Dynamic Parallel Tree Search for Efficient LLM Reasoning

Yifu Ding^{1,2,4}, Wentao Jiang^{3}, Shunyu Liu^{2}, Yongcheng Jing^{2}, Jinyang Guo^{4,5}, Yingjie Wang^{2} Jing Zhang^{3}, Zengmao Wang^{3,*}, Ziwei Liu^{2}, Bo Du^{3}, Xianglong Liu^{4,1,*}, Dacheng Tao^{2,*}

[1] School of Computer Science and Technology, Beihang University, China [2]Nanyang Technological University, Singapore [3]School of Computer Science, Wuhan University, China [4]State Key Laboratory of Complex & Critical Software Environment, Beihang University, China [5]School of Artificial Intelligence, Beihang University, China

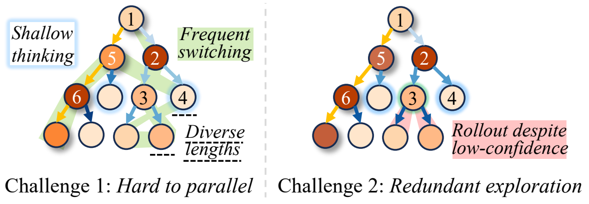
Correspondence: wangzengmao@whu.edu.cn, xlliu@buaa.edu.cn, dacheng.tao@gmail.com.

Abstract

Tree of Thoughts (ToT) enhances Large Language Model (LLM) reasoning by structuring problem-solving as a spanning tree. However, recent methods focus on search accuracy while overlooking computational efficiency. The challenges of accelerating the ToT lie in the frequent switching of reasoning focus, and the redundant exploration of suboptimal solutions. To alleviate this dilemma, we propose Dynamic Parallel Tree Search (DPTS), a novel parallelism framework that aims to dynamically optimize the reasoning path in inference. It includes the Parallelism Streamline in the generation phase to build up a flexible and adaptive parallelism with arbitrary paths by cache management and alignment. Meanwhile, the Search and Transition Mechanism filters potential candidates to dynamically maintain the reasoning focus on more possible solutions with less redundancy. Experiments on Qwen-2.5 and Llama-3 on math and code datasets show that DPTS significantly improves efficiency by 2-4× on average while maintaining or even surpassing existing reasoning algorithms in accuracy, making ToT-based reasoning more scalable and computationally efficient. Codes are released at: https://github.com/yifu-ding/DPTS.

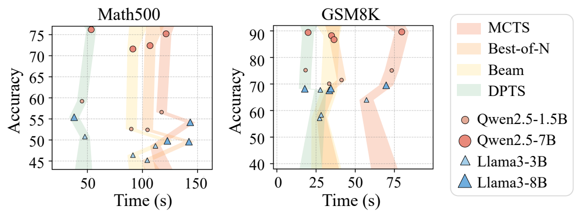
1 Introduction

The advent of OpenAI-o1 (Jaech et al., 2024), a reasoning large language model, has sparked significant interest in the academic community. A key factor behind its success is the Chain-of-Thought (CoT)-based reasoning technique (Wei et al., 2022; Chu et al., 2023; Liu et al., 2025), which improves model's reasoning ability by breaking complex problems into explicit intermediate steps. Building upon this, the Tree of Thoughts (ToT) (Yao et al., 2023) framework has



Shallow thinking
Frequent switching
Diverse lengths
Rollout despite low-confidence
1
2
3
4
5
6
Challenge 1: *Hard to parallel*
Challenge 2: *Redundant exploration*

(a) Challenges of implementing parallelism in reasoning tasks.



(b) Accuracy and efficiency comparisons across tree search algorithms on various models and datasets.

Figure 1: (a) The demonstration of challenges. (b) The experimental results.

been introduced to further elevate LLMs' reasoning capacities, and be widely used in multi-step reasoning tasks (Plaat et al., 2024). ToT restructures reasoning as a tree search process and employs search algorithms, such as Monte Carlo Tree Search (MCTS) (Chaslot et al., 2008; Xie et al., 2024; Yao et al., 2024a), to construct a tree-like structure that explores various reasoning pathways, leading to more refined and accurate responses (Sprueill et al., 2023; Zhang et al., 2025). However, current ToT approaches predominantly focus on improving search accuracy while overlooking computational efficiency (Xie et al., 2024; Cheng et al., 2024; Zhao et al., 2024). We conclude two significant challenges that complicate the acceleration of ToT.

The first challenge arises from the frequent switching of reasoning focus in traditional sequential tree search, making it difficult to effectively parallelize (Snell et al., 2024). Unlike conventional deep learning inference that is compatible with end-to-end parallelism, tree search has irregular computational trajectories. As shown in Fig-

[*]Corresponding author.

ure 1a Challenge 1, it frequently switches among paths, including retrospect and recursion behaviors which complicates the parallelism (green trajectory). For example, if parallelizing nodes 6 and 2 in one generation, the different context lengths and KV cache require additional processing. Moreover, frequent switching also tends to yield shallow exploration (Wang et al., 2025). Due to the limited time and memory budgets, more explored paths mean less exploitation in deep paths.

The second challenge stems from the redundant explorations on suboptimal solutions (Li et al., 2024b; Besta et al., 2024). Previous tree search methods fail in identifying the less potential path and terminating it timely (Wan et al., 2024a). Methods like MCTS attempt to balance the exploitation and exploration paths during node selection, but the selected nodes continue to roll out till the termination conditions (time or token limits) even with small prior confidence (Xie et al., 2024). For example, dark nodes in Figure 1a Challenge 2 have higher confidence. Node 3 with a lower confidence than node 5 will be explored earlier (pink trajectory). However, we observe that paths with low prior confidence have less probability of reaching the optimal solution, as illustrated in Section 3.2.

To address these challenges, we propose a novel and efficient tree search framework, **DPTS (Dynamic Parallel Tree Search)**. This framework implements parallelized tree search and optimizes it by dynamically adjusting reasoning focus during the tree growing, thereby improving computational efficiency. It consists of two key components for both the generation and selection phases. (1) DPTS implements a Parallelism Streamline tailored for LLM reasoning in the generation phase. It facilitates the rollout for arbitrary paths in parallel, allowing the expanded nodes to be rearranged at each step. Additionally, we carefully engineer cached data collection and context alignment, paving the way for parallelized inference with varying path length and node selection. (2) Building on this, to prevent excessive exploitation and focus the reasoning on more potential paths, we introduce a Search and Transition Mechanism in the selection phase. It dynamically balances the exploitation-exploration paths by the bidirectional transition, i.e., *Early Stop* (Exploitation $\rightarrow$ Exploration), *Deep Seek* (Exploration $\rightarrow$ Exploitation), allowing the model to focus on the most promising solutions and mitigate inefficient exploitation on suboptimal solutions.

Our work provides valuable insights into accelerating the ToT for LLM reasoning, paving the way for future work to solve real-world challenges. Our contributions can be concluded as follows:

- We propose the DPTS framework, which solve the frequent switching and redundant exploration issues in previous tree search methods for LLM reasoning.
- The Parallelism Streamline provides a flexible and efficient generation in node parallel, which bridges the gap between the sequential tree structure and parallelized inference.
- The Search and Transition Mechanism exploits more promising solutions and reduces unnecessary exploitation, ensuring that high-confidence nodes receive deeper reasoning.
- Experiments shows that DPTS reaches the best solution with less inference time across various models and widely used reasoning datasets, as plotted in Figure 1b.

2 Related Work

Reasoning with LLMs. LLMs have evolved from System 1 tasks (e.g., translation) (Brown et al., 2020) to System 2 reasoning (e.g., math, logic) (Kojima et al., 2022). CoT (Wei et al., 2022) enhances multi-step reasoning, with variants like Self-Consistent CoT (Wang et al., 2022), but its exploration scope remains constrained, limiting its effectiveness. (Chu et al., 2023). Furthermore, ToT (Yao et al., 2023) enables multi-path exploration, leveraging MCTS (Chaslot et al., 2008) for backtracking and heuristic rollouts (Wan et al., 2024b; Wang et al., 2024a). However, MCTS remains computationally expensive, with limited work on acceleration methods.

LLM Inference Acceleration. While LLM inference has been optimized for linear decoding (Lin et al., 2024), tree-structured reasoning remains underexplored (Li et al., 2024a). Approaches like Deft (Yao et al., 2024b) optimize prefix sharing, while others use self-consistency for early stopping (Li et al., 2024b). Recent work such as SEED (Wang et al., 2024b) explores speculative decoding by parallelizing draft generation with small models. In conclusion, efficient tree search for LLM reasoning remains an open challenge.

We have included a more detailed discussion of related work in Appendix D.

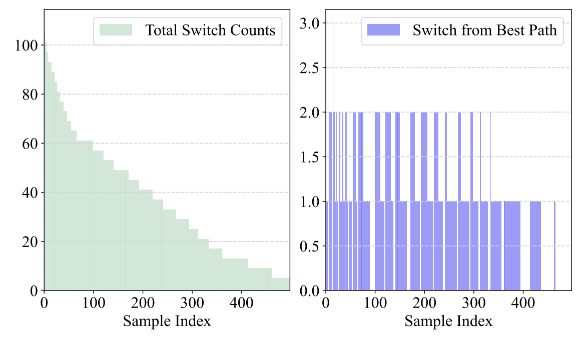


Figure 2: Statistics for total switch (left) and switch from the best path to the suboptimal (right).

