Monte Carlo Tree Search (MCTS)

Christophe Louargant

November 27, 2025

Abstract

This document provides a comprehensive visualization of Monte Carlo Tree Search algorithms for the two-player HEX game.

Introduction to Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that combines the precision of tree search with the generality of random sampling. Unlike traditional minimax algorithms that require evaluation functions, MCTS uses random playouts to estimate the value of positions. In MCTS a game is represented by a **game tree** which is a tree in which every node represents a state of the game. The very root of the tree represent the initial state of the game (before any move[1] is done). After choosing a move you end up in a child node. This child node is a root node for its subtree. Since for HEX game there is no point remembering the path leading to the current state, we can consider the tree starting from the new state of the game.

The Core MCTS Algorithm

The four-phase structure repeated for many iterations:

1. **Selection**: Traverse the tree from root to leaf using UCB1
2. **Expansion**: Add new child node when reaching a leaf
3. **Simulation**[2]: Run random playout from the new node.
4. **Backpropagation**[3]: Update statistics along the path

UCB1[4] Formula: The Bandit Component

$$\text{UCB1}(i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$

Where:

- w_i: Number of wins for node i

[1] a move is a transition from a node to one of its children
[2] Sometimes goes by the names *rollout, playout, and sampling.*
[3] Sometimes called *backup.*
[4] UCB1 applied to trees is essentially UCT (Upper Confidence bounds applied to Trees)

- n_i: Number of visits for node i
- N: Total number of visits to parent node
- c: Exploration constant (typically $\sqrt{2}$)

Naive MCTS

In Naive MCTS, we make a critical assumption: **the opponent plays randomly**. This assumption simplifies the problem significantly:

- **No opponent modeling**: We don't need to consider the opponent's optimal responses
- **Immediate evaluation**: After our move, the outcome depends only on random play
- **Computational efficiency**: Much faster than building deep trees

Since we assume random opposition, there's no benefit to looking deeper than our immediate move. The random playout from depth 1 already gives us a statistically meaningful evaluation of each move's quality.

Step-by-Step Visualization

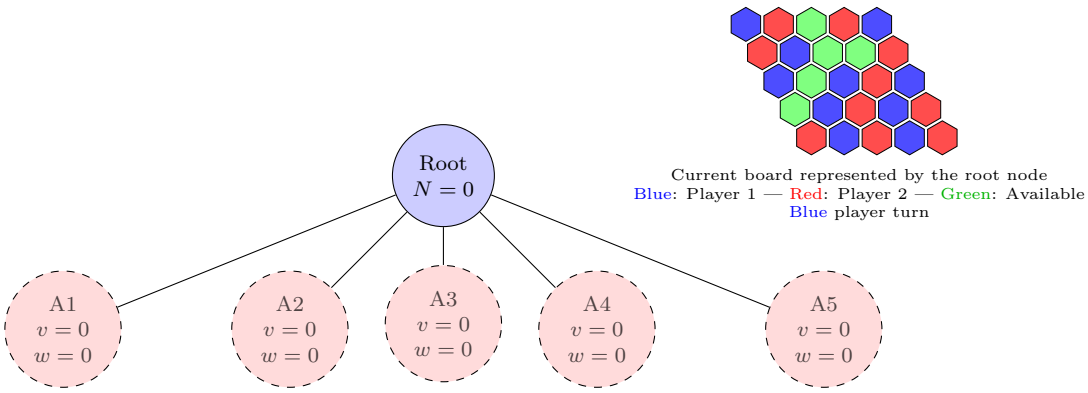
Current State

Since there is no point keeping the path from the initial state (empty board) to current state, the tree that the algorithm build has a maximum depth of 1 with the root representing the current state of the game.

Let's consider that it is the turn of player 1 (Blue player) to play and that it has to choose between 5 free positions.

