Monte Carlo Tree Search (MCTS)
A Comprehensive Visual Guide

Christophe Louargant

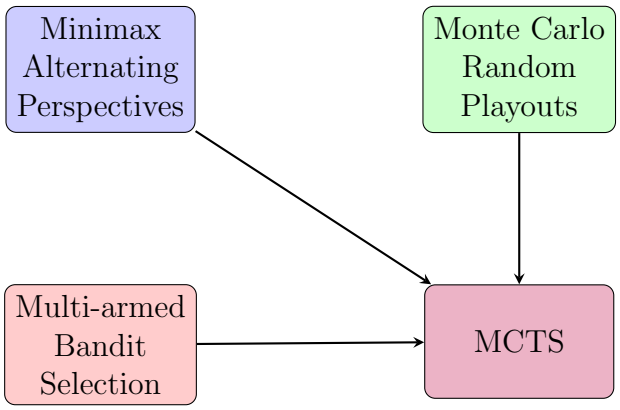
December 4, 2025

Abstract

This document provides a comprehensive visualization of Monte Carlo Tree Search algorithms applied to zero-sum two-player games. Through step-by-step tree diagrams and detailed explanations, we demonstrate how these methods work for two-player games like HEX. We explore the algorithm's components, including the minimax perspective, random playouts, and UCB1 selection strategy.

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. MCTS combines three powerful ideas:



The Perfect Hybrid

The Magic Formula

$$\text{MCTS} = \text{MinimaxStructure} + \text{MonteCarloEvaluation} + \text{BanditSelection}$$

Understanding the Minimax Component

MCTS incorporates minimax thinking through its tree structure. In a two-player zero-sum game:

- **Alternating perspectives**: Each level of the tree represents a different player's turn

- **Player 1 nodes** (odd depths): We want to maximize our win probability
- **Player 2 nodes** (even depths): The opponent tries to maximize their win probability (minimize ours)
- **Win statistics from perspective**: When we backpropagate, each node's win count w represents wins for the player who *made the move* leading to that node

This alternating perspective is what makes MCTS work for adversarial games. When selecting moves using UCB1, each player naturally chooses nodes with high win rates *from their own perspective.*

The Core MCTS Algorithm

The four-phase structure repeated for many iterations:

1. **Selection**: Traverse the tree from root (which represents the current state of the game) to a leaf node using the UCB1 formula
2. **Expansion**: Add one or more new child nodes when reaching a leaf
3. **Simulation**[1]: Run a random playout from the new node to a terminal state
4. **Backpropagation**[2]: Update win and visit statistics along the path from the expanded node back to the root

Algorithm Pseudocode

Algorithm 1 MCTS Main Loop

```
function MCTS(root_state, num_iterations)
    root ← CreateNode(root_state)
    for i = 1 to num_iterations do
        node ← SELECT(root)
        child ← EXPAND(node)
        result ← SIMULATE(child)
        BACKPROPAGATE(child, result)
    end for
    return BESTMOVE(root)
end function
```

Note on Backpropagation and Perspective

A critical detail: during backpropagation, we flip the result as we move up the tree. This is because:

- If Player 1 won the simulation, nodes representing Player 1's moves should increment their win count

[1] Sometimes goes by the names *rollout, playout, and sampling.*
[2] Sometimes called *backup.*

Algorithm 2 MCTS Selection Phase

```
function SELECT(node)
    while node is not a leaf do
        if node has unvisited children then
            return node
        else
            node ← BESTCHILD(node, c)
        end if
    end while
    return node
end function
```

Algorithm 3 MCTS Expansion Phase

```
function EXPAND(node)
    actions ← GetUntriedActions(node)
    action ← SelectRandom(actions)
    child ← CreateChildNode(node, action)
    return child
end function
```

Algorithm 4 MCTS Simulation Phase

```
function SIMULATE(node)
    state ← node.state
    while state is not terminal do
        action ← SelectRandomAction(state)
        state ← ApplyAction(state, action)
    end while
    return GetResult(state)                ▷ 1 for P1 win, 0 for P1 loss
end function
```

Algorithm 5 MCTS Backpropagation Phase

```
function BACKPROPAGATE(node, result)
    while node is not null do
        node.visits ← node.visits + 1
        node.wins ← node.wins + result
        result ← 1 − result                ▷ Flip perspective
        node ← node.parent
    end while
end function
```

- But nodes representing Player 2's moves should *not* increment their win count (since Player 2 lost)
- Each node tracks wins from the perspective of the player who made that move

This alternating perspective is the **minimax component** in action!

