Go4ever: A Computer GO implementation with UCT+RAVE Algorithm

Technical Report
Xie Saining

Abstract

This technical report is for the SJTU AI-course project. In this project, we design and implement a Computer Go Program, named Go4ever. We use the Monte-Carlo Tree Search method with state-of-the-art modifications like Upper Confidence Bound for Trees (UCT) and Rapid Action Value Estimation (RAVE) to improve the competitiveness of our Computer GO program. This technical report is divided into two parts, the first is to illustrate and highlight the general design and implementation of our team. In the second part I will briefly state some personal contributions in the whole project and explain some implementation details which is omitted in part one.

1 Go4ever: The Team Work

1.1 The Board

1.1.1 Basics data structures

The basics design of board class used in go4ever are influenced by GNU Go, and several discussion threads in computer-go mail-list:

- Use one dimension array, and add border cells in order to avoid index checking.

- Use cyclic linked list to represent the connect group of the same color. For each stone, there is a pointer to another (maybe same cell for one cell group) cell, so it’s possible to enumerate all cells of a group, and quite easy to merge two groups.

- There are some extra information need to be stored in each group, such as liberties and Atari status. In order to check whether two cells are in the same group, we compared two approach: one based on union-find set, and another one records the size of group, and enumerates all cells in smaller group to merge it into the larger one. We choosed the latter one since it’s slightly faster in our experiments.

- use the pseudo-liberty instead of liberty to find out dead group in simulation. It’s possible that a liberty cell (which is empty) of a group have more than one directly connected cell in this group. It’s hard to record the exact number of liberty cells, but it’s quite easy to check if it’s non-zero: the idea is to count of edges from liberty cells to this group, instead the count of liberty cells. This value can be maintained by modify respect groups’s pseudo-liberties when adding or removing stones to/from board.
1.1.2 Implementation of Chinese rule

The basic control flow of a random simulator is like this:

- try to find an empty cell which is also legal to put one’s stone
- put down the stone, or pass if can’t find such cell
- evaluation the game if both player passed consecutively

The most tricky part in implementing a fast board class is to check the legal moves. There is a rule to forbid suicide and another rule to forbid move to repeat previous board (ko rule). In a naive approach, it’s necessary to copy the board try to put the stone on the board and. We believe it’s quite inefficient, and we try to check legal moves inplace.

The First approach is to based on revert-back:

- put the stone
- check whether other player’s stone can be captured by neighbor’s pseudo-liberties
- if no stones can be captured, check if it’s a suicide by neighbor’s pseudo-liberties
- if it’s a suicide, revert back by remove the stone we put in first step
- check the hash value of board in the end.
- if it’s same as previous hash of old board, redo last move to revert back

This is difficult to implement this approach, and the it’s still not fast enough.

Fortunately we find another approach based on the atari status of neighbor’s group.

- if there is a neighbor cell with opponent’s stone and in atari status, it can be captured
- if no stones can be captured, check if it’s a suicide
  - if there is a empty cell among neighbor cells, it’s not a suicide
  - if there is a neighbor cell with my stone and not in atari status, it’s not a suicide
  - otherwise it’s a suicide.
- when putting stone, record the position may lead to ko situation in next step if
  - exactly one stone is captured
  - the newly put stone form a one-size group
  - the group is in atari status

The second approach is much better, but it require to maintain Atari status for each group, and also the atari’s position.

1.1.3 Heuristics based simulation

We use eyelike to define eyes, its definition is

- 1. all neighbor’s cell are own color or border
- 2. let $a$ be the number of cell in diagonal neighbor
- 3. let $b$ be 1 if there is border cell in diagonal neighbor
4. It’s a eyelike iff. $a + b < 2$

We checked about three thousands game records, no one ever filled an eyelike defined above.

In order to bring some domain knowledge to our program, we used 3x3 pattern to increase quality of random playouts. Our 3x3 pattern contains the color of 8 neighbors’ cell(Black, White, Empty or Border) and also the 4 neighbor’s atari status. We assign weights to all pattern and randomly choosed legal positions (note legal move can be judge by our 3x3 pattern now) according to the weights.

### 1.2 Monte-Carlo Simulations

Monte-Carlo simulation provides a simple method for estimating the value of positions and moves. N games are simulated using self-play, starting from positions and playing a candidate move $a$, and then continuing until the end of the game using a certain policy. The estimated value $Q(s; a)$ is the average outcome of all simulations in which move $a$ was selected in position $s$,

$$Q(s; a) = \frac{1}{n(s, a)} \sum_{i=1}^{N} I_i(s, a) z_i$$

### 1.3 Upper Confidence for Trees: Main Idea

Monte-Carlo Tree search builds a search tree of positions and moves that are encountered during simulations from the current position. There is one node in the tree for each position $s$ and move $a$, containing a count $n(s, a)$, and a value $Q(s, a)$ that is estimated by Monte-Carlo simulations.

