MCTS-Reasoning: A Canonical Specification of Monte Carlo Tree Search for LLM Reasoning

Technical Report v0.5.2

January 25, 2026

Abstract

This technical report provides a rigorous specification of Monte Carlo Tree Search (MCTS) applied to Large Language Model (LLM) reasoning. We present formal definitions of the search tree structure, the four phases of MCTS (Selection, Expansion, Simulation, Backpropagation), and their adaptation for step-by-step reasoning with language models. Key design choices—including tree-building rollouts and terminal-only evaluation—are explicitly documented and justified. The goal is to establish a canonical reference specification that authentically captures the MCTS algorithm while adapting it for the unique characteristics of LLM-based reasoning tasks.

Contents

1 Introduction

Monte Carlo Tree Search (MCTS) is a best-first search algorithm that uses random sampling to build a search tree and evaluate actions [1, 2]. Originally developed for game playing—notably achieving superhuman performance in Go [4]—MCTS has proven effective in domains where the state space is too large for exhaustive search but where states can be evaluated through simulation.

In this report, we apply MCTS to *reasoning* with Large Language Models. The key insight is that multi-step reasoning can be viewed as a sequential decision problem:

- **States** are partial reasoning traces (text strings)
- **Actions** are reasoning continuations (next steps)
- **Terminal states** are complete solutions with answers
- **Rewards** are quality assessments of final answers

This formulation allows MCTS to systematically explore different reasoning paths, allocating more search effort to promising directions while maintaining exploration of alternatives. Recent work has explored similar ideas under the names "Tree of Thoughts" [5] and "Reasoning via Planning" [6].

1.1 Goals of This Specification

1. Provide **rigorous definitions** of all components
2. Present the **MCTS algorithm** with precise pseudocode
3. Specify the **adaptation to LLM reasoning** with clear interfaces
4. Document **design choices** that distinguish this variant from classical MCTS

1.2 Scope and Assumptions

This specification covers single-threaded MCTS for text-based reasoning. We assume:

- The LLM produces valid text output for each query
- The Evaluator returns scores in $[0, 1]$
- State strings have bounded length (context window limits)

2 Preliminaries

2.1 Notation

We use the following notation throughout:

Symbol	Meaning
$\mathcal{S}$	State space (set of all possible reasoning traces)
$\mathcal{A}$	Action space (set of all possible continuations)
$s \in \mathcal{S}$	A state (partial or complete reasoning trace)
$a \in \mathcal{A}$	An action (reasoning continuation)
$\mathcal{N}$	Set of nodes in the search tree
$n \in \mathcal{N}$	A node in the search tree
$\nu(n)$	Visit count of node n
$v(n)$	Cumulative value of node n
c	Exploration constant in UCB1
B	Branching factor bound (max children per node)
D	Maximum rollout depth

3 Formal Definitions

Definition 3.1 (State)**.** *A* state $s \in \mathcal{S}$ *is a string representing a partial or complete reasoning trace. A state consists of the concatenation of:*

1. *The original question*
2. *Zero or more reasoning steps*
3. *Optionally, a terminal marker with a final answer*

Definition 3.2 (Terminal State)**.** *A state s is* terminal *if a* terminal detector *determines it contains a complete answer. Formally:*

$$\textsc{IsTerminal}(s) = \textsc{TerminalDetector.Check}(s)$$

The detector also extracts the answer: $\textsc{ExtractAnswer}(s) \to \textit{Answer} \cup \{\bot\}$.

Definition 3.3 (Terminal Detector)**.** *A* terminal detector *is a component that determines when reasoning is complete:*

- $\textsc{Check}(s) \to \{\texttt{true}, \texttt{false}\}$*: determines if s is terminal*
- $\textsc{ExtractAnswer}(s) \to \textit{Answer}$*: extracts the answer from terminal s*
- $\textsc{FormatInstruction}() \to \textit{String}$*: provides prompt guidance for the LLM*

Remark 3.1 (Terminal Detector Implementations)**.** *The canonical implementation uses a* marker-based detector *that looks for the string "`ANSWER:`". Alternative implementations include:*