UCB1 Formula: The Bandit Component

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

Where:

- w_i: Number of wins for node i (from the perspective of the player who made the move to reach node i)
- n_i: Number of visits for node i
- n_{p_i}: Total number of visits to the parent[3] node of node i
- c: Exploration constant (typically $\sqrt{2}$)

Understanding UCB1

The UCB1 formula balances two competing objectives:

- **Exploitation** (first term: $\frac{w_i}{n_i}$): Choose nodes with high win rates
- **Exploration** (second term: $c\sqrt{\frac{\ln n_{p_i}}{n_i}}$): Ensure less-visited nodes get explored

The exploration term increases based on the ratio $\frac{n_{p_i}}{n_i}$:

- As n_{p_i} increases (parent visited more), all children's exploration bonuses grow
- As n_i stays constant (child not visited), that specific child's exploration bonus grows relative to others
- This ensures that unvisited or rarely-visited nodes eventually get explored even if they initially seem unpromising

The algorithm in detail

Step-by-Step Visualization

Suppose the board is in a state where 5 positions remain available for the current player, let's say Player 1 (MCTS engine), to choose.

Important notation used throughout:

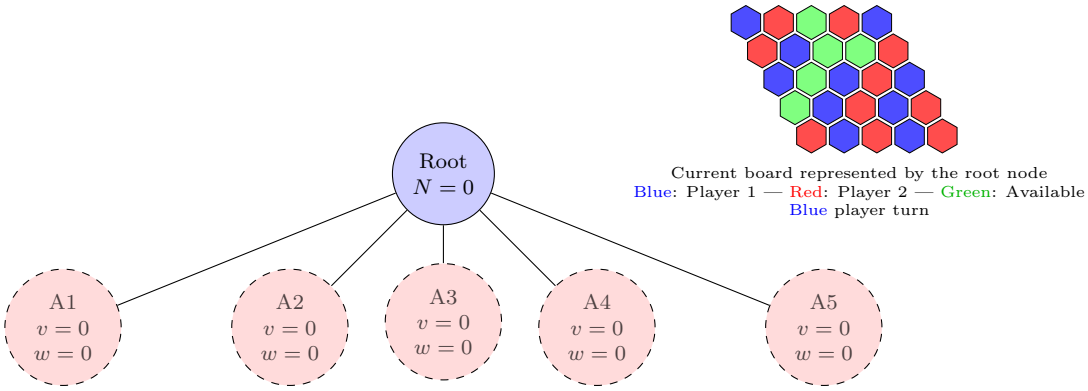
- v = number of visits to this node
- w = number of wins from the perspective of the player who moved to this node
- N = total visits to root (tracks total iterations)

[3] That is N, the total number of iterations, for node i that is a child of the root node.

Initial State (i.e. Current State)

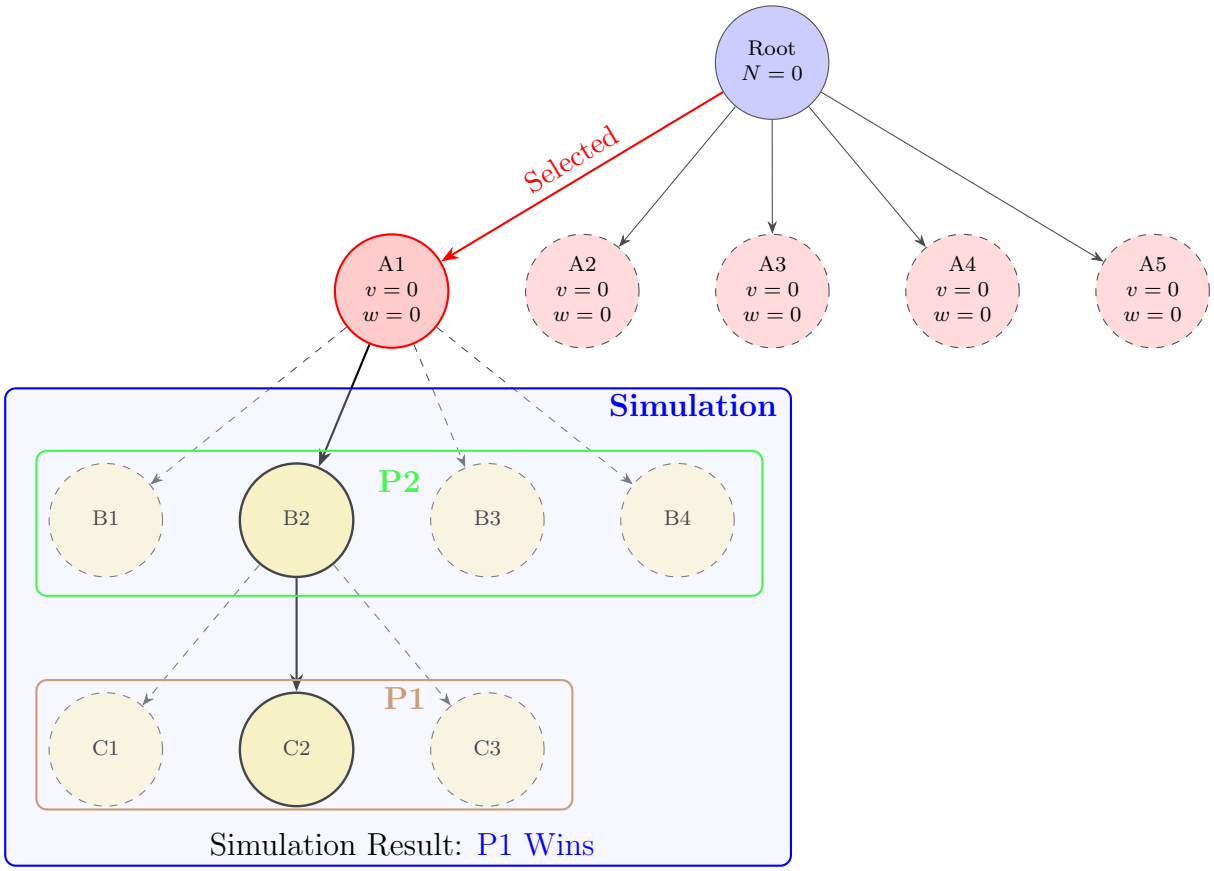
Let's consider that it is the turn of player 1 (Blue player) to play and that it has to choose between 5 free positions.