3 Method Rationale

In this section, we present empirical findings that highlight the key challenges of tree search in LLM and provide the rationale behind our method. First, the difficulty in parallelism hinders the fully utilization of computational resources, preventing efficient deep reasoning (Sec. 3.1). Second, frequent switch between paths and excessive exploitation of low-confidence paths results in shallow thinking and redundant rollouts, wasting effort on fewer possible candidates (Sec. 3.2).

3.1 Difficulty in Parallelism

Tree search is the key for deep LLM reasoning, but its sequential nature of tree structures presents challenges for efficient GPU parallelism. It inherently exhibits retrospective and recursive behaviors, causing inefficient execution when different paths vary in depth and termination points. Even if each node is constrained to generate the same number of tokens, the focus switching between different reasoning trajectories and the diverse path lengths makes it incompatible with the end-to-end parallelism on GPUs. The detailed illustrations for this phenomenon can be found in Appendix A.2.

3.2 Frequent Switching & Redundant Exploration

The focus switching between paths also makes the tree search fail in focused reasoning trajectory (Wang et al., 2025), which prevents deep thinking and leads to a tendency of shallow exploitation. We quantify the switch times of the reasoning focus on each sample in the Math500 dataset. Figure 2 counts the total switch, which is about 36.7 on average. As well as the switch from the best path to a suboptimal or incorrect one, which is up to 3 times for a single sample. It demonstrates the instability of the tree search algorithm in maintaining a

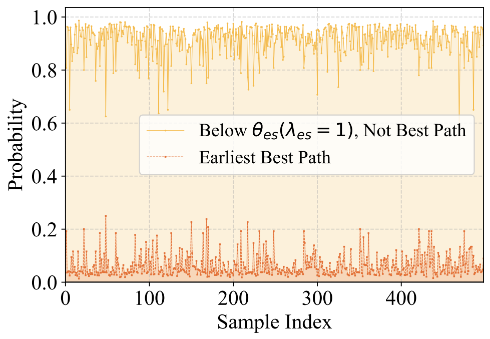


Figure 3: Probabilities with reordered samples of those have prior confidence below $\theta_{es}(\lambda = 1)$ in Eq. (2) and do not terminate with the highest reward score (yellow), and paths that are not the earliest best path (orange), which means there is already at least one path that has terminated with the same reward score.

focused reasoning trajectory.

The lack of early termination in existing tree search algorithms leads to excessive exploitation and redundant searching. Observations in Figure 3 show that low-confidence nodes rarely contribute to the best solutions, either terminated with suboptimal results (yellow) or failing to be the first to reach the best path (orange). The average probability of the suboptimal results brought by low confidence is 91.3%, while the probability of those nodes being the earliest best path is only 6.2%. It suggests that low-confidence nodes have little potential to reach the best solution, it is even hard to be the first one. It means that most low-confidence nodes have less contribution to the final results but waste computational resources.

These findings emphasize the importance of maintaining the focus on deep reasoning and pruning low-confidence paths for efficient inference.

4 Proposed Method

To address the aforementioned challenges, we propose an innovative framework that allows for efficient reasoning, termed Dynamic Parallel Tree Search (DPTS). In the generation phase, the Parallelism Streamline in Sec. 4.1 supports fine-grained and flexible paralleled expansion for arbitrary paths. In the selection phase, the Search and Transition Mechanism in Sec. 4.2 enables less redundant exploration by identifying the highly potential solutions to focus reasoning.

4.1 Parallelism Streamline

As illustrated in Figure 4, We fully parallelize the tree search process in our framework with three main components: **Tree Structure Building**, **KV**

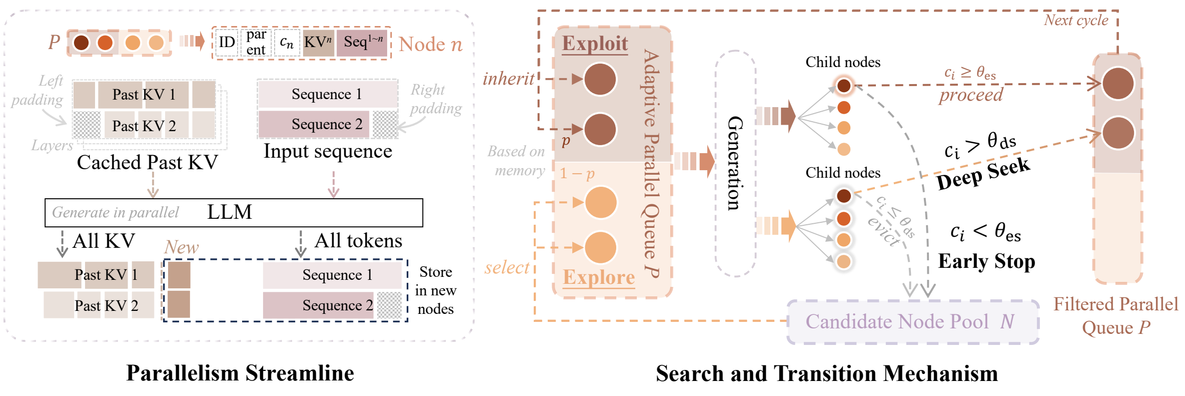


Figure 4: Overview of the proposed DPTS framework. The left part demonstrates the Parallelism Streamline, while the right part illustrate the proposed Search and Transition Mechanism.

Cache Handling, and **Adaptive Parallel Generation**. Each component is designed to optimize memory usage and parallel execution during the reasoning process.

4.1.1 Tree Structure Building

The tree search framework relies on a tree structure where each node represents a reasoning state. Specifically, the node data structure includes the following elements:

- **Node ID**: A unique identifier for each node.
- **Parent Node**: A reference to the parent node, establishing the hierarchical structure of the tree.
- **Prior Confidence**: The confidence of the node, based on prior knowledge and model predictions.
- **Key-Value Cache** (KV^n): The key-value cache specific to this node, storing intermediate results during the reasoning process.
- **Token Sequence** ($Seq^{1\sim n}$): The complete token sequence from the root node to the current node, representing the reasoning path taken so far.

The key challenge lies in managing the KV cache (Floridi and Chiriatti, 2020). Instead of storing the entire sequences, each node only retains its own KV cache. This significantly reduces memory usage, particularly when dealing with a large number of nodes in the tree. By keeping each node's cache isolated, we avoid redundant memory usage while ensuring that each node has necessary information to continue reasoning process.

4.1.2 KV Cache Handling

The KV cache for each node is stored separately, and during parallel execution, these caches need to be collected and concatenated for efficient parallelism. The key challenge is that tree search paths have varying lengths, which means that both the KV caches and the input sequences for different nodes will vary in size and be hard to parallel.

To address this, we use a simple but straightforward padding technique to ensure that all sequences have consistent lengths before being processed. Specifically, for nodes with shorter KV caches, we apply left padding with zeros. Similarly, input sequences are padded with a predefined padding token to match the longest sequence in the batch. This padding ensures that all nodes are processed in parallel with consistent sequence lengths and corresponding KV cache, allowing for efficient batch processing across the tree search. The details of padding and concatenating are given in Appendix Eq. (3) and (4).

Besides data collecting and preparation, we also clean up the useless KV cache either the leaf node is terminated, or all the children's branches are exploited and finished. In this way, we release the memory, making room for new reasoning paths.

4.1.3 Adaptive Parallel Generation

To further utilize the computational resources, we introduce an adaptive parallelism queue, which dynamically adjusts the number of parallel paths based on the available GPU memory. The parallelism queue size, denoted $|P|$, is used to restrict the number of exploitation and exploration paths in Sec. 4.2.1. It is calculated by the available and the peak memory usage during previous generations:

$$|P| = \frac{O_{\text{max}} - O_{\text{init}}}{O_{\text{peak}} - O_{\text{init}}}, \tag{1}$$

where O_{max} is the total memory budget, O_{peak} represents the peak memory usage from the previous generation, and O_{init} is the memory consumption during model initialization. As the tree grows, the memory occupation of intermediate results continues to increase even with KV cache cleaning. Since memory overflow is one of the termination conditions, it is important to adaptively adjust the parallel number, preventing excessive memory allocation and early termination.

After the generation phase, the newly generated sequences and KV caches are stored based on the tree width. The sequences for each node are completely stored, while the KV caches are stored partially with only the new tokens generated at this step (details are provided in Appendix (5)). The new nodes are then added to the candidate node pool N, where they will be available for subsequent selection processes in the tree search.

In summary, our Parallelism Streamline is a well-structured streamline to optimize both memory usage and parallel execution. The overall process is showcased in Algorithm 1. More details can be found in Appendix B.1 due to the limited length.

Algorithm 1 Algorithmic process DPTS

```
Input: LLM generation function $llm(x)$, PRM reward function $prm(x)$, Query $q$, Candidate Node Pool $N = \varnothing$, Parallel Queue $P = \varnothing$, Exploit Node Proportion $p$, Tree Width $w$.
Output: End node with best path reward $n^*$.
    // Step 1: Initialize the root node
 1: $r \leftarrow$ generate_node$(q, \text{None})$
 2: $N \leftarrow N \cup \{r\}$
 3: while within computational budget do
 4:     $P_{\text{size}} \leftarrow$ Eq. (1)
        // Step 2: Perform searching
 5:     $P \leftarrow \text{Search}(P, P_{\text{size}}, N)$ (Algorithm 2)
 6:     $\theta_{\text{es}}, \theta_{\text{ds}} \leftarrow$ Eq. (2)
        // Step 3: Parallelism by Eq. (3) and (4)
 7:     $\mathbf{n} \leftarrow$ generate_node$(\text{Seq}^{all}, \text{KV}^{all})$
        // Step 4: Update new nodes by Eq. (5)
        // Step 5: Terminate and reward
 8:     $N \leftarrow \text{Reward}(P, N)$ (Algorithm 4)
        // Step 6: Perform transition
 9:     $P \leftarrow \text{Transition}(P, \theta_{\text{es}}, \theta_{\text{ds}})$ (Algorithm 3)
10: end while
11: return $\max_{\text{reward}}(\forall n \in N)$
```

4.2 Search and Transition Mechanism

In this section, we introduce the **Search** and **Transition** Mechanism in DPTS, which is a hybrid search algorithm that balances exploitation and exploration through separate management and dynamic conversion.