Each simulation starts from the current position $s_t$, and is divided into two stages: a tree policy is used while all children are contained in the search tree; and a default policy is used for the remainder of the simulation. The simplest implementation of the tree search algorithm uses a greedy tree policy during the first stage, that selects the move with the highest value among all child nodes; and a uniform random default policy during the second stage, that rolls out simulations until completion. The tree is grown by one node per simulation, by adding the first position and move in the second stage.

Monte-Carlo tree search can be extended by enhancing the policy used in each stage of the simulation. A domain specific default policy can be very helpful, e.g. the tree policy can be significantly improved by encouraging exploratory moves.

This is exactly the basic idea of Upper Confidence Tree Search, which is the core of our design. The UCT algorithm treats each individual position as a Multi-armed Bandit, and selects the move $a$ in
position \( s \) with the greatest Upper confidence bound \( Q^\oplus(s, a) \) on its value,

\[
Q^\oplus(s, a) = Q(s, a) + c \sqrt{\frac{\log n(s)}{n(s, a)}},
\]

where \( n(s) \) counts the total number of visits to position \( s \), and \( c \) is a constant determining the level of exploration. Similarly, opponent moves are selected by minimizing a lower confidence bound on the value. Typically, many thousands of games are simulated per move, building an extensive search tree from the current position \( s_t \).

As the tree grows, the tree policy becomes more informed; and as the tree policy improves, the values in the tree become more accurate. Under some assumptions, it is proved in (Kocsis and Szepesvari 2006) that UCT converges on the min-max value.

However, unlike other min-max search algorithms such as alpha-beta search, UCT requires no prior domain knowledge to evaluate positions or order moves. Furthermore, the UCT search tree is highly non-uniform and favors the most promising lines. What is more, 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. These properties make UCT ideally suited to the game of Go, which has a large state space and branching factor, and for which no strong evaluation functions are known.

1.4 Rapid Action Value Estimation: Main Idea

Monte-Carlo tree search separately estimates the value of each state and each action in the search tree. As a result, it cannot generalize between related positions or related moves. To determine the best move, many simulations must be performed from all states and for all actions. The RAVE algorithm uses the \textit{all-moves-as-first} heuristic, from each node of the search tree, to estimate the value of each action. RAVE provides a simple way to share knowledge between related nodes in the search tree, resulting in a rapid, but biased estimate of the action values. This biased estimate can often determine the best move after just a handful of simulations, and can be used to significantly improve the performance of the search algorithm.

Figure 2: An example of using the RAVE algorithm to estimate the value of Black moves a and b from state s. Six simulations have been executed from state s, with outcomes shown in the bottom squares. Playing move a immediately led to two losses, and so Monte-Carlo estimation favors move b. However, playing move a at any subsequent time led to three wins out of five, and so the RAVE algorithm favors move a. Note that the simulation starting with move a from the root node does not belong to the subtree and does not contribute to the RAVE estimate \( \hat{Q}(s, a) \).

\( Q(s, a) = 0/2 \)
\( Q(s, b) = 2/3 \)
\( \hat{Q}(s, a) = 3/5 \)
\( \hat{Q}(s, b) = 2/5 \)
The RAVE value $Q(s, a)$ is the average outcome of all simulations in which move $a$ is selected in position $s$, or in any subsequent position,

$$
\hat{Q}(s, a) = \frac{1}{\hat{n}(s, a)} \sum_{i=1}^{N} \hat{I}_i(s, a) z_i
$$

(3)

where $\hat{I}_i(s, a)$ is an indicator function returning 1 if positions was encountered at any step $k$ of the $i$-th simulation, and move $a$ was selected at any step $t >= k$, or 0 otherwise. $\hat{n}(s, a) = \sum_{i=1}^{N} \hat{I}_i$ counts the total number of simulations used to estimate the RAVE value.

The RAVE value $\hat{Q}(s, a)$ generalizes the value of move $a$ across all positions in the subtree below $s$, and subsequent positions encountered during the default policy. It is closely related to the all moves as first heuristic in Computer Go (Bruegmann1993).

1.5 Combine UCT and RAVE

The rapid action value estimate can quickly learn a low-variance value for each action. However, it may introduce some bias, as the value of an action usually depends on the exact state in which it is selected. Hence we would like to use the rapid estimate initially, but use the original UCT estimate in the limit. To achieve this, we use a linear combination of the two estimates, with a decaying weight $\beta$.

$$
\tilde{Q}^\oplus(s, a) = \hat{Q}(s, a) + c \sqrt{\frac{\log m(s)}{m(s, a)}}
$$

(4)

$$
\beta(s, a) = \sqrt{\frac{k}{3n(s) + k}}
$$

(5)

$$
Q_{UCT-RAVE}^\oplus(s, a) = \beta(s, a) \hat{Q}^\oplus(s, a) + (1 - \beta(s, a)) Q^\oplus(s, a)
$$

(6)

where $m(s, a)$ counts the number of times that action $a$ has been selected at any time following state $s$, and $m(s) = \sum_a (s, a)$. 

\[\text{5}\]