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).



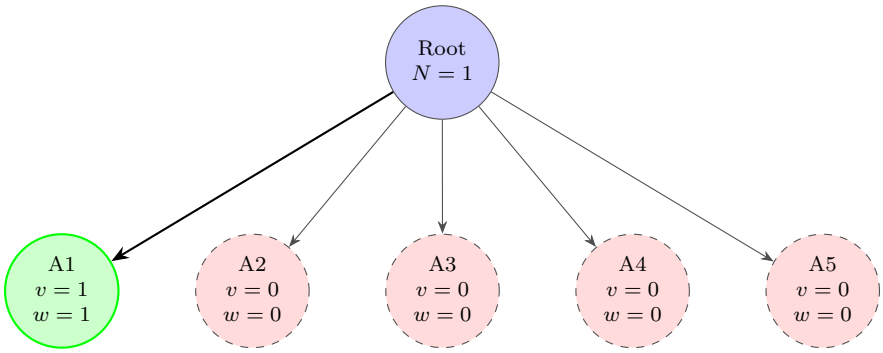
Iteration 1: First Expansion

All nodes have UCB1 = ∞ (division by zero). Select any of the possible move. Let's say child A1. Since A1 is selected the tree is expanded adding a branch to the tree. Then a simulation is performed considering that the board is in state represented by the node A1. During the simulation phase all moves are performed randomly. It is then the turn of player 2 to choose a position on the board among the 4 remaining ones. Suppose that it is the position represented by B2 that was randomly selected. If the game did not end then it is the turn again to player 1 to select a position among the 3 remaining ones. Suppose it is C2 that is selected. Then, let's say that with this last move, player 1 won. The simulation ends and the backpropagation is going to take place.



Backpropagation: P1 wins ⇒ A1: $v = 1$, $w = 1$, Root: $N = 1$
Explanation:

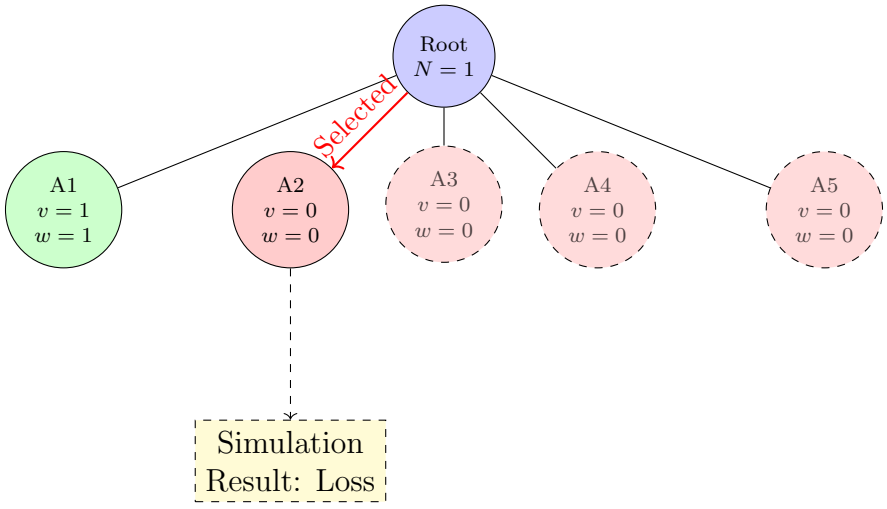
- A_1 gets $w = 1$ because it's a Player 1 move and Player 1 won
- A_1 gets $v = 1$ for being visited
- Root's N increases to count this iteration