- ***BoxedDetector****: Looks for LaTeX-style* `\boxed{...}` *(common in math benchmarks)*
- ***MultiMarkerDetector****: Accepts multiple completion signals*
- ***LLM-as-Judge****: Asks an LLM whether reasoning is complete (more expensive)*

The terminal detector's $\textsc{FormatInstruction}$ *is included in prompts so the LLM knows the expected answer format.*

Definition 3.4 (Action)**.** *An* action *a is an operation that transforms a state into a new state. Formally:*

$$a : \mathcal{S} \to \mathcal{S}$$

The canonical action is $\textsc{Continue}$*, which prompts the LLM to generate the next reasoning step:*

$$\textsc{Continue}(s) = s \oplus \mathcal{G}(s)$$

where $\oplus$ denotes string concatenation with appropriate formatting.

Definition 3.5 (Action Space)**.** *The* action space $\mathcal{A}(s)$ *is the set of actions available at state* s*:*

$$\mathcal{A}(s) = \begin{cases} \emptyset & \text{if } \textsc{IsTerminal}(s) \\ \{\textsc{Continue}\} & \text{otherwise (canonical case)} \end{cases}$$

Remark 3.2 (State-Dependent Actions)**.** *In standard MCTS for games, different positions have different legal moves. Similarly, different reasoning states may have different available actions. The canonical implementation uses only* Continue*, but the* `ExtendedActionSpace` *class allows additional actions such as* Compress *for long traces.*

Definition 3.6 (Node)**.** *A* node n *in the search tree consists of:*

- $\mathit{state}(n) \in \mathcal{S}$*: the reasoning trace at this node*
- $\nu(n) \in \mathbb{N}_0$*: the number of times this node has been visited*
- $v(n) \in \mathbb{R}_{\geq 0}$*: the cumulative value from simulations through this node*
- $\mathit{children}(n) \subseteq \mathcal{N}$*: the set of child nodes*
- $\mathit{parent}(n) \in \mathcal{N} \cup \{\perp\}$*: the parent node (or* $\perp$ *for root)*
- $\mathit{terminal}(n) \in \{\texttt{true}, \texttt{false}\}$*: whether this is a terminal state*
- $\mathit{answer}(n)$*: the extracted answer (if terminal)*

Definition 3.7 (Search Tree)**.** *A* search tree $\mathcal{T} = (\mathcal{N}, E, r)$ *is a rooted tree where:*

- $\mathcal{N}$ *is the set of nodes*
- $E = \{(\mathit{parent}(n), n) : n \in \mathcal{N}, \mathit{parent}(n) \neq \perp\}$ *is the edge set*
- $r \in \mathcal{N}$ *is the root node with* $\mathit{parent}(r) = \perp$

The tree is an arborescence: every node except the root has exactly one parent, and there is a unique path from r *to any node.*

Definition 3.8 (Average Value)**.** *The* average value *of a node* n *with* $\nu(n) > 0$ *is:*

$$\bar{v}(n) = \frac{v(n)}{\nu(n)}$$

For $\nu(n) = 0$*, the average value is undefined.*

Definition 3.9 (Branching Factor Bound)**.** *The* branching factor bound $B \in \mathbb{N}$ *is the maximum number of children any node may have. A node* n *is* fully expanded *if and only if:*

$$\textsc{FullyExpanded}(n) = \big(|\mathit{children}(n)| \geq B\big) \vee \textsc{IsTerminal}(\mathit{state}(n))$$

4 The MCTS Algorithm

MCTS builds a search tree incrementally through repeated *simulations*. Each simulation consists of four phases: Selection, Expansion, Rollout, and Backpropagation.

Algorithm 1 MCTS Main Loop

Require: Question q, number of simulations K, branching factor B, max depth D
Ensure: Search tree with root r

procedure SEARCH(q, K, B, D)
  $r \leftarrow$ CREATENODE(q) ▷ Root state is the question
  for $i = 1$ **to** K **do**
    $n_{\text{leaf}} \leftarrow$ SELECT(r)
    $n_{\text{exp}} \leftarrow$ EXPAND(n_{leaf}, B)
    $n_{\text{term}}, v \leftarrow$ ROLLOUT(n_{exp}, D)
    BACKPROPAGATE(n_{term}, v)
  end for
  return r
end procedure

4.1 Phase 1: Selection

Selection traverses the tree from root to a leaf, choosing children according to the UCB1 policy.

