Monte Carlo Tree Search (MCTS)

Lecture Notes

Nadjib Lazaar
University of Paris-Saclay
lazaar@lisn.fr

Introduction

Monte Carlo Tree Search (MCTS) is a combinatorial optimization algorithm for decision-making in games and other domains. It relies on random simulations to explore decision spaces and identify optimal actions. This algorithm is widely used in games like Go, Chess, and video games.

1 History and Evolution of MCTS

MCTS is a tree exploration method combining statistical simulations based on Monte Carlo techniques with tree search approaches. It was initially developed to solve complex problems, particularly in perfect-information games such as Go.

Monte Carlo Methods: The concept of "Monte Carlo" originated from probabilistic methods developed in the 1940s as part of the *Manhattan Project*. These techniques were designed to estimate approximate solutions to problems where exhaustive search was infeasible.

1.1 Development of MCTS in the 2000s

Early Ideas: The idea of combining Monte Carlo simulations with tree search emerged in the 1990s, driven by the need to enhance performance in strategic games.

Key Advances (2006–2008):

- **2006:** MCTS was formalized in research on combinatorial games by Coulom, who introduced the concept of using Monte Carlo simulations to guide search.
- **2008:** The *Upper Confidence Bounds for Trees (UCT)* algorithm, proposed by Kocsis and Szepesvári, marked a major milestone by integrating an exploration-exploitation policy inspired by multi-armed bandits.

1.2 Major Applications

The Game of Go: One of the most significant breakthroughs for MCTS was its application to the game of Go, where classical algorithms like Minimax failed due to the immense combinatorial complexity. MCTS-based programs, such as `CrazyStone` and `MoGo`, achieved competitive levels of play.

The Era of Artificial Intelligence: MCTS was also a key component in systems like `AlphaGo` (2016), which combined MCTS with neural networks to defeat the best human Go players.

1.3 Current Research and Evolution

Today, MCTS remains an active research area, with applications extending beyond traditional games to fields such as:

- Planning in artificial intelligence.
- Autonomous decision-making systems.
- Resource optimization in computer networks.

MCTS continues to play a central role in the development of modern AI algorithms.

2 Key Concepts of MCTS

MCTS operates through four main steps, repeated to explore the game tree:

1. **Selection:** Traversing the existing tree by selecting nodes with the highest potential based on a criterion (such as UCB1).
2. **Expansion:** Adding new nodes to explore possible actions not yet simulated.
3. **Simulation:** Running a simulated game from a node to a terminal state, often with randomly chosen actions.
4. **Backpropagation:** Updating node statistics (rewards and visits) along the path back to the root.

3 Selection Criterion: UCB1

In MCTS, the selection of actions to explore is based on balancing **exploration** (examining less-explored actions to discover potentially better options) and **exploitation** (focusing on actions already showing promise). This balance is managed using the *Upper Confidence Bound 1* (**UCB1**) criterion.

3.1 UCB1 Formula

The formula for UCB1 is:

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

where:

- w_i: Cumulative reward obtained by following action a_i.
- n_i: Number of times action a_i has been simulated.
- N: Total number of simulations performed from the root.
- c: A control parameter, often empirically tuned, adjusting the balance between exploration and exploitation.

3.2 Interpretation of Terms

- **First Term: Exploitation**
  The first term $\frac{w_i}{n_i}$ represents the average reward obtained for action i, favoring actions that have performed well so far.
- **Second Term: Exploration**
  The second term $c\sqrt{\frac{\ln N}{n_i}}$ encourages exploration of less-tried actions. It grows when n_i is small, promoting exploration.
- **Parameter c**
  The parameter c plays a crucial role in balancing exploration and exploitation:
  - Large c: Greater emphasis on exploration, suitable for uncertain environments.
  - Small c: Prioritizes exploiting actions already showing promise.

3.3 Advantages of UCB1

Key advantages of UCB1 include:

- Simplicity and effectiveness in balancing exploration and exploitation.

- Asymptotic guarantees that the most promising actions are sufficiently explored.
- Dynamic adaptability to the total number of simulations N, making it robust across various problems.

3.4 Limitations and Extensions

While effective, UCB1 has some limitations:

- Requires careful tuning of c in complex reward distributions.
- Assumes normalized rewards, which may not hold in all applications.

Extensions, such as UCB-Tuned and domain-specific criteria, have been proposed to address these challenges.

4 MCTS Algorithm

Algorithm 1: MCTS Algorithm

Input: s_0: Initial state; n: Number of iterations.
Output: `bestAction`: The best action to take from the root state.

begin
  for $i \leftarrow 1$ ***to*** n **do**
    $s \leftarrow$ `Selection`(s_0)
    $s' \leftarrow$ `Expansion`(s)
    `result` $\leftarrow$ `Simulation`(s')
    `Backpropagation`$(s',$ `result`$)$
  return `BestAction`(s_0)

Function `Selection`(s)**:**
  while $\forall s' \in child(s) : isVisited(s')$ **do**
    $s \leftarrow BestUCB\text{-}Child(s)$
  return s `// Return the first unexplored node or the most promising leaf node`

Function `Expansion`(s)**:**
  pick $s' \in child(s) : \neg isVisisted(s')$
  if $s' \neq nil$ **then** $setAsVisited(s')$
  return s' `// Return the newly created child node`

Function `Simulation`(s)**:**
  while $\neg isLeaf(s)$ **do**
    pick $s' \in child(s)$
    $s \leftarrow s'$
  return EVAL(s)

Function `Backpropagation`$(s,$ `score`$)$**:**
  while $s \neq nil$ **do**
    $update\_nbVisits(s)$
    $update\_reward(s,$ `score`$)$
    $s \leftarrow parent(s)$
  return `// Backpropagation is complete`

Function `BestAction(root)`**:**
  $s^* \leftarrow \text{argmax}_{s' \in child(s)}(nbVisits(s'))$
  $a^* \leftarrow a_i : T(s_0, a_i) = s^*$
  return a^* `// Return the action corresponding to the most visited child of root`