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

Iteration 2: Second Expansion

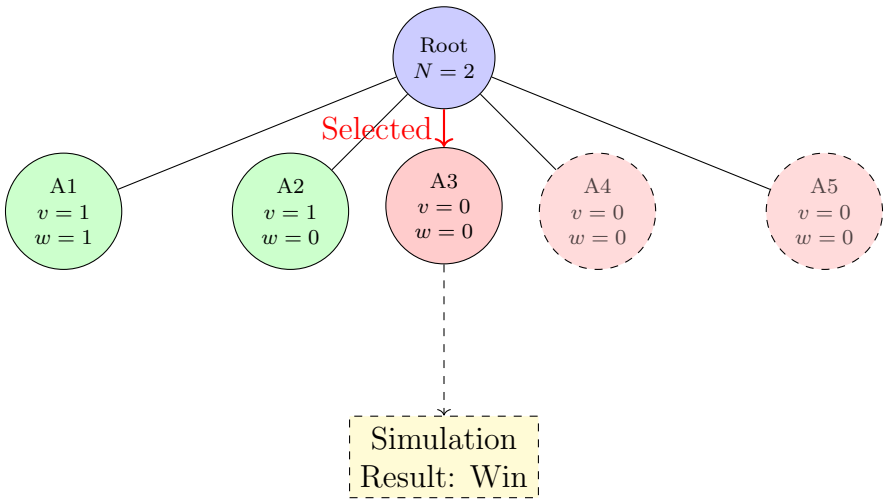
Unvisited nodes still have UCB1 = ∞. Randomly select A2 among the unvisited child A2, A3, A4 and A5.



Backpropagation: P1 loses ⇒ A2: $v = 1$, $w = 0$, Root: $N = 2$

Iteration 3: Third Expansion

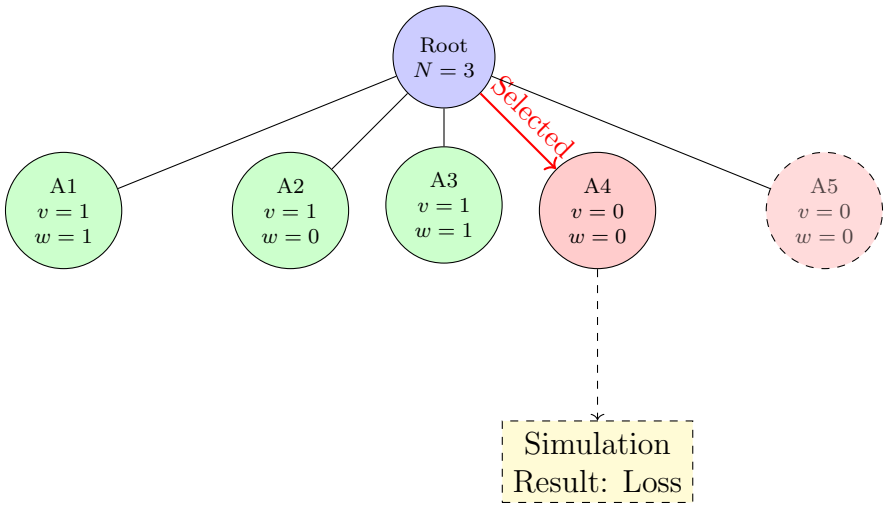
Select A3 (among the remaining unvisited node A3, A4, A5).



Backpropagation: P1 wins ⇒ A3: $v = 1$, $w = 1$, Root: $N = 3$

Iteration 4: Fourth Expansion

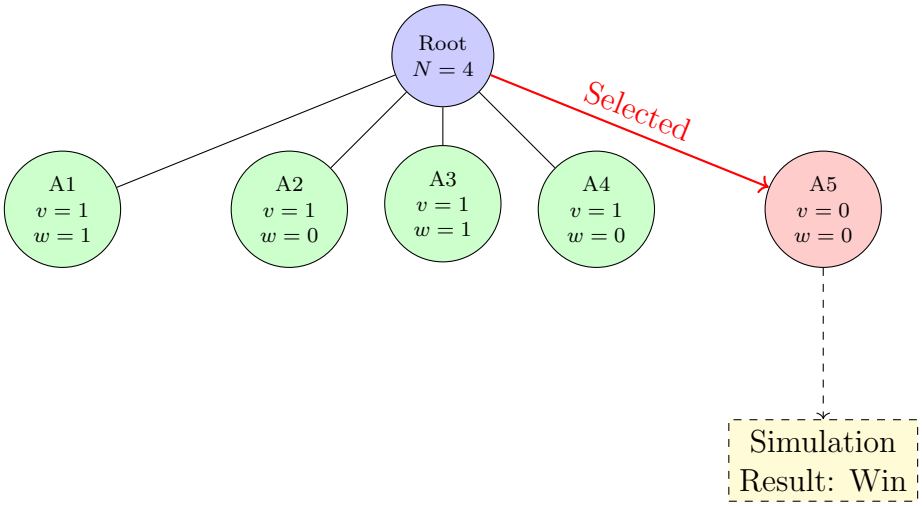
Select A4 (among the remaining unvisited node A4 and A5).



