Transformers in the Dark: Navigating Unknown Search Spaces via Noisy Feedback

Jungtaek Kim[1] **Ziqian Lin**[1] **Thomas Zeng**[1] **Minjae Lee**[3]
Chungpa Lee[4] **Jy-yong Sohn**[4] **Hyung Il Koo**[3] **Kangwook Lee**[1,2]
[1]University of Wisconsin–Madison [2]KRAFTON AI [3]FuriosaAI [4]Yonsei University
{jungtaek.kim,kangwook.lee}@wisc.edu

Abstract

Effective problem solving with Large Language Models (LLMs) can be enhanced when they are paired with external search algorithms. In particular, the search algorithm can navigate a tree-structured search space and guide the LLM toward better solutions more efficiently. While the search algorithm enables effective exploitation and exploration, the need for an external component can complicate the overall problem-solving process. We therefore pose the following question: *can LLMs or their underlying Transformer architectures fully internalize the search algorithm?* To answer this question, we introduce a simplified framework in which tree extensions and feedback signals are externally specified. Under this setting, we show Transformers can implement various search strategies through theoretical weight construction and empirical behavior cloning.

Introduction Effective problem solving often follows an iterative process of (i) generating diverse idea candidates, (ii) selecting the most promising one to pursue, and (iii) evaluating its potential. Large Language Models (LLMs) [2, 18, 1, 5, 7, 23] suggest that these systems may implicitly implement such a process. As discussed in previous literature [41, 9, 45], this problem-solving process can be enhanced when LLMs are paired with external search algorithms. In particular, the search algorithm navigate a tree-structured space, allowing it to efficiently guide the LLM toward better solutions. While the search algorithm lets the model exploit and explore effectively, relying on an external component can complicate the overall process. We therefore raise this question: *can LLMs or their underlying Transformer architectures fully internalize the search algorithm?* Thus, we introduce a synthetic framework where the search space is defined as a tree with sparse initially unknown rewards, and the agent only receives noisy bandit feedback. Finally, we show this expressivity in two ways: theoretically, by constructing explicit Transformer weights that implement these algorithms, and empirically, by training Transformers from scratch that behaviorally clone the reference algorithms.

Problem Setup Each problem instance consists of a finite rooted search tree $\mathcal{T} = (\mathcal{S}, N)$ with maximum depth D, where $\mathcal{S}$ is a state set and $N : \mathcal{S} \to 2^{\mathcal{S}}$ is a successor function; and also a bounded reward function $r : \mathcal{S} \to [0, 1]$. The goal states $\mathcal{S}_{\text{goal}} = s \in \mathcal{S} : r(s) > 0$ are leaves. The tree $\mathcal{T}$ is hidden from the agent. We further define the value $V(s)$ as a distribution that provides noisy estimates of the expected reward from state s (e.g., via Monte Carlo rollouts). Under this setting, the agent interacts with the environment for $T \ll |\mathcal{S}|$ steps. Starting from the root s_0, at each step t the environment reveals the children $N(s_t)$ and a value sample $v_t \sim V(s_t)$. The agent then selects the next state s_{t+1} from the frontier of unvisited children, according to its policy. This produces a trajectory $\tau = [(s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))]$.

Theoretical Analysis We first show that Transformers [38] can implement various search algorithms introduced in Section B.2, by simulating each step of the algorithms. We provide explicit weight constructions for Transformers with constant depth and embedding dimension linear in the search budget T and branching factor B.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: What Can('t) Transformers Do?.

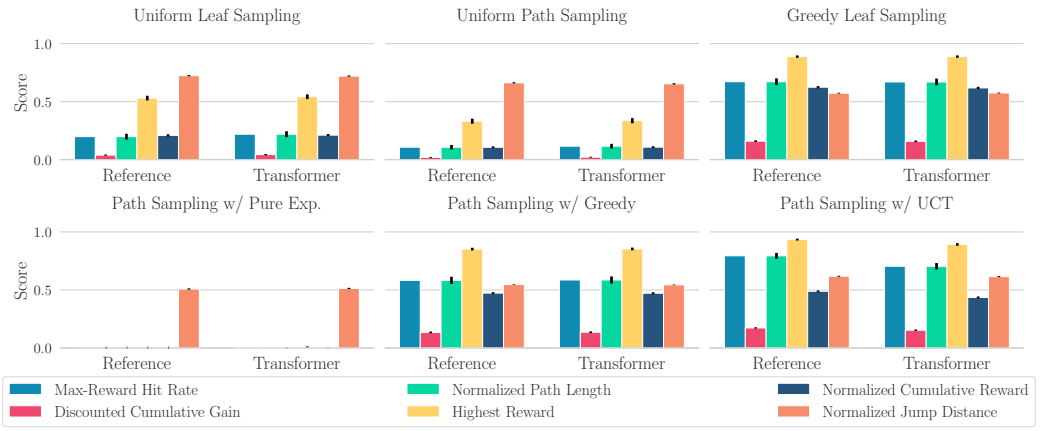
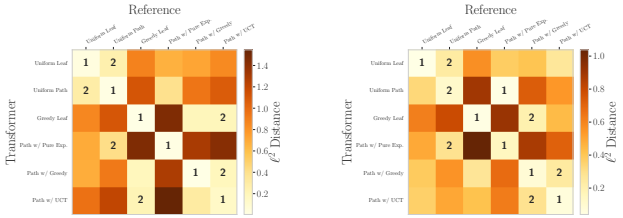


Figure 1: Behavior cloning results on the multi-reward tree search problem, where each binary tree of depth 6 has 8 different goals and a search step budget is 50.



