Monte-Carlo Tree Search

A Heuristic Search Algorithm for Decision Making

Intelligent Agents: Computational Game Solving

October 23, 2025

The Problem: Planning in Vast Search Spaces

What we've seen:

- Classical search algorithms like Minimax and Alpha-Beta
- These require a full-width search up to a certain depth
- They depend on a handcrafted **evaluation function** to score non-terminal positions

The problem:

- Chess: Branching factor $\sim$35 $\rightarrow$ search explodes
- Go: Branching factor $\sim$250 $\rightarrow$ **hopeless** for deep search
- Writing a good evaluation function is extremely difficult (domain expertise)

Key insight: Can we find good moves without exploring the whole tree or needing a static evaluation function?

The Core Idea: Smart Sampling

Classic Search (e.g., Minimax):

$$\text{value}(s) = \begin{cases} \text{utility}(s) & \text{if terminal} \\ \text{eval}(s) & \text{if depth limit} \\ \max_a \text{value}(s, a) & \text{otherwise} \end{cases}$$

Relies on a potentially flawed ‘eval‘ function. **Monte-Carlo Tree Search (MCTS):**

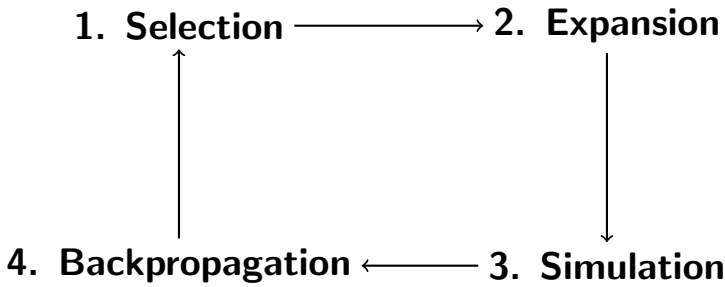
$$\text{value}(s) \approx \text{Average result of many random simulations from } s$$

Estimates value by playing games to the end. No evaluation function needed! **Trade-off:**

- Pro: No domain-specific evaluation function required. Focuses search on promising areas.
- Con: Estimates can be high variance. Can be misled by "trap states".
- Net result: Extremely effective in games with large search spaces.

The Four Steps of MCTS

MCTS grows a search tree one node at a time by repeating a four-step cycle.



For a fixed budget (e.g., 1 second or 10,000 iterations):

1. **Selection:** Start at the root, move down the tree to a leaf node.
2. **Expansion:** Add a new child node to the tree.
3. **Simulation:** From the new node, run a random "playout" to the end of the game.
4. **Backpropagation:** Use the result of the playout to update all nodes on the path back to the root.

Step 1: Selection & The UCB1 Formula

How do we choose which path to take down the tree?

- We need to balance **exploitation** (visiting nodes that look good) and **exploration** (visiting nodes we know little about).
- This is a classic multi-armed bandit problem.
- The standard solution is UCB1 (Upper Confidence Bound for Trees).

At a node p, select the child c that maximizes:

UCB1 Formula

$$\text{UCB1}(c) = \underbrace{\frac{v_c}{n_c}}_{\text{Exploitation}} + \underbrace{C \cdot \sqrt{\frac{\ln n_p}{n_c}}}_{\text{Exploration}}$$

- v_c: Total value (e.g., wins) accumulated at child c
- n_c: Number of times child c has been visited
- n_p: Number of times the parent p has been visited

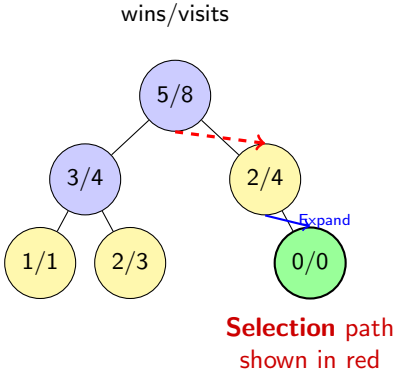
Visualizing the MCTS Cycle: Part 1

1. Selection

- Start at the root.
- Recursively apply UCB1 to select the most promising child.
- Continue until we reach a leaf node (a node that is part of the tree but has unvisited children).

2. Expansion

- Once a leaf node is selected, choose one of its unexplored children and add it to the tree.



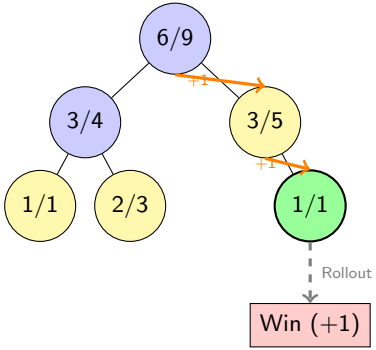
Visualizing the MCTS Cycle: Part 2

3. Simulation (Rollout)

- From the newly expanded node, play out the rest of the game.
- Actions are chosen using a simple, fast **default policy**.
- Most common default policy: choose actions uniformly at random.
- The simulation runs until a terminal state is reached.

4. Backpropagation

- The result of the simulation (e.g., +1 for a win, -1 for a loss, 0 for a draw) is "backed up" the tree.
- Every node along the selection path has



Stats updated along path

Final Move Selection

After the budget is exhausted (e.g., time's up):

- The algorithm has built a search tree that is asymmetrically focused on the most promising variations.
- The root node's children have statistics (v_c, n_c) summarizing thousands of simulations.
- How to choose the final move?