Backpropagation: P1 loses ⇒ A4: $v = 1$, $w = 0$, Root: $N = 4$

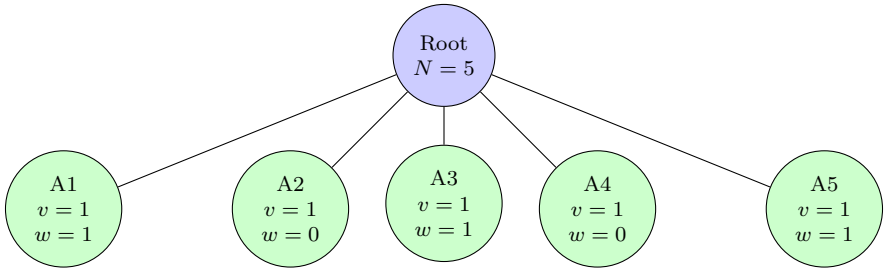
Iteration 5: Final Initial Expansion

Select A5 (last unvisited node).



Backpropagation: P1 wins ⇒ A5: $v = 1$, $w = 1$, Root: $N = 5$

Full expansion of the root node done!

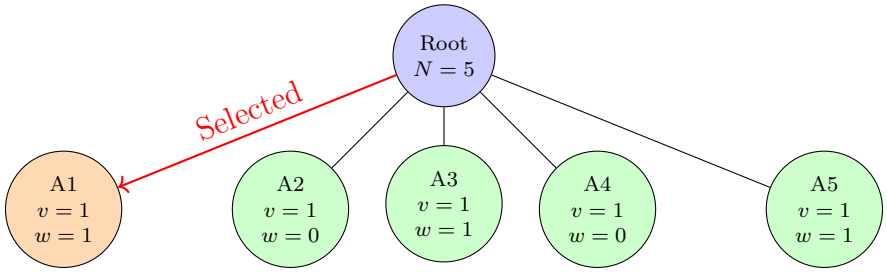


Iteration 6: First (real) UCB1 Selection

Now all nodes have been visited once (the root node is fully expanded and it is not a terminal node). Calculate UCB1 ($c = \sqrt{2}$):

$$\text{UCB1}(A1) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 1 + 1.79 = 2.79$$
$$\text{UCB1}(A2) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 0 + 1.79 = 1.79$$
$$\text{UCB1}(A3) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 1 + 1.79 = 2.79$$
$$\text{UCB1}(A4) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 0 + 1.79 = 1.79$$
$$\text{UCB1}(A5) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 5}{1}} = 1 + 1.79 = 2.79$$

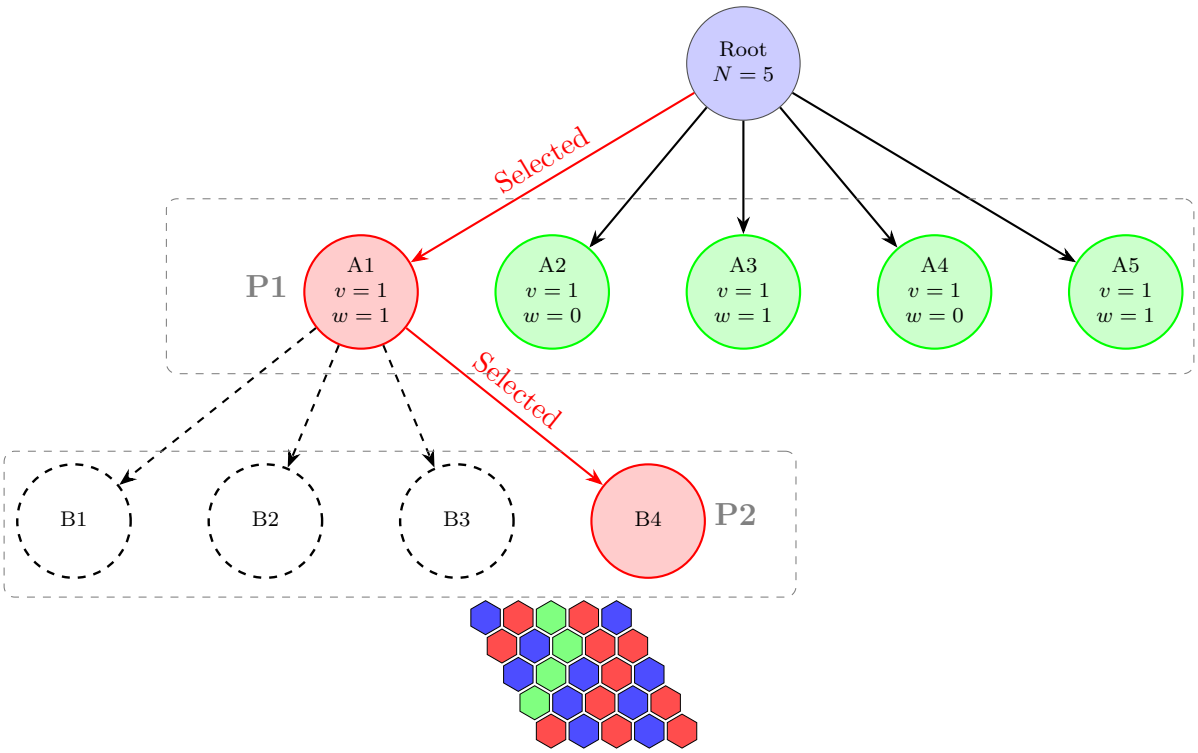
Tie between A1, A3, A5. Randomly select A1.