Definition 4.1 (UCB1)**.** *For a non-root node n with parent $p = \text{parent}(n)$, the* Upper Confidence Bound *is:*

$$UCB1(n) = \begin{cases} +\infty & \text{if } \nu(n) = 0 \\ \bar{v}(n) + c\sqrt{\dfrac{\ln \nu(p)}{\nu(n)}} & \text{if } \nu(n) > 0 \end{cases}$$

where $c > 0$ is the exploration constant. The first term $\bar{v}(n)$ is the exploitation *component; the second term is the* exploration *component.*

Remark 4.1 (Exploration Constant)**.** *The theoretical value $c = \sqrt{2}$ derives from the UCB1 bandit algorithm [1]. In practice, c may be tuned: larger values encourage exploration, smaller values favor exploitation.*

Remark 4.2 (Tie-Breaking)**.** *When multiple children have equal UCB1 values (including multiple unvisited children with $UCB1 = +\infty$), we select uniformly at random among them.*

Algorithm 2 Selection Phase

Require: Node n (typically root)
Ensure: Leaf node for expansion

procedure SELECT(n)
  while FULLYEXPANDED(n) **and** children(n) $\neq \emptyset$ **do**
    $n \leftarrow \arg\max_{c \in \text{children}(n)} \text{UCB1}(c)$ ▷ Break ties randomly
  end while
  return n
end procedure

4.2 Phase 2: Expansion

Expansion adds a new child node by generating a continuation from the Generator.

Algorithm 3 Expansion Phase

Require: Node n, branching factor bound B
Ensure: Expanded node (new child or n if unexpandable)

procedure EXPAND(n, B)
  if ISTERMINAL(state(n)) **then**
    return n ▷ Cannot expand terminal nodes
  end if
  if $|\text{children}(n)| \geq B$ **then**
    return n ▷ Fully expanded
  end if
  $a \leftarrow \mathcal{G}.\text{GENERATE}(\text{state}(n))$ ▷ Get one continuation
  $s' \leftarrow \text{state}(n) \oplus a$
  $n' \leftarrow \text{CREATENODE}(s')$
  $\text{parent}(n') \leftarrow n$
  $\text{children}(n) \leftarrow \text{children}(n) \cup \{n'\}$
  return n'
end procedure

4.3 Phase 3: Rollout (Simulation)

The rollout phase continues from the expanded node until reaching a terminal state or the maximum depth.

Remark 4.3 (Design Choice: Tree-Building Rollout)**.** *In classical MCTS for games, rollouts typically use a fast, random policy* without *adding nodes to the tree. However, for LLM reasoning, we make a deliberate design choice to* ***add rollout nodes to the tree****. This preserves the full reasoning trace and allows:*

1. *Reuse of reasoning steps in future simulations*
2. *Inspection of the complete reasoning path*
3. *Consistent state representation throughout the tree*

This is sometimes called "tree policy all the way" or "persistent tree" MCTS.

Algorithm 4 Rollout Phase (Tree-Building)

Require: Node n, maximum depth D
Ensure: Terminal node n_{term} and its value v

procedure ROLLOUT(n, D)
  current $\leftarrow n$
  depth $\leftarrow 0$
  while $\neg$ISTERMINAL(state(current)) **and** depth $< D$ **do**
    $a \leftarrow \mathcal{G}$.GENERATE(state(current))
    $s' \leftarrow$ state(current) $\oplus\, a$
    $n' \leftarrow$ CREATENODE(s')
    parent(n') $\leftarrow$ current
    children(current) $\leftarrow$ children(current) $\cup \{n'\}$
    current $\leftarrow n'$
    depth $\leftarrow$ depth $+ 1$
  end while
  if ISTERMINAL(state(current)) **then**
    $v \leftarrow \mathcal{E}$.EVALUATE(state(current), answer(current))
  else
    $v \leftarrow 0$ ▷ No terminal reached within depth limit
  end if
  return (current, v)
end procedure

4.4 Phase 4: Backpropagation

Backpropagation updates visit counts and values along the path from the terminal node to the root.