4.2.1 Search

The Search Mechanism aims to balance exploration and exploitation by dynamically partitioning the nodes in parallel queue P into two categories: *explore nodes* and *exploit nodes*.

As illustrated in Figure 4 (right), these nodes are selected from the candidate node pool N. The partition ratio p can be manually adjusted according to the task and memory budget. At initialization, the top $p|P|$ highest-scoring nodes are assigned as *exploitation nodes*, while the remaining $(1-p)|P|$ nodes are assigned as *exploration nodes*. While during searching progress, the proportion of the two types of nodes dynamically fluctuates based on the transition mechanism in Sec. 4.2.2. The primary distinction between these two nodes lies in their origins and roles during the search process.

Exploitation Nodes The exploitation nodes are primarily inherited from parent exploitation nodes, focusing on refining the most promising paths in the search space. When a child node's confidence exceeds a predefined threshold, it inherits the status of its parent exploitation node and continues that path. This inheritance ensures that the most promising paths are deepened and further refined. Additionally, when the number of exploitation nodes falls below a predefined threshold, new high-confidence candidate nodes from the pool N are selected to fill the gap, ensuring that the number of exploitation nodes remains adequate for the search process. This strategy enables the exploitation of high-potential paths while maintaining the focus on areas with high confidence.

Exploration Nodes In contrast to exploitation nodes, the exploration nodes are not inherited from previous nodes but are dynamically selected from the candidate nodes pool. These nodes are responsible for discovering new paths that may have high potential but low current confidence in the search space. At each reasoning step, the exploration nodes are reselected from the candidate pool N, choosing the highest-confidence nodes that are not already assigned as exploitation nodes. The dynamic re-selection of exploration nodes allows the search process to adapt to changing circumstances and uncover new regions of the search space that may lead to better solutions.

4.2.2 Transition

While the Search Mechanism ensures a balance between exploration and exploitation, the redundant issue is not entirely mitigated. One example is that the initial exploitation nodes are not guaranteed to be the optimal solution. However, they only stop exploiting till reach the termination condition. Another issue occurs when high-confidence nodes are assigned as exploration nodes, but they will wait for the computation resources and do not roll out till the previous paths terminate.

To address these issues, we introduce the Transition Mechanism, which consists of two main strategies: *Early Stop* (Exploitation $\rightarrow$ Exploration) and *Deep Seek* (Exploration $\rightarrow$ Exploitation). As illustrated in Figure 4 (middle), these strategies allow an evolving search space with node transits between the two statuses. It helps the tree maintain focused reasoning, ensuring the efficient allocation and utilization of limited computational resources throughout the whole search process.

Early Stop (Exploitation $\rightarrow$ Exploration) The Early Stop (Yao et al., 2007) strategy allows relatively low-confidence exploitation nodes to transition into explore nodes, eliminating redundant exploitation on suboptimal paths. During the expansion process, if the best child node of an explore node has a confidence lower than a certain threshold θ_{es}, the child node will be excluded from the queue P in the next cycle. This prevents further exploration of paths that are unlikely to lead to optimal solutions, saving computational resources. Conversely, if the child node's confidence exceeds θ_{es}, it inherits the parent's status and continues to explore in the next cycle. This mechanism ensures that only the most promising explore nodes continue to expand, optimizing both exploration and resource usage. The threshold θ_{es} is defined as follows:

$$\theta_{\text{es}} = \begin{cases} \lambda_{\text{es}} \frac{1}{|\mathcal{N}|} \sum\limits_{i \in \mathcal{N}} c_i, & \text{if } t \leq t^* \\ \max\limits_{i \in \mathcal{N}} c_i, & \text{otherwise} \end{cases} \tag{2}$$

where $\mathcal{N}$ is the set of previously expanded nodes, c_i represents the confidence of node i, λ_{es} is a coefficient that adjusts the threshold, t is the number of currently terminated paths, and t^* is a predefined threshold after which θ_{es} is adjusted.

Deep Seek (Exploration $\rightarrow$ Exploitation) The Deep Seek strategy addresses the issue of inefficient over-exploration and shallow thinking, ensuring promising exploration nodes can be dug deeper. Specifically, exploration nodes with confidence exceeding a threshold θ_{ds} with λ_{ds} are promoted to exploitation nodes. As a result, the number of exploitation nodes may temporarily exceed $p|P|$. But as more high-confidence nodes are promoted, θ_{es} increases, and thus more exploration nodes are stopped under the Early Stop strategy. This creates a dynamic balance between exploration and exploitation throughout the search process.

In a word, the proposed Search and Transition Mechanism in DPTS effectively manages the tradeoff between exploitation and exploration with dynamic and bidirectional transition. Detailed algorithms in this part can be found in Appendix B.1.

5 Experiments

We conduct extensive experiments to evaluate the efficiency of the DPTS framework. We benchmark its performance against various search algorithms across multiple models and datasets to ensure a comprehensive analysis.

5.1 Settings

Models. We include Qwen2.5-Instruct models with 1.5B, 7B, 14B and 72B variants (Yang et al., 2024), and LLaMA-3.2-Instruct models with 3B and 8B variants (Touvron et al., 2023) to cover various model sizes.

Datasets. We conduct our experiments on both math and code tasks. The evaluation on math datasets include Math500 (Hendrycks et al., 2021) and GSM8K (Cobbe et al., 2021), which are widely used for mathematical problem-solving. And the coding capability is tested on LiveCodeBench-v1.0 (Naman Jain, 2024), which is also a commonly used code generation benchmark.

Comparison Methods. We compare DPTS against three widely used search algorithms: Monte Carlo Tree Search (MCTS) (Sprueill et al., 2023) , Best-of-N (Cobbe et al., 2021) , Beam Search (Yao et al., 2023). Since efficient tree search algorithms have recently regained attention after the emergence of LLM reasoning, the strong baselines are limited. As a result, we primarily compare DPTS against these typical and well-established search algorithms to demonstrate the effectiveness of our method. An introduction of the comparison methods and other details about experimental settings are provided in Appendix C.

5.2 Comparisons on Search Algorithms

We conduct a comprehensive comparison across different search algorithms on various models and

Table 1: Comparisons across existing search algorithms on LLM reasoning tasks under math datasets.

Model	Algo.	Math500		GSM8K	
		Acc.	Time (s)	Acc.	Time (s)
Qwen-2.5 1.5B	MCTS	56.6	117.37	75.1	73.28
	Best-of-N	52.6	89.87	70.1	33.37
	Beam	52.4	104.58	71.5	41.27
	DPTS	59.2	**45.10**	75.2	**18.32**
Qwen-2.5 7B	MCTS	75.2	121.46	89.6	79.68
	Best-of-N	71.6	91.29	88.2	34.89
	Beam	72.4	106.89	86.7	36.49
	DPTS	76.2	**53.50**	89.4	**19.95**
Llama-3 3B	MCTS	48.6	111.80	64.0	57.19
	Best-of-N	46.4	91.34	57.1	27.27
	Beam	45.2	104.36	58.4	28.27
	DPTS	50.8	**47.75**	67.8	**27.74**
Llama-3 8B	MCTS	54.2	143.36	69.5	69.74
	Best-of-N	49.8	122.63	67.6	33.48
	Beam	49.6	142.21	68.3	34.51
	DPTS	55.4	**37.98**	68.2	**17.82**

Table 2: Performance comparison on the code generation task under LiveCodeBench-v1.0 (easy subset).

Method	Pass (%)	Pass@1 (%)	Avg. Time (s)
MCTS	43.7	28.8	230.4
Best-of-N	40.3	27.1	148.2
Beam Search	41.5	27.8	165.7
DPTS	41.5	**29.7**	**127.2**

Table 3: Comparison of accuracy and average time on larger models. 250 seconds limit for Qwen2.5-72B and 150 seconds limit for Qwen2.5-14B per sample.

Model	Method	Acc.	Time (s)
Qwen-2.5 14B	MCTS	76.0	135.0
	DPTS	78.4	**81.8**
Qwen-2.5 72B	MCTS	67.0	248.2
	DPTS	74.7	**200.8**

sizes. We emphasize the search efficiency of our method while maintaining accuracy.

Efficiency Comparisons. Results in Table 1 show that DPTS significantly reduces inference time compared to other search methods across various models and datasets, demonstrating superior efficiency. On Math500, DPTS achieves the lowest inference time across all models. Particularly, in Qwen-2.5, DPTS reduces the search time from 117.37s (MCTS) to 45.10s in the 1.5B model, achieving nearly a 2.6× speedup, and reduces from 121.46s (MCTS) to 53.50s in 7B model, accelerating nearly 2.2×. The impact is even more pronounced on the GSM8K, where DPTS achieves a 3.9× speedup from 79.68s to 19.95s in Qwen-2.5-7B. And DPTS even only requires 17.82s for each sample using Llama-3-8B, also 3.9×. It forcefully suggests that DPTS effectively mitigates redundant rollouts and optimizes search efficiency. We highlight that, especially on the more challenging tasks, the Early Stop plays a crucial role. Without it, trees often run till timeout, significantly increasing inference time. Our approach allows the search tree to terminate earlier within a limited number of expansions, effectively reducing computation time.