Common selection criteria:

1. **Most robust child:** The one with the most visits (n_c). This is the most explored and reliable option.
$$a^* = \arg\max_{a \in \text{children}} n_a$$
2. **Highest value child:** The one with the best win rate (v_c/n_c). This is more aggressive but can be higher variance.
$$a^* = \arg\max_{a \in \text{children}} \frac{v_a}{n_a}$$

MCTS: High-Level Pseudocode

```
1: function MCTS(root, num_iterations)
2:     for i = 1 → num_iterations do
3:         node ← SelectAndExpand(root)                ▷ Steps 1 & 2
4:         result ← Simulate(node)                     ▷ Step 3
5:         Backpropagate(node, result)                 ▷ Step 4
6:     end for
7:     return BestChild(root)                          ▷ Based on visits or value
8: end function

9: function SelectAndExpand(node)
10:     while node is not a terminal game state do
11:         if node has unexpanded children then
12:             return Expand(node)
13:         else
14:             node ← BestChildUCB(node)               ▷ Select using UCB1
15:         end if
16:     end while
17:     return node
```

Case Study: AlphaGo (DeepMind)

Game: Go (19x19 board) **Challenges:**

- Branching factor ~ 250, Game depth ~ 150
- State space $\sim 10^{170}$ (more than atoms in universe!)
- Traditional search methods had completely failed.

AlphaGo's breakthrough approach (Silver et al., 2016):

- **MCTS as the core search algorithm.**
- Used two deep neural networks to guide the search:
  - **Policy Network:** Instead of expanding one child, or trying all, this network predicts which moves are most promising. Used to guide *Selection* and *Expansion*.
  - **Value Network:** Instead of a random rollout, this network gives a quick estimate of the position's value. Replaces the *Simulation* step.

Result: First program to beat a top professional Go player. A landmark achievement for AI.

Strengths and Weaknesses

Strengths

- **Anytime algorithm:** Can be stopped at any time to return the current best move.
- **Asymmetric tree growth:** Focuses computation on more promising parts of the search space.
- **No evaluation function needed:** The simulation step replaces it.
- **Highly parallelizable:** Can run many simulations simultaneously.

Weaknesses

- **Needs many simulations:** Can be slow to converge in games where rollouts are not representative of good play.
- **Trap states:** Can be overly optimistic about a move if all random rollouts from it happen to avoid a deep tactical refutation.
- **Very long games:** If rollouts are very long, fewer iterations can be completed.
- **Knowledge-poor:** A pure MCTS doesn't understand "obvious" moves without many simulations.

Practical Enhancements

Improving on the basic MCTS algorithm:

1. **RAVE (Rapid Action Value Estimation):**
   - Also known as AMAF (All-Moves-As-First).
   - Idea: An action's value is influenced not only by simulations starting with it, but also by simulations where it was played *later* in the rollout.
   - Speeds up learning dramatically in early stages.
2. **Domain Knowledge:**
   - Use a non-random default policy (e.g., prefer captures in chess).
   - Prune obviously bad moves during expansion.
3. **Deep Learning Integration (like AlphaGo):**
   - Use a policy network to guide selection.
   - Use a value network to replace or mix with rollouts.
4. **Parallelization:**
   - **Root Parallelization:** Run multiple independent MCTS searches.
   - **Tree Parallelization:** Have multiple threads working on the same tree, requires locking.

Comparison: MCTS vs. Alpha-Beta

Feature	Alpha-Beta Search	Monte-Carlo Tree Search
Core requirement	Hand-crafted evaluation function	None (or a simple default policy)
Search shape	Full-width, uniform depth	Asymmetric, selective deepening
Output quality	Deterministic (for a given depth)	Stochastic, improves with more time
Best for	Games with smaller branching factors (Chess, Checkers)	Games with huge branching factors (Go, Hex)
Knowledge	Encoded in the evaluation function	Discovered through simulation
Anytime?	No (must complete search to depth D)	Yes (can be stopped anytime)

Summary: Monte-Carlo Tree Search

Key ideas:

- Intelligently build a search tree using results from random simulations.
- Balances **exploitation** and **exploration** using the UCB1 formula.
- Replaces the need for a complex, hand-crafted evaluation function.

The Four-Step Cycle:

- **Selection** $\rightarrow$ **Expansion** $\rightarrow$ **Simulation** $\rightarrow$ **Backpropagation**

Impact:

- Revolutionized computer Go, leading to superhuman performance (AlphaGo).
- State-of-the-art method for many board games, general game playing, and other planning problems.
- A powerful example of combining search with statistical sampling.

Further Reading

Key papers:

- **Kocsis & Szepesvári (2006):** "Bandit based Monte-Carlo Planning" (ECML)
  - The original UCT (UCB for Trees) paper.
- **Coulom (2006):** "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search"
  - Coined the term "Monte-Carlo Tree Search" and applied it to Go.
- **Silver et al. (2016):** "Mastering the game of Go with deep neural networks and tree search" (Nature)
  - The AlphaGo paper.

Excellent Surveys:

- Browne et al. (2012): "A Survey of Monte Carlo Tree Search Methods"
- Chaslot et al. (2008): "Monte-Carlo Tree Search: A New Framework for Game AI"