Sensei's Library: UCT

UCT for Monte Carlo computer go

A Monte Carlo (*MC*) go program plays random games and easily evaluates the terminal position after two passes using Chinese rules. A MC program searches for moves that have high win rates, calculated from playing out at least a few hundred random games.

A basic MC program would simply collect statistics for all children of the root node, but MC evaluation of these moves will not converge to the best move. It is therefore necessary to do a deeper search.

The UCT-method (which stands for Upper Confidence bounds applied to Trees) is a very natural extension to MC-search, where for each played game the first moves are selected by searching a tree which is grown in memory, and as soon as a terminal node is found a new move/child is added to the tree and the rest of the game is played randomly. The evaluation of the finished random game is then used to update the statistics of all moves in the tree that were part of that game.

The UCT-method solves the problem of selecting moves in the tree such that important moves are searched more often than moves that appear to be bad. One way of doing this would be to select the node with the highest winrate 50% of the time and select a random move otherwise.

UCT also selects the best move most of the time but also explores other moves in a more sophisticated way. It does this by adding a number to the winrate for each candidate move that goes down every time the node has been visited. But this number also goes up a little every time the parent was visited and some other move is selected. This means that the win rate + the number will grow for unexplored moves so that at some point the sum is higher than for all moves that have higher winrates. If the move works, then the winrate goes up and it may soon be selected again. If the move failed to win then the winrate goes down as well as the added number and the move may have to wait for a long time before the added number has grown large enough. A move can also be selected if all other moves are refuted so that the winrates for all competitors goes down.

Starting from the root, UCT searches a path of moves through the tree by calculating a value for each candidate position according to the rate of win and how many times the position has been played as well as how many times the position has been visited. If there are children to a node that has not been visited then one of those moves are selected randomly. As this method does not assume any knowledge about game of Go it is natural to go through every move at least once.

The sum of the winrate and the number (UCTvalue) for each candidate move n is computed with

UCTValue(parent, n) = winrate + sqrt((ln(parent.visits))/(5*n.nodevisits))

and the candidate move n with highest UCTvalue is selected to be played next.

With UCT one can strike a good balance between searching the best move so far and exploring alternative moves. I think the innovation of UCT (to me at least) is the logarithm of the number of visits to the parent node for the UCT-Value. This value goes up quickly early but then it becomes flat, but

never stops rising. It goes to infinity although very very slowly. All moves will be searched no matter how bad winrates they have, so that UCT-search given close to infinite time and memory will find any winning move no matter how misleading the winrates are initially.

Reference

For further reading

Implementation details

There are at least two ways to implement the tree:

- Build a tree in memory Pro: Easy to program Con: You quickly run out of memory using this method.
- Transposition Table: Pro: Identical positions reached with different move orders will not be searched twice Con: Depending on implementation details one might have problems with the correctness of the algorithm since one need to overwrite old data so it is perhaps more complicated to get bug free code.

Pseudo code for UCT (Based on Valkyria-UCT 22 September 2006)

This pseudo code example ignores what happens close to the end of the game. When Valkyria finds a terminal node (after two passes) it assigns INFINITY to n.Visits (and 0 or INFINITY to n.wins depending if it was a win or loss) to indicate that this node is solved. Higher up in the tree one must also check if a child refuted the node or that all children failed to refute it.

const UCTK = 1; // Larger values give uniform search // Smaller values give very selective search type TMove = (...); PNode = ^Node; Node = record Wins: integer; Visits: integer; Move: TMove; BestNode: PNode; Child: PNode; Sibling: PNode; end; function UCTSelect(n: Node): Node; begin bestuct := 0; result := nil; next := n.Child; while next <> nil do begin if next.Visits > 0 then begin winrate := next.Wins/next.Visits; uct := UCTK*Sqrt(ln(n.Visits)/(5*next.Visits)); uctvalue := winrate + uct; end else // Always play a random unexplored move first uctvalue := 10000 + random(1000); if uctvalue > bestuct then begin bestuct := uctvalue; result := next; end; next := next.Sibling; end; end; procedure PlaySimulation(n: Node); begin if n.Visits = 0 then randomresult := clone.PlayRandomGame else begin if n.Child = nil then CreateChildren(n); next := UCTSelect(n); clone.MakeMove(next.Move); PlaySimulation(next); end; Inc(n.Visits); UpdateWin(n, randomresult); if n.Child <> nil then SetBest(n); end; function UCTSearch: TMove; begin root := GetNode; nsimulations := 0; while (nsimulations < MaxSimulations) do begin clone.CopyState(position); PlaySimulation(root); Inc(nsimulations); end; result := root.BestMove; end;

PlayRandomGame This method plays a random game and returns the color of the winner

CreateChildren(n: Node); This procedure generates all legal successors in the position and adds those moves as children to the node n.

UpdateWin(n: Node; randomresult: TGameResult); This procedure adds 1 to n.Wins if the randomresult indicates that this is correct.

CopyState(position: TGameState); In order to play a random game the boardstate has to be copied

SetBest(n: Node) This procedure sets n.BestMove to the child with the highest winrate.

Note that the variable randomresult is a global variable.

Sensei's Library: UCT

Programs

The CGOS Basic UCT Bots list contains tuning guidelines for various UCT parameters and CGOS ratings you should expect from a working UCT implementation.