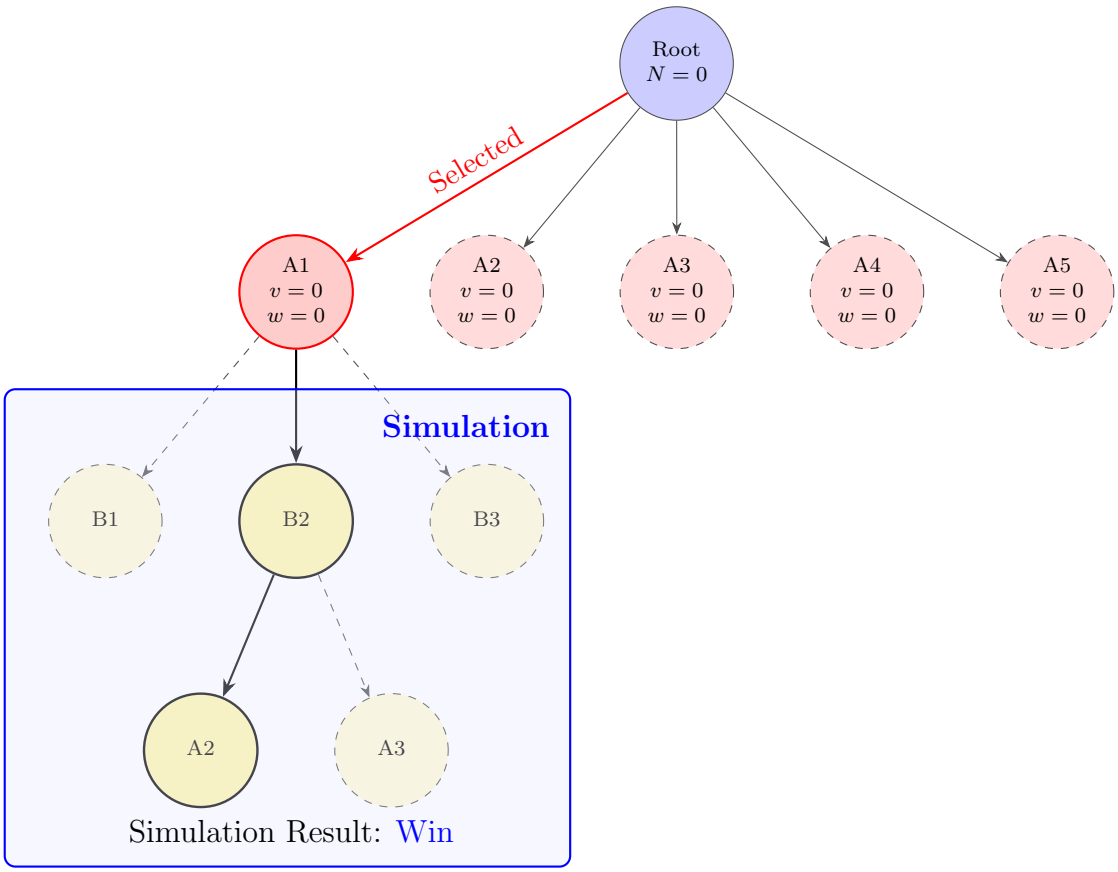
The root node of the tree represent the board current state and it has 5 children (5 available moves: A_1, A_2, A_3, A_4, A_5. All unvisited).

A simple MCTS tree visualization.



Iteration 1: First Expansion

All nodes have UCB1 = ∞ (division by zero). Randomly select A1.

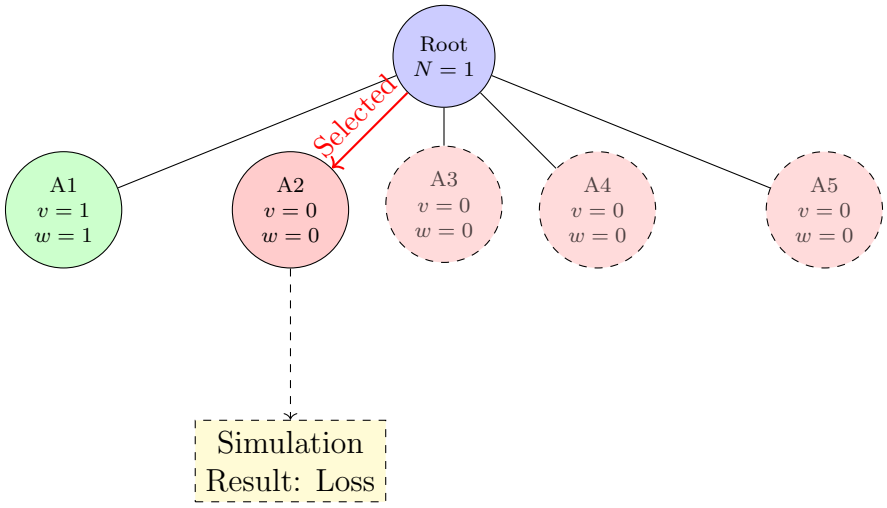


Backpropagation: A_1 wins $\Rightarrow v = 1$, $w = 1$, Root $N = 1$

From now on, I will not draw the full simulation tree but just the result of the simulation.