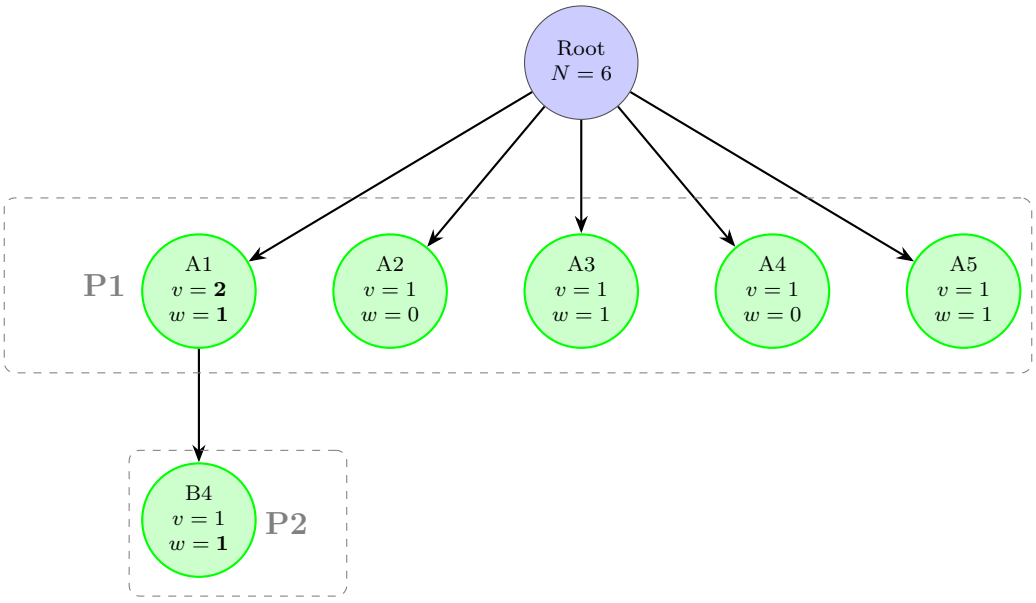
Then expand the selected child node A1. So, for the position that could occupy player 2, there is the choice between B1, B2, B3 and B4. Let's say B4 is selected (for example it is popped from the end of the list of available moves).

Has it happened, by selecting the move (represented by) B2, player 2 wins the game (hence player 1 lost it).
The tree is updated accordingly (backpropagation):

- B4: $v = 1$, $w = 1$
- A1: $v = v + 1 = 2$, $w = w + 0 = 1$
- N: $N = N + 1 = 6$



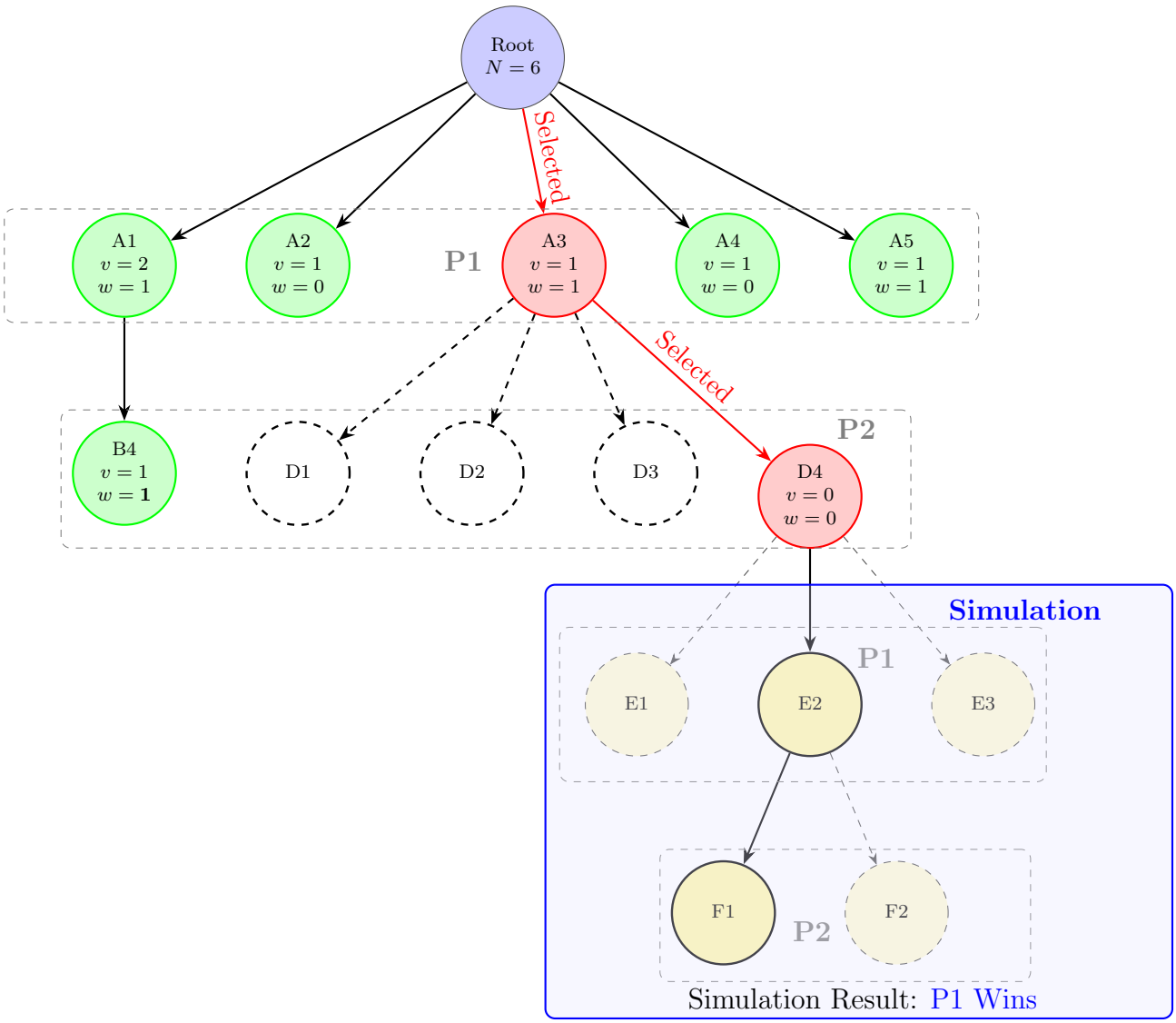
Key observation: B_4 gets $w = 1$ because it's a Player 2 move and Player 2 won. A_1 doesn't increase its win count because Player 1 lost. This is the minimax perspective in action!



