaccurate after all. UCT Search addresses this problem by combining the two approaches, it gives each node a UCT bound [7]:

$$\text{UCT Bound} = x_j + c\sqrt{\frac{\log n}{T_j(n)}} = \text{Estimated Move Value} + \text{UCT Bias} \tag{1.1}$$

- x_j = reward for move j = weighted mean of move value and RAVE value
- j = move index
- n = #times father node visited
- $T_j(n)$ = #times move j has been played
- C = appropriate constant (default is 0.7 in Fuego)

And when expanding a path, we choose the child of a parent node with the largest of UCT bound.

Intuitively, the UCT bound consists of two parts, the Estimated Move Value represents the exploitation part, and the UCT Bias represents the exploration part. We will favor those nodes that has high estimated move values and has been visited few times compared to its parent.

1.3 Rapid Action Value Estimation

We need some heuristics to pick the most hopeful moves from the large number of possible moves of a given game state. RAVE (Rapid Action Value Estimation) [6] provides a good way for doing that, using information generated in the Monte-Carlo simulation. The basic observation of RAVE is that is a move several steps later contributes to a win in the current state, then it may benefit us to play it now as well. Now given a sequence of simulated moves $x_0, x_1, \ldots x_{2m-1}$, and suppose that x_{2k} is played by Black and x_{2k+1} is played by White. Suppose in this simulation Black wins. Then we will add some positive rating to those nodes of Black Moves that have the same moves with $x_0, x_2, \ldots, x_{2m-2}$ (showing that these moves may be desirable), and add some negative rating to those nodes of White Moves that have the same moves with $x_1, x_3, \ldots, x_{2m-1}$ (showing that these moves maybe undesirable). Further more, the larger i is, the less impact x_i will have on the corresponding nodes.

1.4 The UCT Search Framework

Now we are ready to state the UCT Search framework for computer Go. We wish to expand the whole game tree in the memory, with the current state being the root node of the tree. But as this is impossible for Go, we only store some nodes near to the root in memory, and store four values in each node: visit-count, visit-rating, rave-count, rave-rating.

- visit-count: how many times this node has been visited in a simulation path.

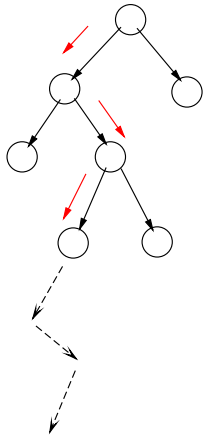

Figure 1: A sample game tree in during a UCT search. The red dashed arrow-headed lines represents a search path, including a in-tree part and a play-out simulation part

- visit-rating: the win-rate of the Black player of all simulations visiting this node.
- rave-count: the weight of the RAVE heuristics.
- rave-rating: the value of the RAVE heuristics.

Every time we we start a search from the root of the tree, and the search consists of two phases: in-tree phase and play-out phase. During the in-tree phase, we select child of the current node by the heuristics of UCT and RAVE combined (note that however in every step the corresponding player will choose a move that benefits him best and does most damage to his opponent). When we reach a terminal node of the tree, we check whether this node represents the end of a game. If it does, we directly update the four parameters above of the nodes in the tree. Otherwise, we start a simulation, evaluate the result of the simulation and then update the statistics.

1.5 Engineering and Some Innovations

1.5.1 Parallel Search Infrastructure

In SuperGo we implemented a simple UCT Search Go framework, equipped with RAVE heuristics a greedy simulator. As more simulations usually results in a more accurate evaluation, SuperGo supports multi-threaded simulations. As the game-tree is a public resource, it should be protected with a lock. [4] also proposes a lock-free infrastructure that can improve the number of simulations at the cost of some minor inconsistencies, and that structure was employed in Fuego. In our experiments, SuperGo can do about 7000 simulations in one core. And with two threads running one two cores, we can double the number.

1.5.2 Combining UCT Simulations with Expert Knowledge

With our observation, UCT search with Monte-Carlo simulation results in a AI player with great overall understanding of the board, but is usually weak in some local, strategic moves. To compensate this, we add expert knowledge with the statistics generated by UCT search when we generate our next move for the game. Originally, after many times of simulation, the AI will select the move corresponding to the child of the tree root with the highest visit-value. Now we multiply visit-value by an additional coefficient, representing how the expert knowledge. For example, we may give a capture move a 1.5 coefficient, and give a cut move a 1.2 coefficient.

1.5.3 Different Strategies for End Game Phase

Monte-Carlo simulation can be quite weak in the open game phase and end game phase. In the open game phase, the visit-rating and visit-count of most moves are similar, because after so many steps to the end of the game, the influence of the moves at the open game is too small to be detect by the Monte-Carlo simulator. In the end game phase, since the the win/loss state is already determined, the simulator often gives a win-rate of of near 1 (indicating that Black is very likely to win) or near 0 (indicating that White is very likely to win). So this would give little hit about what move should be executed next. To cope with this problem, in the end game phase (when there are less than 60 possible moves for a player), we give more weight to those wins with a better results (wins more area). This improvement will make SuperGo more rational at the end of the game, like trying to occupy more area at the boundary.