Accuracy Comparisons. DPTS demonstrates strong and generalizable performance across both mathematical reasoning and code generation tasks. In math tasks, DPTS maintains the searching quality and even outperforms the existing algorithms with half or even less of the reasoning time. On the Math500 dataset, DPTS achieves the highest accuracy in all experiment cases, surpassing MCTS, Best-of-N, and Beam search. Notably, for Qwen-2.5-1.5B, DPTS improves accuracy from 56.6% (MCTS) to 59.2%. A similar trend is observed on GSM8K, where DPTS either matches or slightly improves accuracy over MCTS, and surpasses Best-of-N and Beam Search by a wide margin. On Llama-3-3B, DPTS has 67.8% accuracy, outperforming the previous best MCTS by 3.8% with only 48.5% time consumption.

To estimate the generalizability of DPTS, we conduct experiments on a code generation dataset, LiveCodeBench-v1.0 benchmark, using its "easy" subset. We utilize Qwen2.5-coder-7B-Instruct as the generation model and skywork-reward-llama-3.1-8B as the reward model. As reported in Table 2, DPTS achieves a higher Pass@1 score compared to standard MCTS with significantly less time (14.2%↓), indicating an improved ability to identify optimal solutions within the time constraints. These results highlight that DPTS offers robust quality and efficiency for complex reasoning and generation tasks across domains.

Generalization on Larger Models. To further evaluate the scalability of our method under larger language models, we conduct experiments using Qwen2.5-72B-Instruct and Qwen2.5-14B-Instruct as generation models on the Math500 dataset. Given the increased computational cost associated with these models, we extend the per-sample inference time limits to 250 seconds for the 72B model and 150 seconds for the 14B model. The results are shown in Table 3. We observe that standard MCTS often runs until the time or memory limit due to the absence of early stopping mechanisms. This behavior is particularly severe for

Table 4: Performance with Qwen2.5-Math-PRM-7B as reward model tested on Math500. DPTS consistently improves both accuracy and efficiency over MCTS.

Model	Method	Acc.	Time (s)
Qwen-2.5 14B	MCTS	75.0	133.9
	DPTS	77.7	**86.1**
Llama-3 8B	MCTS	52.8	101.6
	DPTS	56.4	**52.8**

larger models. For instance, the 250-second limit appears insufficient for the 72B model to complete a meaningful search using MCTS, leading to sub-optimal accuracy. In contrast, our DPTS achieves higher accuracy (74.7 vs. 67.0 on 72B and 78.4 vs. 76.0 on 14B) within the same time constraints, due to its capability of focusing the search on promising trajectories. Notably, DPTS often terminates the search early, contributing to lower average inference time. These demonstrate that DPTS is also scalable when applied to larger models.

Robustness to PRM Configuration. We examine the impact of using different Process Reward Models (PRMs) on the performance of our DPTS algorithm. Without specified, our default PRM is the math-shepherd-mistral-7b-prm in (Wang et al., 2023). We adopt the default configuration specified in official repository[1]. Detailed introduction can be found in Appendix C.2. To further assess the robustness of our approach, we also replace the PRM with Qwen2.5-Math-PRM-7B and evaluate on Math500 dataset with two generation models: Qwen2.5-14B-Instruct and LLaMA3.2-8B-Instruct. Results are summarized in Table 4. It shows that our method consistently achieves better accuracy and efficiency across different PRMs and generation models. These results highlight the robustness and generalizability of DPTS under varying reward estimation strategies.

5.3 Ablation Study

To analyze the contribution of each component within the DPTS framework, we conduct an ablation study on Qwen2.5-1.5B-Instruct with Math500 in Table 5. In this study, we use the classical MCTS as the baseline and incrementally integrate our proposed techniques to evaluate their impact.

We begin with the original MCTS (non-parallel, $|P| = 1$) as the baseline. It spends the most time per sample and has the largest best index 10.45. We

[1]https://huggingface.co/peiyi9979/math-shepherd-mistral-7b-prm/blob/main/config.json

Table 5: Ablation study of each component in DPTS framework. "AP": adaptive parallelism. "S": Searching. "T": Transition. "Best Index": The average index of terminated path leads to the best solution.

Algo.	$\|P\|$	**Acc.**	**Time (s)**	**Best Index**
Baseline	1	56.6	117.37	10.45
Baseline	AP	58.8	108.06	8.27
+ S	AP	58.2	76.81	4.66
+ T	AP	57.0	32.22	2.51
+ S + T	AP	59.2	45.10	4.17

then apply Parallelism Streamline with Adaptive Parallel Generation (AP), and accuracy improves. It shows that the trees are able to grow faster and larger to include a better solution with parallelism.

Next, we assign the node status as exploit or explore nodes for each expansion with Search Mechanism, denoted as "+S | AP" in Table 5. The search process becomes significantly more structured and targeted, leading to a boost in efficiency. The time of each sample saves by 31.24s (28.9%$\downarrow$). It finds the best path within an average of 4.66 terminated paths, much fewer than Baseline AP.

Moreover, when only applying the Early Stop strategy in the Transition Mechanism (denoted as "+T | AP), we obtain fast inference with much less time and paths. However, since we only use the exploitation nodes without exploring the possible branches, the accuracy is relatively low. Therefore, we claim that the Search and Transition Mechanism should be used as a whole: the Search mechanism provides different node statuses for exploitation and exploration, while the Transition mechanism makes them flexibly change and update.

Finally, we combine our Search and Transition Mechanism (denoted as "+S+T | AP"), enabling Early Stop and Deep Seek. It shows the best search results in accuracy and efficiency. It demonstrates that DPTS is efficient in quickly identifying optimal solutions and conducting deep reasoning.

Results show that each component of DPTS contributes significantly to improving inference speed and reasoning accuracy, making it a robust and scalable framework for parallel tree search.

5.4 Hyperparameter Analysis

We conduct a hyperparameter study in Table 6 on the thresholds θ_{es} and θ_{ds} in the Transition mechanism. When $t < t^*$, the threshold θ follows the mean-based strategy determined by λ. When $t \geq t^*$, it turns to a max-based one. Empirically, we

Table 6: Hyperparameter λ_{es} and λ_{ds} in transition thresholds θ_{es} and θ_{ds}. "ES (Early Stop) %" and "DS (Deep Seek) %" are the ratios of the node type transition between the exploitation and exploration. More results can be found in Appendix C.5.

λ_{es}	λ_{ds}	**Acc.**	**Time (s)**	**ES (%)**	**DS (%)**
1.0	1.0	53.0	47.59	41.4	10.5
0.9	0.9	58.6	43.30	15.6	20.9
0.8	0.8	58.0	46.33	8.1	23.9
0.6	0.6	57.4	44.39	6.1	32.3
0.4	0.4	56.6	38.41	0	0.1

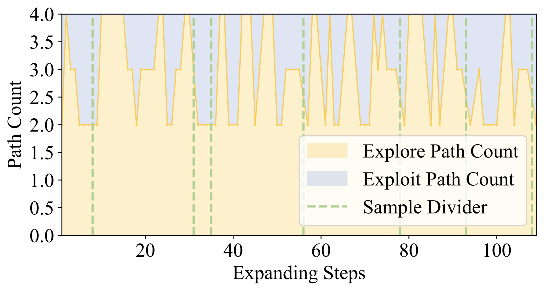


Figure 5: Proportions of exploit and explore nodes throughout the search process.

set $t^* = 5$ to balance the flexibility and efficiency.

Experimental results demonstrate that our method is robust to λ. We highlight that λ_{es} and λ_{ds} can be set differently based on the specific task. But DPTS consistently works well around $\lambda \in [0.6, 0.9]$. It demonstrates that the Transition mechanism is effective in mitigating the redundancy issue during search progress. However, we notice that, if λ is large, the ES transition may be aggressive, which leads to unsatisfactory results (e.g. $\lambda = 1.0$). Meanwhile, if λ is too small, it degrades to all exploitation nodes, resulting in low efficiency as well. Additional results are shown in Appendix C.5, we observe that the method remains effective across a wide range of hyperparameter settings, i.e., settings within the range of $[0.6, 0.9]$ consistently yield improvements. Therefore, selecting a setting within this robust interval is relatively straightforward and reliable.

5.5 Visualizations

To provide an intuitive understanding of the effectiveness of our proposed method, we present visualizations of searching trajectories.

First, we analyze the dynamic changes in the number of exploitation and exploration nodes throughout the search process in Figure 5. The Deep Seek transition temporarily increases the proportion of exploit paths, allowing promising nodes to receive deeper reasoning. However, as the threshold θ_{es} increases, exploit nodes are more likely to reach the threshold and stop. As a result, the number of exploit nodes naturally decreases, reinforcing a balance between exploitation and exploration. This dynamic adaptation ensures that DPTS stretches on the most promising branches under the constraint of computational resources.

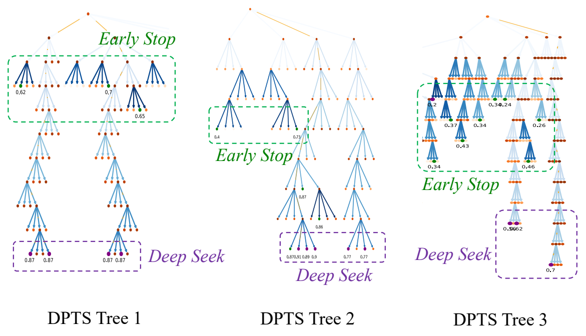


Figure 6: Visualization of DPTS Tree. The green boxes are early stopped nodes based on their prior confidence using our *Early Stop* mechanism, and the purple boxes are the terminated nodes with posterior reward scores.