Unless the time credit or iteration credit expired, the process goes on.
So, we are back to selection step, starting from the root node.
Calculate UCB1 ($c = \sqrt{2}$):

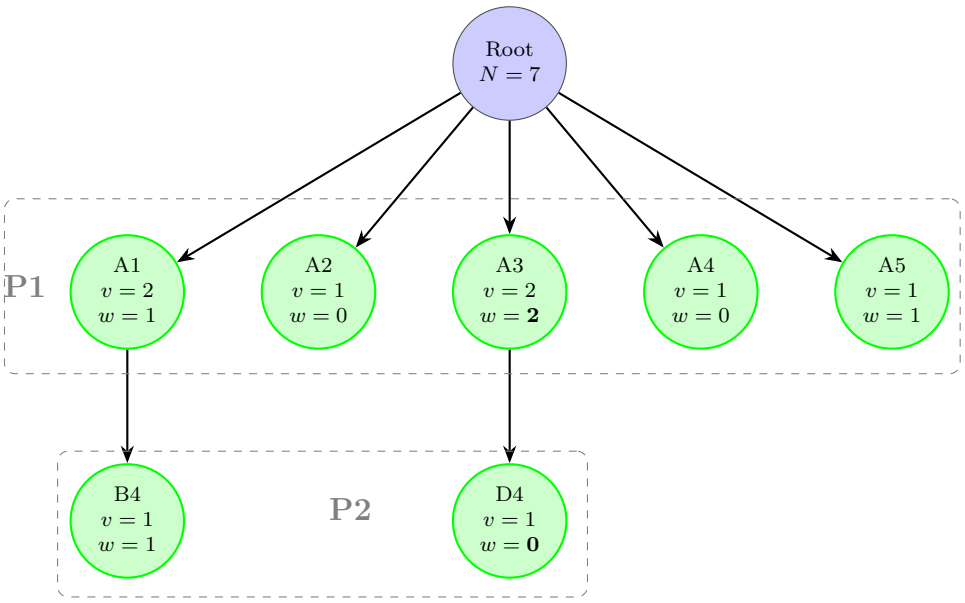
$$\text{UCB1}(A1) = \frac{1}{2} + \sqrt{2}\sqrt{\frac{\ln 6}{2}} = 0.5 + 1.89 = 2.39$$
$$\text{UCB1}(A2) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 6}{1}} = 0 + 1.89 = 1.89$$
$$\text{UCB1}(A3) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 6}{1}} = 1 + 1.89 = 2.89$$
$$\text{UCB1}(A4) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 6}{1}} = 0 + 1.89 = 1.89$$
$$\text{UCB1}(A5) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 6}{1}} = 1 + 1.89 = 2.89$$

Tie between A3, A5. Randomly select A3.
Then expand the selected child node A3. So, for the position that could occupy player 2, there is now the choice between D1, D2, D3 and D4.
Let's say D4 is selected.
The simulation then take place. There are 3 moves available for player 1, represented by E1, E2 and E3. Suppose E2 is chosen. There remains 2 moves possible for player 2, represented by F1 and F2. Let's say F1 is selected which terminate the game with the player 2 losing (i.e. player 1 wins).