Algorithm 5 Backpropagation Phase

Require: Terminal node n_{term}, value v

procedure BACKPROPAGATE(n_{term}, v)
  $n \leftarrow n_{\text{term}}$
  while $n \neq \perp$ **do**
    $\nu(n) \leftarrow \nu(n) + 1$
    $v(n) \leftarrow v(n) + v$
    $n \leftarrow$ parent(n)
  end while
end procedure

Property 4.1 (Visit Count Invariant)**.** *For any node n in the tree:*

$$\nu(n) = \sum_{c \in \mathit{children}(n)} \nu(c) + \mathbb{1}[n \text{ is a rollout endpoint}]$$

where $\mathbb{1}[\cdot]$ is the indicator function. In particular, $\nu(\mathit{root}) = K$ after K simulations.

5 Application to LLM Reasoning

The MCTS algorithm requires two external components: a *Generator* to produce continuations and an *Evaluator* to score terminal states.

5.1 Generator

Definition 5.1 (Generator)**.** *A* Generator $\mathcal{G}$ *is a function that produces reasoning continuations:*

$$\mathcal{G} : \mathcal{S} \to \mathcal{A}$$

Given a state s, it returns a continuation a. The continuation may result in a terminal state (if it contains the answer marker) or a non-terminal state (requiring further reasoning).

For LLM-based generation, we use the following procedure:

Algorithm 6 LLM Generator

Require: State s, temperature τ
Ensure: Continuation a

```
procedure GENERATE(s)
    prompt ← FORMATPROMPT(s)
    response ← LLM(prompt, temperature = τ)
    a ← PARSECONTINUATION(response)
    return a
end procedure
```

Definition 5.2 (Prompt Template)**.** *The prompt template formats the current state for the LLM:*

```
You are solving a problem step by step.
Current reasoning:
{state}

Continue with the next step. When you reach a final answer,
write "ANSWER: " followed by your answer.

Next step:
```

The temperature $\tau > 0$ controls diversity: higher temperatures yield more varied continuations.

5.2 Evaluator

Definition 5.3 (Evaluator)**.** *An* Evaluator $\mathcal{E}$ *is a function that scores terminal states:*

$$\mathcal{E} : \mathcal{S} \times \mathit{Answer} \to [0, 1]$$

Given a terminal state and its extracted answer, it returns a quality score.

Remark 5.1 (Terminal-Only Evaluation)**.** *The Evaluator is invoked* only *on terminal states. This design choice reduces computational cost: intermediate reasoning states are not evaluated, and LLM-as-judge calls occur only when a complete answer is produced.*

Algorithm 7 LLM Evaluator (LLM-as-Judge)

Require: Terminal state s, extracted answer a
Ensure: Score $\in [0, 1]$

```
procedure EVALUATE(s, a)
    prompt ← FORMATJUDGEPROMPT(s, a)
    response ← LLM(prompt, temperature = 0)
    score ← PARSESCORE(response)
    return score
end procedure
```

Alternative Evaluators:

- **Ground Truth**: Compare answer to known correct answer with normalization (lowercase, strip punctuation). Supports partial credit when answer contains the truth or vice versa.
- **Numeric**: For mathematical problems, compares numeric values with configurable tolerance (absolute and relative). Extracts numbers from text, handles fractions (e.g., "1/2"), scientific notation, and provides partial credit based on relative error.
- **Process**: Evaluates reasoning quality using heuristics—presence of step-by-step structure, mathematical notation, verification statements, and logical connectives. Can be combined with an answer evaluator via weighted sum.
- **Composite**: Weighted combination of multiple evaluators. For example: $0.7 \times$NumericScore$+$ $0.3 \times$ ProcessScore.
- **Self-Consistency**: Score based on agreement with other sampled solutions (not implemented in canonical version).