1.5.4 Dynamic Komi

With our observation, the Monte-Carlo method suffers from the problem of being weak then it is in large advantage of in large disadvantage. Because in either of these two situations, the win-rate of every moves will be very similar no matter what move the AI plays. To cope with this problem, we can give those results with large win (loss) higher reward (punishment). This approach, however, is theoretically unsound, and doesn't give good results in experiments. We take a different approach by changing the Komi dynamically. That is, we record the most recent K (say $K = 8000$) evaluations and update the Komi with the median of the recent evaluation results after every K simulations. In this way, the win-rate of the simulations will vary around 0.5 no matter whether the AI is in advantage or disadvantage, and the Monte-Carlo simulations can always give us valuable information about what is the best move next.

In our experiment, this technique seems to make SuperGo more reliable, and can greatly improvement its performance against our competitors.

2 Improving Monte-Carlo Simulations

As introduced in the last section, we use the result of Monte-Carlo (MC) simulations to evaluate the probability of winning of nodes in the MC tree search. In this section, we combine some expert knowledge to help us better estimate the probability.

2.1 Efficient board representation

In order to run more MC simulations in limited time, we need more efficient board representation. The class *GoBoard* defines a Go board that implements the rules of Go and provides a lot of helper functions to get blocks, liberties, adjacent blocks, and so on. We use a 1-D array to represent the board which is faster than the normal 2-D representation. To accelerate the operations of play a move, we need to maintain a data structure that stores the information of blocks in the current board and their liberties.

The class *GoUctBoard* is optimized for Monte-Carlo simulations. In contrast to class *GoBoard*, this board makes certain assumptions that are usually true for Monte Carlo simulations for better efficiency:

- No undo
- Alternating play
- Simple-Ko rule
- Suicide not allowed

2.2 UCT Patterns Matching

Human players conclude some similar moves that are often important and can lead to win. To combine this knowledge, we implement patterns matching in the class *UCTPatterns*.

Our hard-coded patterns has evolved from the the one originally used by MoGo [5]. At the highest priority levels, capture moves, atari-defense moves, and moves matching a small set of hand-selected 3×3 "MoGo" patterns are chosen if they are near the last move played on the board. If no move was selected so far, a global capture moves are attempted next. Finally, a move is selected randomly among all legal moves on the board.

2.3 The 'Nakade' Problem

Nakade refers to a situation in which a group has a single large internal, enclosed space that can be made into two eyes by the right move — or prevented from doing so by an enemy move. When the group is dead, the MC simulator which plays random moves sometimes estimates that it lives with a high probability. Therefore, the tree will not grow in the direction of right move.

This will lead to a false estimate of the probability of winning. To reflect the effect of a nakade, we use the below algorithm[3]: if a contiguous set of exactly 3 free locations is surrounded by stones from the opponent, then we play at the center (the vital point) of this "hole".

2.4 Unconditional Live

A set of chains of a player is said to be pass-alive if none of the chains in this set can be captured even if the player always passes. However, the above definition can be tedious to apply, because the required search space to prove pass-alive can be large. Benson [1] defined the concept of unconditional life in a way that is easier to apply, because no search is required. At the same time, Benson also proved that a set of chains is unconditionally alive if and only if it is pass-alive. As such, unconditional life and pass-alive have become synonyms.

Benson's definition of unconditional life also leads to an algorithm for finding chains that are unconditionally alive, known as Benson's algorithm. This algorithm is not easy to apply manually, and therefore is intended for computer life-and-death programs.

With Benson's algorithm, when we find a set of chains is unconditional live, we can concentrate on other areas on the board which can reduce meaningless moves in MC simulations.

References

[1] D.B. Benson. Life in the game of go. *Information Sciences*, 10(2):17–29, 1976.

[2] B. Bouzy and T. Cazenave. Computer go: an ai oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[3] Guillaume Chaslot, Louis Chatriot, C. Fiter, Sylvain Gelly, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Combining expert, offline, transient and online knowledge in monte-carlo exploration.

[4] M. Enzenberger, M. Muller, B. Arneson, and R. Segal. Fuego - an open-source framework for board games and go engine based on monte-carlo tree search. *Computational Intelligence and AI in Games, IEEE Transactions on*, (99):1–1, 2010.

[5] S. Gelly. A contribution to reinforcement learning; application to computer-go. 2007.

[6] S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 2011.

[7] I. L. Grace. Fuego go: The missing manual. 2010.

[8] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.