Second, we show the trees generated by DPTS and analyze the search behavior in Figure 6. It does not continue exploitation on low-confidence nodes, effectively pruning unpromising branches after shallow exploration. Additionally, the trees are capable of stable reasoning focus, with deep exploitation on promising paths. Therefore, the generated trees exhibit a relatively narrow width, as DPTS primarily expands nodes that are more relevant to the optimal path and spend less time on unnecessary regions. It demonstrates that DPTS ensures high-potential paths receive deeper thinking within a limited time and memory budget.

6 Conclusion

We propose DPTS (Dynamic Parallel Tree Search) that effectively enhances the computational efficiency of LLM reasoning. DPTS introduces Parallelism Streamline, allowing efficient inference with arbitrary path lengths and nodes. The Dynamic Search and Transition Mechanism mitigates redundant rollouts and focuses on promising solutions. Experiments on math and code benchmarks show that DPTS achieves the highest reasoning accuracy while delivering 2-4× speedup. Our work offers valuable insights into optimizing computation for efficient LLM reasoning, strengthening problem-solving capabilities to tackle real-world challenges.

Limitations

Our DPTS framework focuses on selecting and refining the search paths, but does not involve hardware design. Therefore, it is orthogonal to low-level methods. For example, we can integrate DEFT (Yao et al., 2024b) to reduce the data transportation of the shared prefixes, leading to further acceleration. Also, our method is validated only on math reasoning tasks, and has not been tested on other domains, such as coding or scientific problems. However, we believe its generalizable capabilities make it applicable across a wide range of fields. Additionally, we envision that this method can also be applied to online training by improving generation quality. We leave these attempts to our future work.

Acknowledgments

This project is supported by the Beijing Municipal Science and Technology Project (No. Z231100010323002), and the National Natural Science Foundation of China (Nos. 62306025, 92367204), CCF-Baidu Open Fund, and the National Research Foundation, Singapore, under its NRF Professorship Award No. NRF-P2024-001.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Maciej Besta, Florim Memedi, Zhenyu Zhang, Robert Gerstenberger, Nils Blach, Piotr Nyczyk, Marcin Copik, Grzegorz Kwaśniewski, Jürgen Müller, Lukas Gianinazzi, et al. 2024. Topologies of reasoning: Demystifying chains, trees, and graphs of thoughts. *arXiv preprint arXiv:2401.14295*.

Zhenni Bi, Kai Han, Chuanjian Liu, Yehui Tang, and Yunhe Wang. 2024. Forest-of-thought: Scaling test-time compute for enhancing llm reasoning. *arXiv preprint arXiv:2412.09078*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43.

Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*.

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 4, pages 216–217.

Pengyu Cheng, Tianhao Hu, Han Xu, Zhisong Zhang, Yong Dai, Lei Han, nan du, and Xiaolong Li. 2024. Self-playing adversarial language game enhances llm reasoning. In *Advances in Neural Information Processing Systems*, volume 37, pages 126515–126543. Curran Associates, Inc.

Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. A survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Linger Deng, Yuliang Liu, Bohan Li, Dongliang Luo, Liang Wu, Chengquan Zhang, Pengyuan Lyu, Ziyang Zhang, Gang Zhang, Errui Ding, et al. 2024. R-cot: Reverse chain-of-thought problem generation for geometric reasoning in large multimodal models. *arXiv preprint arXiv:2410.17885*.

Luciano Floridi and Massimo Chiriatti. 2020. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. 2024. Break the sequential dependency of llm inference using lookahead decoding. *arXiv preprint arXiv:2402.02057*.

Tingxu Han, Chunrong Fang, Shiyu Zhao, Shiqing Ma, Zhenyu Chen, and Zhenting Wang. 2024. Token-budget-aware llm reasoning. *arXiv preprint arXiv:2412.18547*.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Yuhan Dong, Yu Wang, et al. 2024. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. 2024. V-star: Training verifiers for self-taught reasoners. *arXiv preprint arXiv:2402.06457*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Fangkai Jiao, Chengwei Qin, Zhengyuan Liu, Nancy F Chen, and Shafiq Joty. 2024. Learning planning-based reasoning by trajectories collection and process reward synthesizing. *arXiv preprint arXiv:2402.00658*.

Yongcheng Jing, Chongbin Yuan, Li Ju, Yiding Yang, Xinchao Wang, and Dacheng Tao. 2023. Deep graph reprogramming. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 24345–24354.

Levente Kocsis, Csaba Szepesvári, and Jan Willemson. 2006. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1:1–22.

Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286.

Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, et al. 2024a. Large language model inference acceleration: A comprehensive hardware perspective. *arXiv preprint arXiv:2410.04466*.

Yiwei Li, Peiwen Yuan, Shaoxiong Feng, Boyuan Pan, Xinglin Wang, Bin Sun, Heda Wang, and Kan Li. 2024b. Escape sky-high cost: Early-stopping self-consistency for multi-step reasoning. In *The Twelfth International Conference on Learning Representations*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *arXiv preprint arXiv:2305.20050*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100.

Shunyu Liu, Wenkai Fang, Zetian Hu, Junjie Zhang, Yang Zhou, Kongcheng Zhang, Rongcheng Tu, Ting-En Lin, Fei Huang, Mingli Song, and Dacheng Tao. 2025. A survey of direct preference optimization. *arXiv preprint arXiv:2503.11701*.

Alex Gu Wen-Ding Li Fanjia Yan Tianjun Zhang Sida Wang Armando Solar-Lezama Koushik Sen Ion Stoica Naman Jain, King Han. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*.

Sania Nayab, Giulio Rossolini, Marco Simoni, Andrea Saracino, Giorgio Buttazzo, Nicolamaria Manes, and Fabrizio Giacomelli. 2024. Concise thoughts: Impact of output length on llm reasoning and cost. *arXiv preprint arXiv:2407.19825*.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. 2024. Skeleton-of-thought: Prompting llms for efficient parallel generation. In *The Twelfth International Conference on Learning Representations*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc.

Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki van Stein, and Thomas Back. 2024. Reasoning with large language models, a survey. *arXiv preprint arXiv:2407.11511*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Henry W Sprueill, Carl Edwards, Mariefel V Olarte, Udishnu Sanyal, Heng Ji, and Sutanay Choudhury. 2023. Monte carlo thought search: Large language model querying for complex scientific reasoning in catalyst design. *arXiv preprint arXiv:2310.14420*.

Hanshi Sun, Momin Haider, Ruiqi Zhang, Huitao Yang, Jiahao Qiu, Ming Yin, Mengdi Wang, Peter Bartlett, and Andrea Zanette. 2024. Fast best-of-n decoding via speculative rejection. *arXiv preprint arXiv:2410.20290*.

Ganchao Tan, Daqing Liu, Meng Wang, and Zheng-Jun Zha. 2020. Learning to discretely compose reasoning module networks for video captioning. *arXiv preprint arXiv:2007.09049*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Guangya Wan, Yuqi Wu, Jie Chen, and Sheng Li. 2024a. Dynamic self-consistency: Leveraging reasoning paths for efficient llm sampling. *arXiv preprint arXiv:2408.17017*.

Ziyu Wan, Xidong Feng, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. 2024b. Alphazero-like tree-search can guide large language model decoding and training. In *Forty-first International Conference on Machine Learning*.

Chaojie Wang, Yanchen Deng, Zhiyi Lyu, Liang Zeng, Jujie He, Shuicheng Yan, and Bo An. 2024a. Q*: Improving multi-step reasoning for llms with deliberative planning. *arXiv preprint arXiv:2406.14283*.

Peiyi Wang, Lei Li, Zhihong Shao, RX Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. 2023. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. *arXiv preprint arXiv:2312.08935*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Yue Wang, Qiuzhi Liu, Jiahao Xu, Tian Liang, Xingyu Chen, Zhiwei He, Linfeng Song, Dian Yu, Juntao Li, Zhuosheng Zhang, et al. 2025. Thoughts are all over the place: On the underthinking of o1-like llms. *arXiv preprint arXiv:2501.18585*.

Zhenglin Wang, Jialong Wu, Yilong Lai, Congzhi Zhang, and Deyu Zhou. 2024b. Seed: Accelerating reasoning tree construction via scheduled speculative decoding. *arXiv preprint arXiv:2406.18200*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. 2024. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models. *arXiv preprint arXiv:2408.00724*.

Yuxi Xie, Anirudh Goyal, Wenyue Zheng, Min-Yen Kan, Timothy P Lillicrap, Kenji Kawaguchi, and Michael Shieh. 2024. Monte carlo tree search boosts reasoning via iterative preference learning. *arXiv preprint arXiv:2405.00451*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Huanjin Yao, Jiaxing Huang, Wenhao Wu, Jingyi Zhang, Yibo Wang, Shunyu Liu, Yingjie Wang, Yuxin Song, Haocheng Feng, Li Shen, and Dacheng Tao. 2024a. Mulberry: Empowering mllm with o1-like reasoning and reflection via collective monte carlo tree search. *arXiv preprint arXiv:2412.18319*.

Jinwei Yao, Kaiqi Chen, Kexun Zhang, Jiaxuan You, Binhang Yuan, Zeke Wang, and Tao Lin. 2024b. Deft: Flash tree-attention with io-awareness for efficient tree-search-based llm inference. *arXiv preprint arXiv:2404.00242*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 11809–11822. Curran Associates, Inc.

Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. 2007. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024a. Rest-mcts*: Llm self-training via process reward guided tree search. *arXiv preprint arXiv:2406.03816*.

Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. 2024b. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*.

Kongcheng Zhang, Qi Yao, Baisheng Lai, Jiaxing Huang, Wenkai Fang, Dacheng Tao, Mingli Song, and Shunyu Liu. 2025. Reasoning with reinforced functional token tuning. *arXiv preprint arXiv:2502.13389*.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Yu Zhao, Huifeng Yin, Bo Zeng, Hao Wang, Tianqi Shi, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. 2024. Marco-o1: Towards open reasoning models for open-ended solutions. *arXiv preprint arXiv:2411.14405*.

Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294*.

Appendix

Contents

A More Observations of Motivation 13

A.1 Wasted Tokens and Expansions 13

A.2 Examples of DFS and BFS Trees 14

B Formulas and Algorithms 15

B.1 Formulas in Parallelism Streamline ... 15

B.2 Algorithms for Searching and Transition Mechanism 15

C Additional Details about Experiment 16

C.1 Comparison Methods 16

C.2 Experimental Settings 16

C.3 Distribution of Best Path Index 17

C.4 Alleviating Frequent Switching 17

C.5 Additional Results of λ 17

C.6 Stopping Cause Analysis 18

D Related Work 18

E Future Work Discussion 19

A More Observations of Motivation

A.1 Wasted Tokens and Expansions

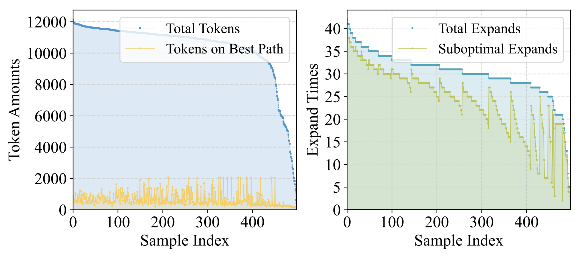


Figure 7: The proportion of tokens required for the best path relative to the total tokens generated (left), and the proportion of expansions on suboptimal paths relative to the total number of expansions (right).

To better understand the inefficiencies caused by frequent node switching, we conduct a statistical analysis on Qwen-2.5-1.5B with the Math500 dataset and evaluate the redundancy in token generation and node expansion.

Token redundancy analysis: In Figure 7(left), we compare the total number of tokens generated for each sample (blue line) against the number of tokens required for the best path (yellow line). The samples are sorted in descending order primarily by total token count and secondarily by best-path token count. Our analysis shows that the total token count does not exhibit a strict multiplicative relationship with the best-path token count, but in general, the number of tokens required for the best path is significantly lower than the total token count. This suggests that traditional tree search algorithms generate a large number of unnecessary tokens during exploration.

Expansion redundancy analysis: We also examined the number of node expansions during tree growth (Figure 7(right)). The blue line represents the total number of expansions for each sample, while the green line represents the number of expansions on suboptimal paths (i.e., nodes that do not contain any part of the optimal solution). While there is no strict multiplicative correlation between these two metrics, the green line closely follows the blue line, indicating that a significant proportion of expansions occur on suboptimal paths. This further supports the observation that traditional tree search algorithms frequently explore unnecessary areas before finding the best solution.

A.2 Examples of DFS and BFS Trees

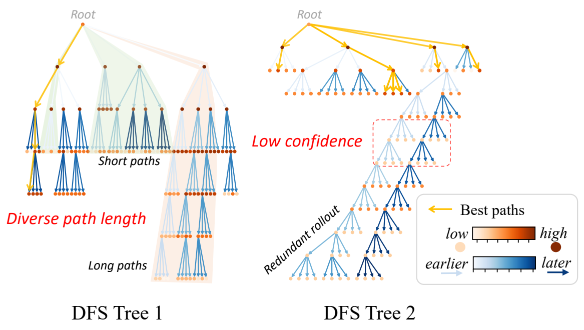


Figure 8: Growth of two trees with DFS algorithm.

As illustrated in Figure 8, which shows two typical depth-first search (DFS) trees, we visualize the node expansion process in layers based on tree depth. The darker reddish-brown nodes represent high-confidence nodes, while the lighter nodes indicate lower-confidence ones. The arrows denote parent-child relationships, where dark blue arrows indicate later-generated nodes and light blue arrows represent earlier-generated nodes.

From the figure, we can clearly observe the reasoning trajectory of tree search: starting from the root node, the search prioritizes the child node with the highest confidence, then recursively expands deeper by selecting the most promising child node at each level. This continues until a termination condition is met, at which point the search backtracks and explores alternative paths from the root node. Due to the nature of this process, different paths vary significantly in their depth and termination points. Moreover, the next explored path does not follow a strict spatial or hierarchical pattern within the tree.

We also observe redundant exploration issues in the right two branches. At tree depths 4/5, the confidence scores of the expanded nodes are noticeably lower compared to previously explored nodes. However, due to the inherent mechanics of depth-first search (DFS), the algorithm continues expanding these nodes until the termination condition is met, even if the intermediate confidence scores remain consistently low. As a result, considerable computation is wasted on redundant expansions and token generations, with little contribution to improving the final output quality.

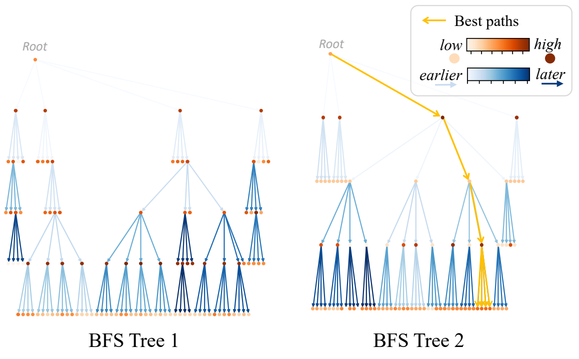


Figure 9: Growth of two trees with BFS algorithm.

As illustrated in Figure 9, BFS results in a flatter, more uniform, top-down expansion structure compared to the trees observed in Figure 8. This behavior creates two key inefficiencies: (1) Incomplete reasoning before termination: In our experiments on the Math500 dataset, a pure BFS approach resulted in 178 (about 35.6%) of reasoning paths terminating without generating an answer (e.g., Tree 1 in Figure 9). The algorithm explores many different areas of the tree but often fails to pursue any one path deeply enough to reach a valid conclusion. (2) Excessive expansions and token redundancy: Even when BFS eventually finds a correct answer, it tends to consume significantly more expansions and tokens than necessary (e.g., Tree 2 in Figure 9). The best path (highlighted in yellow arrows) has a depth of only 4, yet before discovering this optimal solution, BFS explores a large number of additional nodes (light blue arrows), many of which do not contain any part of the optimal path.

B Formulas and Algorithms

B.1 Formulas in Parallelism Streamline

Data Collection and Preparation. Before executing parallel inference, we should collect the input data that is stored in separate memory locations. Based on the Node Data Structure (refer to Appendix B.1), which is $[\text{id}, \text{parent}, \text{conf.}, \text{KV}^n, \text{Seq}^{1\sim n}]$, we need to concatenate the past KV caches and input sequences of different nodes into single large batch matrices. However, as discussed in Sec. 3, tree search paths exhibit varying path lengths, meaning that both past KV caches and context sequences have different sizes. To handle the length disparity and support arbitrary node parallelism, we apply padding for shorter past KV and context sequence:

$$\begin{aligned}
&\text{KV}^{1\sim n} = \texttt{concat}\left(\text{KV}^{r^n}, \cdots, \text{KV}^{a_1^n}, \text{KV}^n\right),\\
&\text{padding}^n = \mathbf{0}_{\max\left(\forall_{m\in P}|\text{KV}^{1\sim m}|\right)-|\text{KV}^{1\sim n}|},\\
&\text{KV}^{1\sim n}_{\text{pad}} = \texttt{concat}\left(\text{padding}^n, \text{KV}^{1\sim n}\right),\\
&\text{KV}^{all} = \texttt{stack}\left([\forall_{n\in P}\, \text{KV}^{1\sim n}_{\text{pad}}]\right),
\end{aligned} \tag{3}$$

where r and $a_1, a_2, \ldots$ are the root node and $1^{st}, 2^{nd}, \ldots$ ancestors of n, respectively. $\mathbf{0}_{len}$ is a matrix of zeros with length len. Similar to above, the input sequences are also padded and stacked:

$$\begin{aligned}
&\text{padding}^n = \textit{padding\_token}_{\max(\forall_{n\in P}|\text{Seq}^{1\sim n}|)},\\
&\text{Seq}^{1\sim \text{n}}_{\text{pad}} = \texttt{concat}\left(\text{Seq}^{1\sim \text{n}}, \text{padding}^n\right),\\
&\text{Seq}^{all} = \texttt{stack}\left(\forall_{n\in P}\text{Seq}^{1\sim \text{n}}_{\text{pad}}\right),
\end{aligned} \tag{4}$$

where $\textit{padding\_token}_{len}$ is a vector of the predefined padding token id with length len. In this way, the data is fed into the LLM for parallel generation. Additionally, techniques like DEFT (Yao et al., 2024b) are orthogonal to our DPTS and can be integrated to identify and merge shared prefixes, further optimizing inference efficiency.