6 Result Extraction

After search completes, we extract the best answer from the tree.

Algorithm 8 Best Answer Extraction

Require: Root node r
Ensure: Best answer and confidence score

procedure GetBestAnswer(r)
  $n \leftarrow r$
  while children(n) $\neq \emptyset$ **do**
    $n \leftarrow \arg\max_{c \in \text{children}(n)} \nu(c)$ ▷ Most-visited child
  end while
  if IsTerminal(state(n)) **then**
    return (answer(n), $\bar{v}(n)$)
  else
    return (`null`, 0)
  end if
end procedure

Remark 6.1. *Following most-visited children (rather than highest-value) is standard practice in MCTS, as visit counts are more robust to value noise than raw averages.*

7 Complexity Analysis

Property 7.1 (Time Complexity)**.** *For K simulations with maximum depth D:*

- ***Selection**: $O(D)$ node traversals per simulation*
- ***Expansion**: $O(1)$ plus one Generator call*
- ***Rollout**: $O(D)$ Generator calls (worst case)*
- ***Backpropagation**: $O(D)$ node updates*

Total tree operations: $O(KD)$. The dominant cost is typically LLM calls: up to $O(KD)$ Generator calls and $O(K)$ Evaluator calls.

Property 7.2 (Space Complexity)**.** *The search tree contains at most $O(KD)$ nodes, each storing a state string. With bounded state length L, space complexity is $O(KDL)$.*

8 Additional Capabilities

This section describes additional capabilities implemented in the canonical MCTS-Reasoning system.

8.1 Tool-Augmented Reasoning (MCP)

Tool-augmented reasoning is supported via the Model Context Protocol (MCP):

- **ToolAwareGenerator**: Wraps any Generator with tool support, intercepting tool calls in LLM output and executing them before continuing generation.
- **ToolContext**: Manages MCP server connections, tool discovery, and execution. Provides a unified interface for tool invocation.
- **Native Function Calling**: Direct API integration for OpenAI and Anthropic providers, using their native tool-use APIs for improved reliability.
- **RAG Server**: Built-in MCP server for retrieval-augmented guidance, exposing tools for retrieving similar examples and domain-specific guidance.

Tools are invoked during reasoning steps via XML or JSON format:

```
<tool_call name="calculator">
  <expression>15 * 7 + 23</expression>
</tool_call>
```

Results are injected back into the state, allowing the LLM to continue reasoning with tool outputs.

8.2 Path Sampling and Self-Consistency

Beyond extracting the single best answer, path sampling strategies support analysis of the search tree:

- **Value-based**: Sample paths with highest average values
- **Visit-based**: Sample most frequently visited paths (robust to noise)
- **Diverse**: Maximize answer diversity across samples
- **Top-k**: Sample k best paths by specified criterion

Self-consistency voting aggregates across multiple terminal states to improve answer reliability:

- **Majority vote**: $\arg\max_a |\{p : \text{answer}(p) = a\}|$ — simple count of each answer
- **Weighted vote**: $\arg\max_a \sum_{p:\text{answer}(p)=a} \bar{v}(p)$ — answers weighted by path values

The `PathSampler` class provides these operations:

```
sampler = PathSampler(result.root)
paths = sampler.sample(n=5, strategy="diverse")
answer, confidence = sampler.majority_vote()
```

8.3 Tree Serialization and Continuation

Search trees can be serialized to JSON and resumed, enabling:

- Incremental search (add more simulations to existing tree)
- Tree inspection and debugging
- Checkpointing long-running searches
- Sharing trees between sessions

The operations are:

- $\textsc{Save}(\mathcal{T}, \text{path})$: Serialize tree structure, visit counts, values, and terminal states to JSON
- $\textsc{Load}(\text{path}, \mathcal{G}, \mathcal{E}) \to \mathcal{T}$: Restore tree from file, reconnecting to new Generator/Evaluator instances
- $\textsc{ContinueSearch}(K)$: Add K more simulations to the loaded tree