Iteration 2: Second Expansion

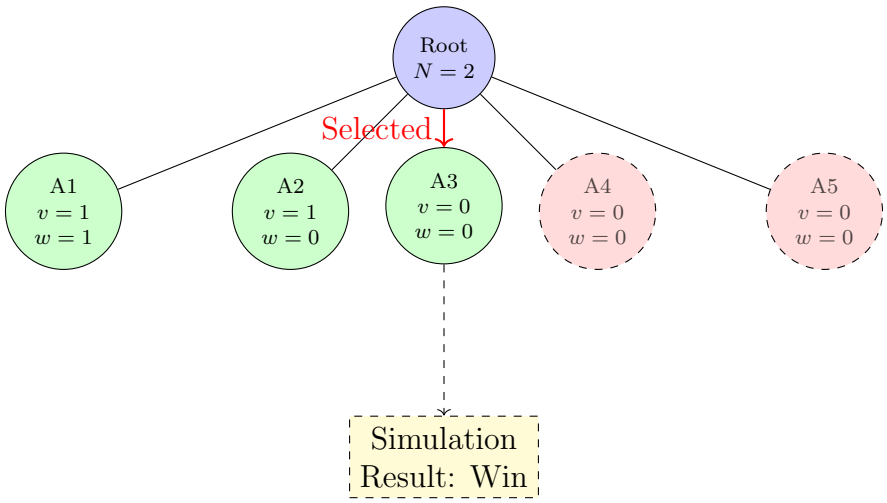
Unvisited nodes still have UCB1 $= \infty$. Randomly select A2.



Backpropagation: A2 loses $\Rightarrow v = 1$, $w = 0$, Root $N = 2$

Iteration 3: Third Expansion

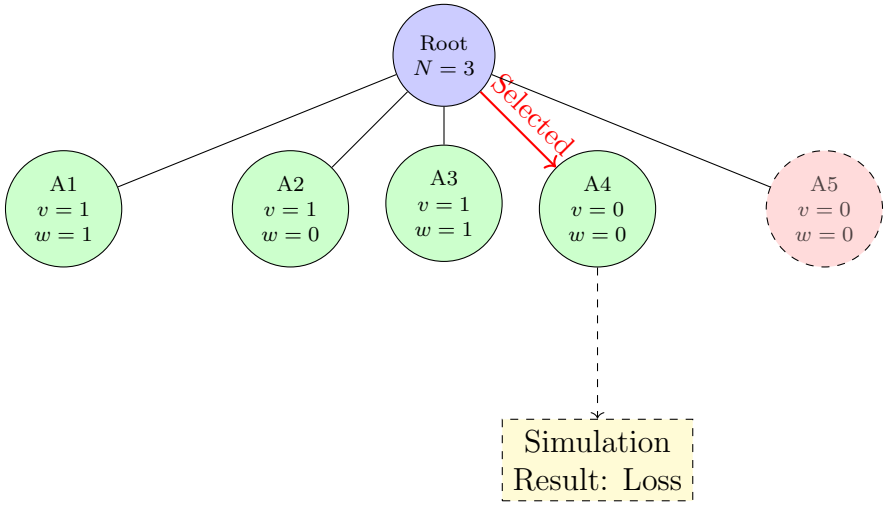
Select A3 (remaining unvisited node).



Backpropagation: A3 wins $\Rightarrow v = 1$, $w = 1$, Root $N = 3$

Iteration 4: Fourth Expansion

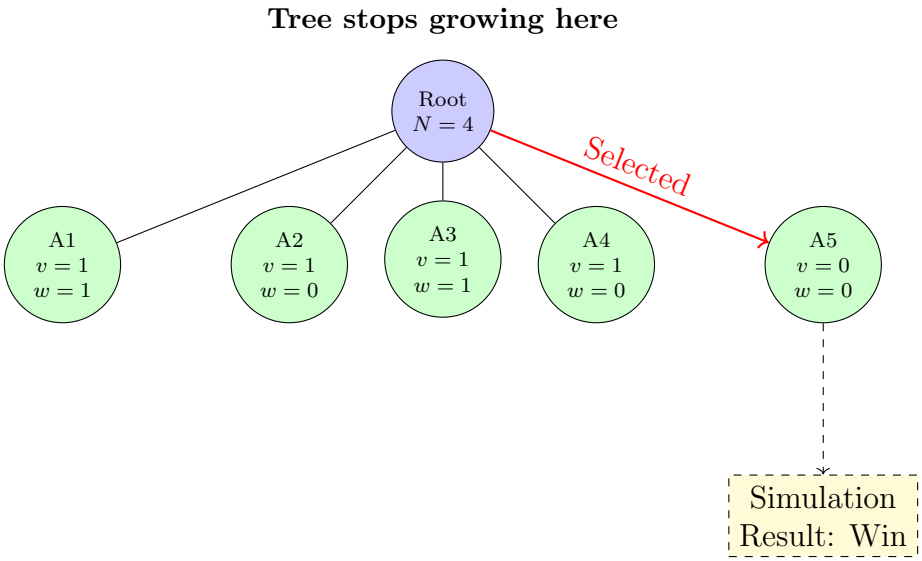
Select A4 (remaining unvisited node).



