Monte Carlo Tree Search

Brian Hare
Fall 2018

We've seen that there are several ways of dealing with the problem of adversarial search when the game tree is too large to search exhaustively. Various pruning strategies can reduce the number of nodes that must be evaluated, and memoization can be used to reduce the amount of work that must be re-done, but that's usually not enough. So we use some sort of static evaluation function to estimate the likely outcome if the game were played forward from this point. But now our decisions are based on estimates, and there are some problems with that. First, our evaluation function may be incorrect or biased in ways that we're not aware of. This can lead us to make poor choices, or prune away promising lines of play that we don't recognize. Second, a game may have subtle variations in strategy and position that make static evaluation difficult—Go is an example of this. It's often difficult to tell who's winning in Go until very late in the game. So just relying on deeper search to make up for inaccurate evaluation will only go so far.

So if we must rely on estimates, we'd at least like them to be unbiased—to be some approximation of our chance of winning from a particular position. And we recognize that just as with most methods, greater accuracy is going to require more work.

Begin with some terminology. The *current* node is the one we're trying to select a move for, usually the current position of the game. It will serve as the root of our tree. Each possible move by the side to move creates a *child* node representing the position that results from making that move. We wish to select the child node that gives us the greatest chance of winning. Of course, each child node can itself be the root of a tree, showing the results of that move, and so on down the tree.

We derive these estimates by *sampling* each child, following a line of randomly-selected moves out to a *terminal position*, when the game is definitely won, lost, or drawn. We use the proportion of winning samples for each child as an estimate of its chances.

But wait a minute. Suppose we have two possible lines of play; one of them wins in 8 of the 10 times we've sampled it (80%), the other wins 1 in 10 (10%). It would probably be a more effective use of our time to sample the better-looking move, to be sure it's as promising as it looks. After all, if we win 8 of 10 times using random moves, it's probably a very good move—but it only takes *one* effective counter-move by the opponent to ruin it, and we probably haven't eliminated that possibility in only 10 samples. So we should weight our selection by the probability of winning. Since 80+10 = 90, we should sample the first line of play with 80/90 = 89% probability and the second with 10/90 = 11% probability.

But wait another minute. There are two issues here. First, we have no wins for a line of play we haven't sampled yet, but "0 because it hasn't been sampled" and "0 because we've checked at least once and not found a win" are different situations. So we'll add a housekeeping rule: Give preference to any move that hasn't been sampled yet. If there's more than one that hasn't been sampled, select uniformly from the unsampled children, in keeping with the spirit of Monte Carlo sampling.

But suppose we select a move, follow it out to the end, and it loses. Then it has an estimate of 0% winning. By our method above, it'll never be chosen for further exploration, no matter how many samples we take. We should probably check it again at some point. If we've only looked at 10 positions and we checked that one once, that may be relatively adequate for the crude sample we expect. On the other hand, if we've sampled another line of play 5,000 times and this one once, maybe we should check again a time or two, just to be sure this is really a weak move.

We can use the quality of our estimate as a measure of when it's time to take another look. In general, we can be confident that the actual sample mean lies within an interval around our estimate given by

$$\frac{w_i + d_i/2}{n_i} \pm c\sqrt{\frac{\ln N}{n_i}}$$

where w_i is the number of wins for that move, d_i is the number of draws for that move, n_i is the number of samples of that move, N is the total number of samples across all children, and $c > 0$ is a constant scaling factor. Your strategy is to select the move with the highest upper confidence limit. As you sample this move more, its confidence interval will narrow, and the confidence intervals of other moves will widen. Eventually a move that has a lower expected value (but hasn't been sampled in a while) will have a higher upper limit, and be chosen for sampling. So the first term (the expected payoff of this move) encourages us to closely examine the most promising moves (*exploitation*), and the second encourages looking at moves that haven't been looked at in a while (*exploration*). The scaling factor *c* determines how much weight we give those factors (and in practice may take some experimentation to find the right balance).