Example usage:

```
# Save after initial search
result = mcts.search("What is 2+2?", simulations=50)
mcts.save("tree.json")

# Later: load and continue
mcts = MCTS.load("tree.json", generator, evaluator)
result = mcts.continue_search(simulations=50)  # Now 100 total
```

9 Conclusion

This report establishes a canonical specification for MCTS applied to LLM reasoning. The key contributions are:

1. **Rigorous formal definitions** of states, actions, nodes, and the search tree
2. **Precise specification** of the four MCTS phases with pseudocode
3. **Clear interfaces** for Generator and Evaluator components
4. **Explicit documentation** of design choices (tree-building rollouts, terminal-only evaluation, bounded branching)
5. **Practical capabilities** including tool integration, path sampling, and tree serialization

The specification serves as a reference for implementation, ensuring the algorithm is authentically represented while being appropriately adapted for language model reasoning tasks.

Future research directions—including extended action spaces (e.g., Compress, Verify), algorithm variants (parallel MCTS, learned value functions), and graph-based reasoning structures—are discussed in a companion working paper on research directions for MCTS-based LLM reasoning.

References

[1] Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. *European Conference on Machine Learning*, 282–293.

[2] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. *International Conference on Computers and Games*, 72–83.

[3] Browne, C. B., et al. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 1–43.

[4] Silver, D., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484–489.

[5] Yao, S., et al. (2023). Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

[6] Hao, S., et al. (2023). Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

A Implementation Notes

This appendix provides practical implementation details for the canonical MCTS-Reasoning system.

A.1 State Lifecycle

States are accumulated text strings. The transition from initial to terminal state follows this pattern:

```
Initial State (root):
  "Question: What is 15*7+23?
   Let me solve this step by step."
          |
    [CONTINUE action]
          v
Child State:
  "Question: What is 15*7+23?
   Let me solve this step by step.
   Step 1: First, calculate 15*7 = 105"
          |
    [CONTINUE action]
          v
Terminal State:
  "Question: What is 15*7+23?
   ...
   Step 2: Then add 23: 105 + 23 = 128
   ANSWER: 128"
```

Key Properties:

Property	Status	Notes
Well-defined	Yes	Strings with clear semantics
Markov	Yes	Generator sees only current state
Traceable	Yes	Parent pointers reconstruct full path
Bounded	No	States grow unboundedly (limitation)

A.2 UCB1 Visualization

The UCB1 formula balances exploration and exploitation:

```
UCB1 = average_value + c * sqrt(ln(parent_visits) / visits)
       -------------   ------------------------------------
       EXPLOITATION              EXPLORATION
```

- Unvisited nodes have UCB1 = $+\infty$ (highest priority)
- High-value, low-visit nodes are attractive (explore more)
- High-value, high-visit nodes are reliable (exploit)
- Low-value nodes are deprioritized regardless of visits

Default exploration constant: $c = \sqrt{2} \approx 1.414$. Higher values encourage exploration; lower values favor exploitation.

A.3 Evaluator Comparison

Evaluator	Scoring Method	Best For
NumericEvaluator	Compare to ground truth with tolerance	Math problems
GroundTruthEvaluator	Exact/partial string match	Known answers
ProcessEvaluator	Heuristic scoring of reasoning quality	Open-ended
LLMEvaluator	LLM-as-judge (0–1 score)	Subjective quality
CompositeEvaluator	Weighted combination	Multiple criteria

NumericEvaluator Scoring:

- Exact match (within tolerance): 1.0
- Within 10% relative error: 0.8
- Within 50% relative error: 0.5
- Beyond 50% error: 0.0

A.4 Code Location Reference

Primary implementation files in the `mcts_reasoning/` package:

Component	File
Node (UCB1, tree structure)	`node.py`
MCTS (search algorithm)	`mcts.py`
Generator (LLM continuation)	`generator.py`
Evaluator (terminal scoring)	`evaluator.py`
Terminal detection	`terminal.py`
Actions (Continue, Compress)	`actions.py`
Path sampling	`sampling.py`
MCP tool integration	`tools/`
LLM providers	`compositional/providers.py`
RAG stores	`compositional/rag.py`