Once the simulation is done, the backpropagation is performed.
Since player 1 won, we have the updates:

- D4: $v = 1$, $w = 0$
- A3: $v = v + 1 = 2$, $w = w + 1 = 2$
- N: $N = N + 1 = 7$



And we are ready to start a new iteration from the root node.

- $\text{UCB1}(A1) = \frac{1}{2} + \sqrt{2}\sqrt{\frac{\ln 7}{2}} = 0.5 + \sqrt{\ln 7} = 0.5 + 1.39496 = 1.895$
- $\text{UCB1}(A2) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 7}{1}} = 0 + \sqrt{2\ln 7} = 0 + 1.97299 = 1.973$
- $\text{UCB1}(A3) = \frac{2}{2} + \sqrt{2}\sqrt{\frac{\ln 7}{1}} = 1 + \sqrt{2\ln 7} = 1 + 1.97299 = 2.973$
- $\text{UCB1}(A4) = \frac{0}{1} + \sqrt{2}\sqrt{\frac{\ln 7}{1}} = 0 + \sqrt{2\ln 7} = 0 + 1.97299 = 1.973$
- $\text{UCB1}(A5) = \frac{1}{1} + \sqrt{2}\sqrt{\frac{\ln 7}{1}} = 1 + \sqrt{2\ln 7} = 1 + 1.97299 = 2.973$

We have 2 winners, A3 and A5. Let's say A3 is selected, then as it is not a terminal node and it is not fully expanded, one of its children (D1, D2, or D3) is going to be selected for the expansion of the tree[4].

This process continues for many iterations (typically hundreds to thousands), gradually building up the tree and improving win rate estimates.

Final Move Selection

After all iterations are complete, how do we decide which move to actually play? There are several common strategies:

Most Visits (Recommended)

Select the child of the root with the highest visit count:

$$\text{Best move} = \arg\max_i v_i$$

This is the most robust strategy because:

- Visit counts are more stable than win rates

[4]They all have $UCB1 = \infty$ so there is no preference.

- A highly-visited node has been thoroughly explored
- The UCB1 formula naturally directs visits toward promising moves

Highest Win Rate

Select the child with the highest win rate:

$$\text{Best move} = \arg\max_i \frac{w_i}{v_i}$$

This can be overly optimistic, especially for less-visited nodes. Generally less reliable than the visit-count method.

Robust Child (Max-Robust)

Select the child that maximizes a lower confidence bound:

$$\text{Best move} = \arg\max_i \left(\frac{w_i}{v_i} - c\sqrt{\frac{\ln N}{v_i}} \right)$$

This is a pessimistic approach that guards against uncertainty.

Example Final Selection

From our example after iteration 7, the visit counts are:

- A_1: $v = 2$, $w = 1$ (win rate: 50%)
- A_2: $v = 1$, $w = 0$ (win rate: 0%)
- A_3: $v = 2$, $w = 2$ (win rate: 100%)
- A_4: $v = 1$, $w = 0$ (win rate: 0%)
- A_5: $v = 1$, $w = 1$ (win rate: 100%)

With more iterations, one move would clearly emerge as having both high visit count and high win rate. The most-visited node strategy would be the standard choice.

Key Insights and Best Practices

Why MCTS Works

1. **No evaluation function needed**: Random playouts provide position estimates without domain knowledge
2. **Adaptive exploration**: UCB1 automatically balances exploring new moves and exploiting good ones
3. **Asymmetric tree growth**: More computational effort is spent on promising branches
4. **Any-time algorithm**: Can be stopped at any point and return the best move found so far

Practical Considerations

- **Exploration constant**: $c = \sqrt{2}$ is theoretically optimal for the multi-armed bandit problem, but can be tuned for specific games
- **Progressive widening**: For games with large branching factors, consider expanding only a subset of children initially
- **Domain knowledge**: While MCTS works with random playouts, incorporating heuristics in the simulation phase can improve performance
- **Parallelization**: Multiple simulations can run in parallel, though care must be taken with tree updates

When to Use MCTS

MCTS is particularly effective for:

- Games with large state spaces where minimax with alpha-beta pruning is impractical
- Domains where it's hard to design good evaluation functions
- Any-time scenarios where you need to return a reasonable move quickly but can improve given more time
- Two-player zero-sum games (though variants exist for other settings)

Conclusion

Monte Carlo Tree Search elegantly combines three powerful ideas:

- **Minimax structure** for adversarial reasoning
- **Monte Carlo evaluation** for estimating position values
- **UCB1 selection** for balancing exploration and exploitation

This combination has proven highly successful in game AI, most famously in AlphaGo's victory over human champions in Go. The algorithm's simplicity, effectiveness, and ability to work without domain-specific evaluation functions make it a powerful tool for a wide range of applications.