Generation and Updating. After the generation phase, we obtain new output sequences and past KV caches. These are then partitioned based on the tree width. Specifically, we segment past KV caches and only store those corresponding to new tokens. Output sequences are completely stored rather than fragmented, due to the negligible memory overhead. For clearer demonstration, the output of i^{th} node in P can be written as

$$\begin{aligned}
&\mathbf{KV}^{n^{ij}} = \mathbf{n}.\texttt{past\_kv}_{[ij,\ldots,|\mathbf{KV}^{\text{all}}|:]}, \forall j \in [1, \ldots, w]\\
&\mathbf{Seq}^{1\sim n^{ij}} = \mathbf{n}.\texttt{output}_{[ij]}\\
&n^{ij} = \left[\text{id}, n^i, \mathbf{KV}^{n^{ij}}, \mathbf{Seq}^{1\sim n^{ij}}\right]\\
&\mathbf{n}^{\mathbf{new}} = \left\{n^{i1}, \ldots, n^{iw}\right\}
\end{aligned} \tag{5}$$

where w is the tree width, $\mathbf{n}$ is the generation output in parallel manner. If sequences were stored in a fragmented manner, every inference step would require additional collection and concatenation, introducing unnecessary latency. This approach is a trade-off between inference speed and memory consumption. Newly generated nodes are then updated into the candidate node pool N, making them available for subsequent selection processes.

B.2 Algorithms for Searching and Transition Mechanism

In the main text, due to paper length constraints, we only present the overall process in Algorithm 1, which connects the algorithms and formulations within the framework, including Parallelism Streamline and the Search and Transition Mechanism.

In the above section, we provided the mathematical formulation of the Parallelism Streamline. In the following, we further supplement the algorithmic details of Search, Transition, and Reward, offering readers a clear and intuitive representation of the algorithmic process.

Firstly, Algorithm 2 demonstrates the searching mechanism. Specifically, during initialization or when the number of nodes in the parallel queue P is less than the maximum parallelism τ_P, the algorithm selects the $\tau_P - |P|$ highest-confidence nodes from the candidate node pool N as supplementary nodes (Lines 8-9).

Then, the highest-confidence nodes are designated as exploit nodes (Line 12) until the proportion of exploit nodes reaches the ratio p. The remaining selected nodes are assigned as explore nodes. Finally, these newly selected nodes are merged into P to prepare for the next cycle of parallel expansion.

Next is the Transition Algorithm 3. After each expansion, we iterate through all nodes in the parallel queue P and identify the best child node n^* for each node.

Algorithm 2 Searching

```
Input: Parallel queue $P$, current parallel queue size $\tau_P$, candidate node pool $N$.
Output: Updated $P$.
 1: $e_1 \leftarrow 0$
 2: for all $n \in P$ do
 3:     if $n$.mode= EXPLOIT then
 4:         $e_1 \leftarrow e_1 + 1$
 5:     end if
 6: end for
 7: if $|P| < \tau_P$ then
 8:     $N' \leftarrow$ Descending $N$ based on conf.
 9:     $\mathbf{u} \leftarrow N'[: \tau_P - |P|]$
10:     for all $u \in \mathbf{u}$ do
11:         if $e_1 < p|P|$ then
12:             $u$.mode $\leftarrow$ EXPLOIT
13:             $e_1 \leftarrow e_1 + 1$
14:         else
15:             $u$.mode $\leftarrow$ EXPLORE
16:         end if
17:     end for
18:     $P \leftarrow P \cup \mathbf{u}$
19: end if
20: return $P$
```

Algorithm 3 Transition

```
Input: Parallel queue $P$, transition thresholds $\theta_{\text{es}}$ and $\theta_{\text{ds}}$.
Output: Updated $P$.
 1: for all $n \in P$ do
 2:     $n^* = \max_{\text{conf.}}(n.\text{children})$
 3:     if $n$.mode = EXPLOIT and $c_{n^*} > \theta_{\text{es}}$ or $n$.mode = EXPLORE and $c_{n^*} > \theta_{\text{ds}}$ then
 4:         $P \leftarrow P \cup \{n^*\}$
 5:         $n^*$.mode $\leftarrow$ EXPLOIT
 6:     end if
 7:     $P \leftarrow P \setminus n$
 8: end for
 9: return $P$
```

Algorithm 4 Reward

```
Input: Parallel queue $P$, candidate node pool $N$.
Output: Updated $N$.
 1: for all $n \in P$ do
 2:     $n$.children $\leftarrow$ Eq. (5)
 3:     for all $m \in n$.children do
 4:         if is_terminate($m$) then
 5:             $m$.reward $\leftarrow$ reward($m$)
 6:         else
 7:             $N \leftarrow N \cup \{m\}$
 8:         end if
 9:     end for
10: end for
11: return $N$
```

Then, based on the category of node n, we compare n^* with the corresponding threshold θ. If the early stop condition is not met or the deep seek condition is met, we add n^* as a new exploit node into P. Otherwise, the node will no longer be expanded and will be evicted from P.

After completing an expansion, we check whether each node's path meets the termination conditions, such as reaching the maximum token limit or generating an end token. If a path satisfies the termination condition, exploration of that path stops, and a reward is computed as the final path score. We demonstrate this process in Algorithm 4, which is identical to previous tree search algorithms and is not discussed in detail. However, for the sake of algorithmic completeness, we explicitly include it here.

C Additional Details about Experiment

C.1 Comparison Methods

We compare DPTS against three widely used search algorithms: (1) Monte Carlo Tree Search (MCTS) (Sprueill et al., 2023) balances exploration and exploitation when sampling, while the selected paths rollout till termination, (2) Best-of-N (Cobbe et al., 2021) performs multiple independent rollouts and selects the highest-scoring output, (3) Beam Search (Yao et al., 2023) expands multiple hypotheses in parallel, pruning low-scoring candidates at each step to maintain a fixed-width search (Snell et al., 2024). Since efficient tree search algorithms have recently regained attention after the emergence of LLM reasoning, the strong baselines are limited. As a result, we primarily compare DPTS against these typical and well-established search algorithms to demonstrate the effectiveness of our method.

C.2 Experimental Settings

The experimental settings are as follows: we set the tree width to 4, tree depth to 16, mini step to 100, and the maximum token limit to 2048. The MCTS time limit is 120 seconds, and the threshold parameter is empirically set to $t^* = 5$ on math tasks. We set $t = 20$ for a larger search space on code generation benchmark. All models were downloaded from Hugging Face.

Our default PRM is the math-shepherd-mistral-7b-prm as introduced in (Wang et al., 2023). This model is trained using automatically generated data without human annotations. Specifically, it leverages a completer to extend intermediate steps of a reasoning chain into multiple complete paths, then estimates the quality of each step based on the correctness of the resulting answers. This design aligns with the principle that high-quality steps are more likely to lead to correct final solutions.

For evaluation, we implemented a custom codebase. All of the experiments are conducted on NVIDIA A800 GPUs with 80GB of memory. Inference automatically terminates if it exceeds the timeout limit or encounters a memory overflow. Within these constraints, the search tree can expand and roll out indefinitely, ensuring comprehensive exploration during inference.

On math tasks, we record the correctness of the answers as the accuracy (Acc.). On code generation benchmark, we report *Pass@1* to measure the generation quality. *Pass* denotes the proportion of problem instances where at least one correct solution is found among the generated answers. *Pass@1*, on the other hand, refers to the probability that the first generated answer is correct.

C.3 Distribution of Best Path Index

We use a histogram to visualize the earliest (blue bar) and shortest (green bar) best path index. Through an ablation study, we examine how the best solution in the search path evolves as our method is progressively introduced.

In the baseline method, the best path typically appears around the 8th terminated path. Incorporating the parallelism streamline does not directly affect the accuracy of the search path. However, after adding the searching and transition mechanism, DPTS finds the best solution region more quickly and reaches the best path earlier.

C.4 Alleviating Frequent Switching

To investigate whether our method mitigates the issue of frequent node switching during the search process, we analyze and compare the switching frequency under different settings. We first report the switching count of standard non-parallel MCTS, which is 37 per sample (see Table 7). As the degree of parallelism increases, the number of switches also grows. For instance, with a parallelism of 4, MCTS exhibits an average of 122 switches per sample. A baseline strategy that always selects the node with the highest confidence at each generation step results in an even higher switching count of 354, due to the frequent redirection to different paths. In contrast, our proposed method achieves only 97 switches per sample. This demonstrates that our Search and Transition Mechanism effectively reduces frequent switching by employing a confidence threshold λ_{es} to early-stop low-confidence paths, while preserving the continuity of high-confidence explorations without frequent shifting when higher-scoring nodes emerge, thus significantly mitigating the switching issue.

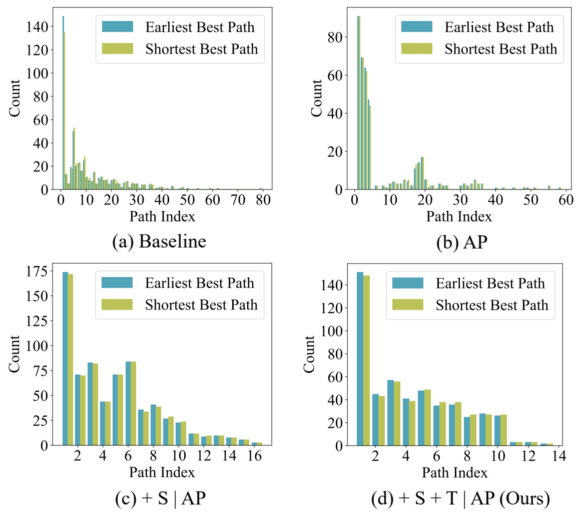


Figure 10: The distribution of the earliest (blue bar) and shortest (green bar) best path index.

Table 7: Average switch counts per sample under different methods.