Backpropagation: A4 loses $\Rightarrow v = 1$, $w = 0$, Root $N = 4$

Iteration 5: Final Initial Expansion

Select A5 (last unvisited node).



Backpropagation: A5 wins $\Rightarrow$ $v = 1$, $w = 1$, Root $N = 5$

Iteration 6: First UCB1 Selection

Now all nodes have been visited. Calculate UCB1 ($c = \sqrt{2}$):

$$\text{UCB1}(A1) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 1 + 1.87 = 2.87$$

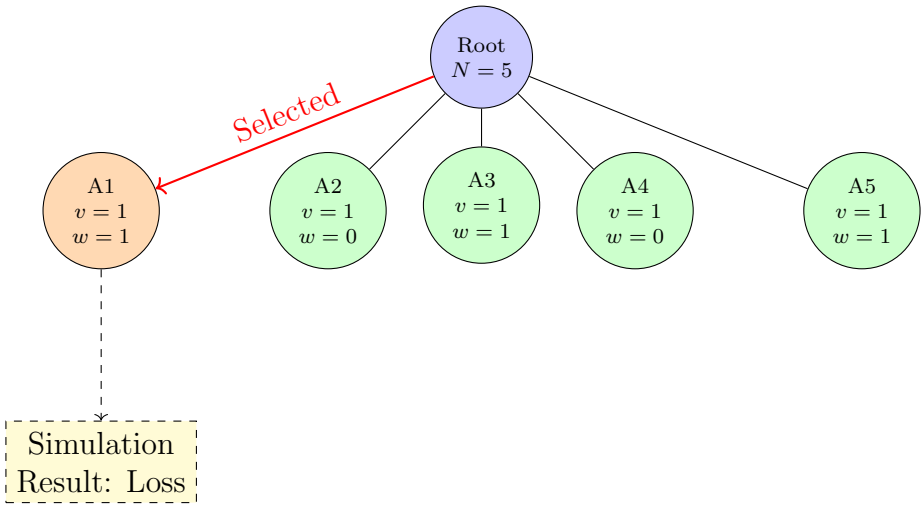
$$\text{UCB1}(A2) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 0 + 1.87 = 1.87$$

$$\text{UCB1}(A3) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 1 + 1.87 = 2.87$$

$$\text{UCB1}(A4) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 0 + 1.87 = 1.87$$

$$\text{UCB1}(A5) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 1 + 1.87 = 2.87$$

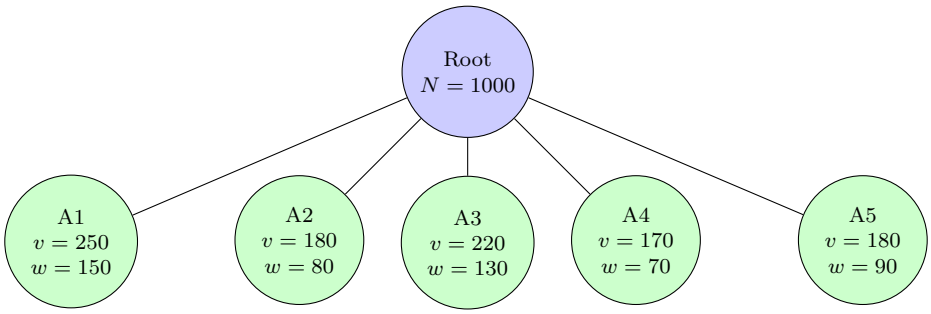
Tie between A1, A3, A5. Randomly select A1.



Backpropagation: Only updates selected node A_1: $v = 2$, $w = 1$, Root $N = 6$

Final Tree State

After many iterations, the tree might look like:



Final Decision: Choose move with highest win rate[5]: A1 ($150/250 = 60\%$)

When to Use Naive MCTS

- **Simple opponents**: When the opponent plays randomly or weakly
- **Computational constraints**: When deep lookahead is too expensive
- **Rapid prototyping**: Quick implementation for testing ideas
- **Very large branching factors**: When even one-ply exploration is challenging

[5]Or choose move with highest visit count