Monte Carlo Tree Search (MCTS, 蒙特卡罗树搜索)

- MCTS algorithm
- Comparing alpha-beta and MCTS

Why MCTS?

The game of Go illustrates two major weaknesses of heuristic alpha-beta tree search:

- Go has a branching factor that starts at 361, which means alpha-beta search would be limited to only 4 or 5 ply.
- it is difficult to define a good evaluation function for Go

MCTS: main ideas

- The value of a state is estimated as the average utility over a number of simulations of complete games starting from the state.
- A simulation (also called a playout or rollout) chooses moves first for one player, than for the other, repeating until a terminal position is reached.
- At that point the rules of the game determine who has won or lost, and by what score.
- For games in which the only outcomes are a win or a loss, "average utility" is the same as "win percentage".

Playout policy

How do we choose what moves to make during the playout?

- If we just choose randomly, then after multiple simulations, we get an answer to the question "what is the best move if both players play randomly?"
- But we want "what is the best move if both players play well?"
- So we need a playout policy that biases towards good moves.
- For Go and other games, playout policies have been successfully learned from self-play by using neural networks.
- Sometimes game-specific heuristics are used, such as "take the corner square" in Othello.

Selection policy

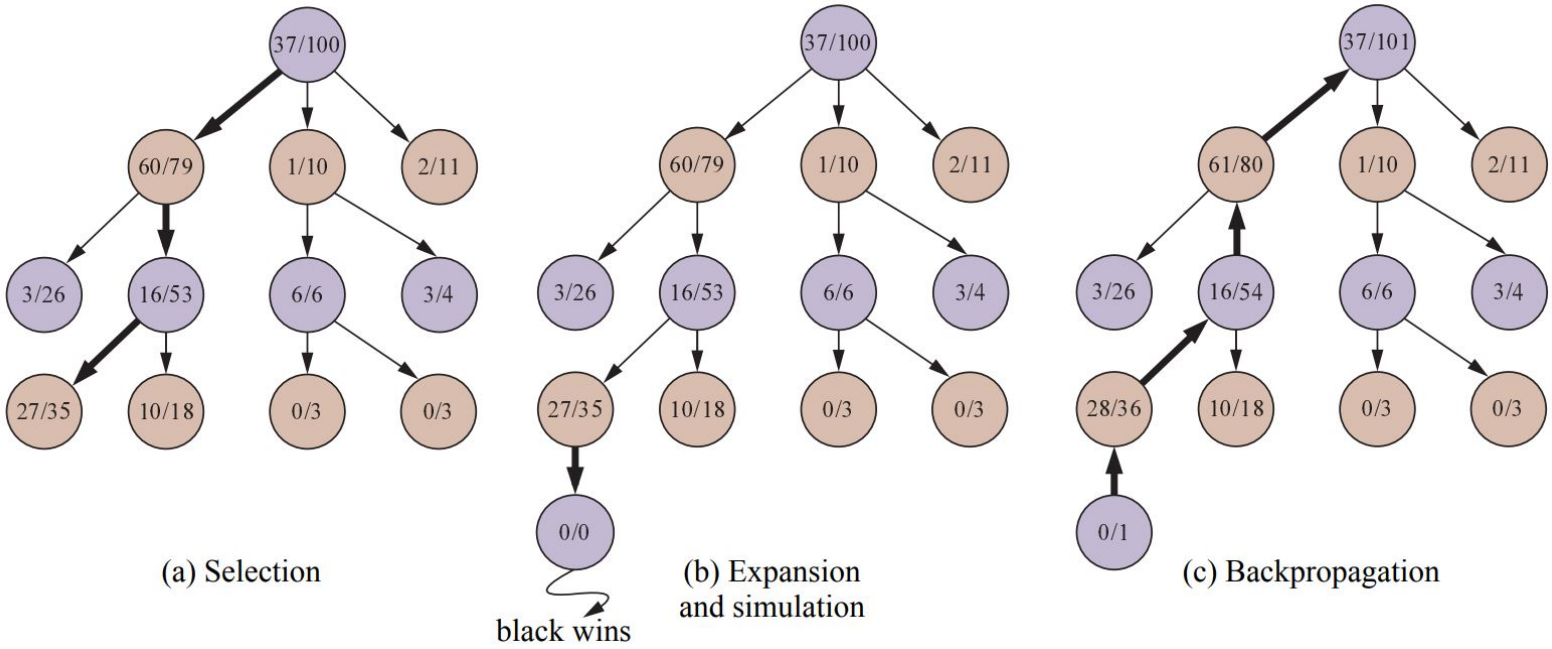
- From what positions do we start the playouts, and how many playouts do we allocate to each position?
- The simplest way, called pure Monte Carlo search: do N simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.
- However, we need a selection policy that focuses on the important parts of the game tree.
- It balances two factors: exploration of states that have had few playouts, and exploitation of states that have done well in past playouts, to get a more accurate estimate of their value.

MCTS

MCTS maintains a search tree and grows it on each iteration of the following four steps:

- Selection: Starting at the root of the search tree, choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf.
- Expansion: grow the search tree by generating a new child of the selected node;
- Simulation: perform a playout from the newly generated child node, choosing moves for both players according to the playout policy.
- Back-propagation: use the result of the simulation to update all the nodes going up to the root.

An example iteration



$U(n)/N(n)$:
$U(n)$ is the total utility of all playouts through node n,
$N(n)$ is the number of playouts through node n

UCT selection policy

UCT (upper confidence bounds applied to trees)

Rank each possible move based on an upper confidence bound (上限置信区间) formula called UCB1

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

- $U(n)$: the total utility of all playouts through node n,
- $N(n)$ is the number of playouts through node n,
- PARENT(n) is the parent node of n in the tree

UCB1 formula

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

- $\frac{U(n)}{N(n)}$ is the exploitation term: the average utility of n.
- The term with the square root is the exploration term:
  - it will be high for nodes explored only a few times
- C is a constant that balances exploitation and exploration.
  - There is a theoretical argument that C should be $\sqrt{2}$,
  - but in practice, game programmers try multiple values for C and choose the one that performs best.

The UCT MCTS algorithm

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

Why returning the node with the most playouts?

- UCB1 ensures that the node with the most playouts is almost always the node with the highest win percentage,
- because the selection process favors win percentage more and more as the number of playouts goes up

The time to compute a playout is linear, not exponential, in the depth of the game tree

Advantages of MCTS

- Alpha-beta chooses the path to a node with the highest achievable evaluation function score
- Thus, if the evaluation function is inaccurate, alpha-beta will be inaccurate.
- A miscalculation on a single node can lead alpha-beta to erroneously choose (or avoid) a path to that node.
- But MCTS relies on the aggregate of many playouts, and thus is not as vulnerable to a single error.

Advantages of MCTS

- MCTS can be applied to brand-new games, in which there is no experience to draw upon to define an evaluation function.
- As long as we know the rules of the game, MCTS does not need any additional information.
- The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone.

Advantages of MCTS

- It was long held that alpha-beta search was better suited for games like chess with low branching factor and good evaluation functions,
- but recently Monte Carlo approaches have demonstrated success in chess and other games.

Disadvantages of MCTS

- It is likely that a single move can change the course of the game, MCTS might fail to consider that move, because of its stochastic nature.
- There may be game states that are "obviously" a win for one side (according to human knowledge and to an evaluation function), but it will take many moves in a playout.