There are a couple of variations—for example, Alpha Zero multiplies *c* by the estimated probability a move is good, adds 1 to the denominator under the square root, and uses the resulting formula even for nodes that haven't been sampled yet.

$$\frac{w_i + d_i/2}{n_i} \pm c\,Pr(BestMove)\sqrt{\frac{\ln N}{1+n_i}}$$

This encourages us to explore promising-looking moves first, and prevents the math from blowing up if a node is still unsampled and $n_i = 0$. Otherwise the principle is the same.

In actual play, this continues until our clock time available for the move has run out, or some pre-chosen maximum number of samples has been reached. We then select the move that was the most-often sampled, as it was the best move we found and has been estimated most thoroughly.

So how do we do each sample?

Technically we only need to store the immediate children of the node we're examining; in practice our estimates will be more accurate if we store some additional nodes as well. The issue is that as we move farther down the tree, the probability that we'll ever need to examine this position again drops sharply; the higher the branching factor, the more this holds. For chess, where the branching factor may be 50 or more in the mid-game and averages about 35, we may only store a couple of generations out; for Connect Four, with a maximum branching factor of 7, we can keep more extensive records.

But in general: For each node we sample, we follow the same idea. If there are unsampled children, we select one at random, otherwise we choose based on our estimated probabilities. (If we select at random like this, we are doing a *lightweight* playout. If, on the other hand, we generate all possible children and use a more computationally intensive process to choose the most promising line, we are doing a *heavyweight* playout. For games with a low branching factor, lightweight playouts are often sufficient. For games with a high branching factor and only a few really good moves, such as chess, more careful selection may pay for itself in better results. Alpha Zero uses a neural network that accepts a game position and returns a probability distribution to estimate which moves are most worth exploring.)

We continue working down the game tree, selecting moves randomly, until we reach a *terminal node*, where the game is definitively won, lost, or drawn. We then begin the *backup* phase of the process. As we work our way back up the tree, each node explored has its n_i count increased by 1, as it's been sampled; if it represents a move that was just made by the winning side, its w_i count is increased as well. This goes all the way back up to the root node, saving nodes near the top by whichever criteria we have decided on.

Practical concerns:

- The move selected by this method will converge on the best choice—eventually. The higher the branching factor and the farther from the end state, obviously, the more effort it will take.
- Fast move generation is vital. In some games this is simple: Go allows you to place a stone on any vacant position. Connect Four lets you drop a piece into any column that isn't full. Chess or backgammon, on the other hand, can be more time consuming. Generally the approach for lightweight playout is some variety of selecting pieces at random until a piece is found with any legal moves, then randomly selecting from its legal moves. For heavyweight playout, all children are generated and some sort of static evaluation applied to select the move to explore. Memoization can help with performance speed, at the cost of large amounts of memory. (If you have a large table of every position you've seen and the list of legal moves for each, you can do a lookup in even a huge table in less time than it'd take to re-generate the moves—assuming you have room for the huge table, whether on memory or on a fast disk.) Which leads to…
- The question of which sampled positions to remember and which to discard must be managed, and in part is a function of the branching factor of the game and how much room is available for hash tables. One approach is to keep track of which ply the game itself is on and which ply a position was generated for. Any position within some range *r* of the current play may be kept.
- Likewise, if the game is currently on ply *p*, anything stored as part of an earlier ply is probably obsolete and irrelevant; from time to time the data structure can be cleaned out of positions that will no longer occur in any search. (This is more possible in Go and Connect Four, for example, where pieces cannot move once placed.)
- On the other hand, if we have the storage space, there's no reason not to remember as much as possible. Since the information we store is independent of our evaluation function, we can use it in future games, or even during a learning process when the evaluation function (used to rank/select moves) may be changing. As long as the definition of what makes a game won or lost doesn't change, the information we store (in *x* positions sampled, *y* were wins) remains valid.

References/further reading:

Introduction to Monte Carlo Tree Search
Wikipedia: Monte Carlo Tree Search
Beginner's Guide to Monte Carlo Tree Search