Method	$\lvert P \rvert$	Switch Count (Mean$_{\text{Std}}$)
MCTS	1	37_{25}
MCTS	4	122_{44}
Max-confidence	AP	354_{98}
DPTS	AP	$\mathbf{97}_{42}$

C.5 Additional Results of λ

We conducted a more detailed experiment on λ, with results presented in Table 8 and Figure 11. It is evident that when λ is set within a reasonable range (e.g., $[0.7, 0.9]$), both accuracy and inference time exhibit optimal performance. In this range, the proportion of DS% (deep seek transitions) is higher than ES% (early stop transitions), indicating that more high-confidence nodes are being timely converted into exploit nodes. At the same time, a small number of paths are reassigned as low-score paths

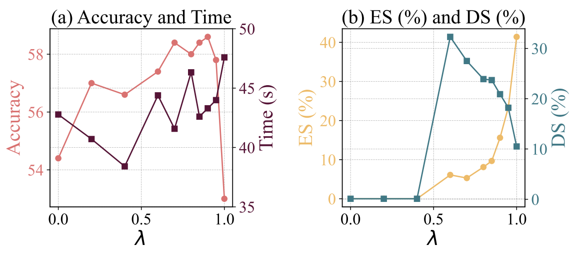


Figure 11: Illustrations of hyperparameter analysis.

Table 8: Hyperparameter λ_{es} and λ_{ds} in transition thresholds θ_{es} and θ_{ds}. "ES (Early Stop) %" and "DS (Deep Seek) %" are the ratios of the node type transition between the exploitation and exploration.

λ_{es}	λ_{ds}	**Acc.**	**Time (s)**	**ES (%)**	**DS (%)**
1.0	1.0	53.0	47.59	41.4	10.5
0.95	0.95	57.8	43.99	23.4	18.2
0.9	0.9	58.6	43.30	15.6	20.9
0.85	0.85	58.4	42.60	9.67	23.7
0.8	0.8	58.0	46.33	8.1	23.9
0.7	0.7	58.4	41.58	5.3	27.5
0.6	0.6	57.4	44.39	6.1	32.3
0.4	0.4	56.6	38.41	0	0.1
0.2	0.2	57.0	40.71	0	0.1
0	0	54.4	42.78	0	0.1

during inference and subsequently terminated.

However, when λ is too large (e.g., close to 1), the proportion of es increases aggressively. This suggests that many exploit nodes being expanded have scores within the range of $[0.9, 1.0]$, and setting the threshold in this range may cause some correct paths to be prematurely stopped. Conversely, when λ is too small (e.g., < 0.4), both ES and DS proportions drop to nearly zero. This occurs because most node scores exceed the threshold, causing nearly all paths to expand under exploitation mode. Moreover, an overly small early stop threshold causes no paths to terminate, effectively degrading the search into an exploitation-only strategy.

Therefore, selecting a suitable λ is important. A larger λ imposes stricter exploitation conditions, leading to more paths being stopped and fewer paths being converted to deep seeking. Conversely, a smaller λ results in looser conditions, allowing more exploiting paths to continue rolling out until they reach a termination condition, while more high-confidence paths transition into deep seeking.

Table 9: Comparison of search algorithms in terms of termination reasons, accuracy, and runtime.

Search algo.	**Stop by OOM (%)**	**Stop by Time (%)**	**Finish (%)**
MCTS	19.6	62.1	18.3
Ours	32.8	0.0	67.2

C.6 Stopping Cause Analysis

Our adaptive parallelism mechanism dynamically adjusts the number of parallel paths based on available memory. Under memory-constrained conditions, the level of parallelism gradually decreases and may eventually decrease to 1 until out-of-memory (OOM) occurs. Compared to the original MCTS, which always maintains a fixed parallelism of 1, our method starts with a higher degree of parallelism when memory is sufficient and adaptively reduces it as the KV cache and context computation consume more GPU memory.

To further investigate the behavior under GPU memory constraints and also to evaluate the performance of our method on larger models, we conducted experiments using Qwen2.5-14B-Instruct. We recorded the stopping cause for each sample and the overall performance. The results are summarized in Table 9.

We observed that performing multi-path inference in parallel with the larger model led to more frequent OOM occurrences. With our DPTS, 32.8% of the samples were stopped due to OOM. However, we discovered that 92.1% of those samples had already found at least one complete reasoning path before OOM occurred. Meanwhile, most samples (67.2%) successfully completed the search within a limited GPU memory. In contrast, although the non-parallel MCTS had a lower OOM rate, its slower search speed resulted in a significantly higher proportion of samples (62.1%) being terminated due to time limits, with only a small number of samples completing the search.

Overall, our method outperformed MCTS in both accuracy and time efficiency. It is attributed to the core objectives of DPTS, leveraging a space-time trade-off to maximize GPU utilization and accelerate reasoning.

D Related Work

Reasoning with LLMs. The success of ChatGPT (Achiam et al., 2023) has driven significant interest in Transformer-based LLMs (Bai et al., 2023; Touvron et al., 2023; Yang et al., 2024), initially applied to simple System 1 tasks like translation and summarization (Brown et al., 2020; Ouyang et al., 2022). As LLM capabilities grew, research shifted toward enhancing their ability to handle more complex System 2 tasks, such as mathematical reasoning and logic (Kojima et al., 2022; Hao et al., 2023; Zhang et al., 2024b; Hosseini et al., 2024). The introduction of Chain-of-Thought (CoT) (Wei et al., 2022) advanced this domain by breaking down problems into intermediate steps, which proved effective for multi-step reasoning (Kojima et al., 2022). Recent research (Zhang et al., 2022; Bi et al., 2024) has proposed numerous variations, such as Self-Consistent CoT (Wang et al., 2022), R-CoT (Deng et al., 2024) to improve its ability, but its exploration scope remains constrained, limiting its effectiveness. (Chu et al., 2023).

Tree Search for Reasoning. To address the limitations of linear reasoning in CoT, the Tree of Thoughts (ToT) (Yao et al., 2023) framework was introduced, which aligns with the inference scaling law, showing that increasing Test-Time Compute can enhance LLM reasoning abilities (Snell et al., 2024; Wu et al., 2024). The simplest form of tree search, Best-of-N, samples multiple reasoning paths and selects the best one (Cobbe et al., 2021; Lightman et al., 2023; Jiao et al., 2024). Beam search extends this approach by considering multiple paths in parallel (Yao et al., 2023). Recent work (Hao et al., 2023) has leveraged the exploration capabilities of Monte Carlo Tree Search (MCTS) (Chaslot et al., 2008; Kocsis et al., 2006; Browne et al., 2012). For instance, MCTS-rollout (Wan et al., 2024b) introduces backtracking to explore different paths, while Q* (Wang et al., 2024a) uses heuristic functions to guide rollout, and ReST-MCTS* (Zhang et al., 2024a) uses MCTS to sample traces for self-training. However, challenges remain in its computational efficiency, with limited research on accelerating tree search for LLM reasoning.

LLM Inference Acceleration. LLMs face efficiency bottlenecks during inference, leading to significant efforts aimed at accelerating inference (Lin et al., 2024; Hong et al., 2024; Leviathan et al., 2023; Fu et al., 2024; Jing et al., 2023; Cai et al., 2024). However, most of these approaches are designed for linear decoding tasks and are not directly applicable to tree-structured reasoning (Li et al., 2024a; Zhou et al., 2024). Recent work (Ning et al., 2024; Han et al., 2024; Nayab et al., 2024), such as Deft (Yao et al., 2024b) focuses on optimizing common prefixes in tree reasoning using operator-level enhancements, , while others leverage self-consistency to enable early stopping (Li et al., 2024b; Wan et al., 2024a). SEED (Wang et al., 2024b) adopts a speculative decoding paradigm by using multiple small models in parallel to generate draft outputs, followed by a target model to verify and select the best candidate. The method by Sun et al. (Sun et al., 2024) is based on the best-of-N and is particularly effective when N is large (e.g. $N \in [960, 3840]$). It progressively reduces batch size via early stopping, retaining only the highest-rewarded sequences. While in ToT, we typically adopt a much smaller N (e.g., up to 8), greatly limiting the benefits of the algorithm in our context. Despite such advancements, building efficient tree search algorithms for reasoning remains an under-explored area. The need for specialized algorithms that can scale with complex tree-structured reasoning, while maintaining efficiency, remains a key open challenge in LLM inference acceleration.

E Future Work Discussion

Our method inherently supports multi-batch processing, which aligns well with the overall design. In practice, acceleration under a multi-batch setting can be realized by adjusting the number of nodes expanded at each search step to match the batch size, enabling concurrent processing across multiple requests.

Formally, we can incorporate batch size explicitly into Eq. (1) in a revised formulation of the algorithm. Furthermore, during the handling of input, output, and context key-value (KV) caches, as described in Eq. (3), (4) and (5), we will concatenate multi-batch data such that each request preserves an independent reasoning tree. This design ensures that requests proceed in parallel without mutual interference while efficiently sharing the available GPU memory.

However, we acknowledge that in the current version of our implementation, multi-batch processing may reduce the degree of parallelism under memory constraints, potentially degrading to the worst-case scenario where the effective paralleled number of nodes equals one, equivalent to the standard MCTS. Meanwhile, it is worth noting that due to potential differences in the depth and length of reasoning paths across requests, static batching may result in underutilization of GPU resources.

To mitigate this, we identify continuous batching as a promising enhancement. By dynamically forming micro-batches from pending requests, continuous batching could better exploit idle GPU resources and further improve system throughput. We consider this a valuable direction for future work.