(a) Multi-reward tree search (b) Multi-reward navigation

Figure 2: Comparison of the metric values obtained by reference algorithms and Transformers using the results in Figures 1 and 8. The shortest and second shortest distances in each row are marked.

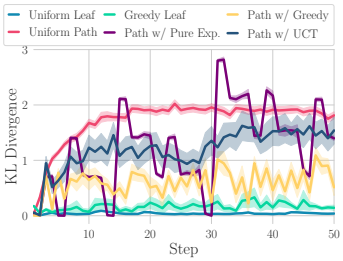


Figure 3: KL divergences between the probabilities of references and Transformers.

Theorem 1 *There exist 3-layer Transformers with embedding dimension $d = \Theta(B\,T)$ that exactly implements the uniform and greedy leaf sampling policies when using a sequential encoding of the trajectory using tokens linear in the number of search iterations.*

Theorem 2 *There exist 12-layer Transformers with embedding dimension $d = \Theta(B\,T)$ that exactly implements random, greedy and UCT based path-sampling policies when using a sequential encoding of the trajectory using tokens quadratic in the number of search iterations.*

The details of exact tokenization for the trace format are presented in Section D.1, and the full proofs of Theorems 1 and 2 are in Sections E.2 and E.3.

Empirical Analysis Following our theoretical analysis, we empirically test whether Transformers can imitate the search algorithms via behavioral cloning. We demonstrate that Transformers trained from scratch are capable of effectively performing behavior cloning in the two environments shown in Section B.1. Tokenization methods are detailed in Section D.2, and other details are in Section F.2.

As shown in Figures 1 and 8, Transformers successfully mimic the performance of the respective reference algorithms. To quantitatively analyze these results, we calculate ℓ^2 distance between two vectors of the statistics of the metric values obtained by a reference algorithm and its corresponding Transformer model. We simply employ their means and standard deviations of all six metric values to measure the distance. Figure 2 presents that the results of Transformers are mostly closest to ones of the associated reference algorithms. Moreover, Figure 3 shows KL divergence results between reference algorithms and Transformers. Missing detailed discussion is described in Section G.

Conclusion In this paper, we have introduced synthetic and fully controllable search benchmarks. Our theoretical and empirical analyses demonstrate that Transformers can effectively implement classical search strategies within these environments. Thus, our findings suggest that the reasoning abilities of LLMs can be investigated through the framework of search.

Acknowledgments and Disclosure of Funding

This work was supported by National Science Foundation (NSF) Award DMS-2023239, NSF CAREER Award CCF-2339978, Amazon Research Award, Google Cloud Research Credits Program, and a grant from FuriosaAI. We also acknowledge support from the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (MSIT), Korea (RS-2024-00345351, RS-2024-00408003); the ICT Challenge and Advanced Network of HRD (ICAN) support program (RS-2023-00259934, RS-2025-02283048) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP); and the Yonsei University Research Fund (2025-22-0025).

References

[1] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R. J. Hewett, M. Javaheripi, P. Kauffmann, J. R. Lee, Y. T. Lee, Y. Li, W. Liu, C. C. T. Mendes, A. Nguyen, E. Price, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, X. Wang, R. Ward, Y. Wu, D. Yu, C. Zhang, and Y. Zhang. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*, 2024.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901, Virtual, 2020.

[3] S. Bubeck, R. Munos, G. Stoltz, and C. Szepesvári. x-armed bandits. *Journal of Machine Learning Research*, 12(5):1655–1695, 2011.

[4] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. Decision Transformer: Reinforcement learning via sequence modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 34, pages 15084–15097, Virtual, 2021.

[5] DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[6] N. Dziri, X. Lu, M. Sclar, X. L. Li, L. Jiang, B. Y. Lin, P. West, C. Bhagavatula, R. Le Bras, J. D. Hwang, S. Sanyal, S. Welleck, X. Ren, A. Ettinger, Z. Harchaoui, and Y. Choi. Faith and fate: Limits of Transformers on compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 70293–70332, New Orleans, Louisiana, USA, 2023.

[7] Gemini Team, Google. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[8] T. Han, Z. Wang, C. Fang, S. Zhao, S. Ma, and Z. Chen. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL*, pages 24842–24855, Vienna, Austria, 2025.

[9] S. Hao, Y. Gu, H. Ma, J. J. Hong, Z. Wang, D. Z. Wang, and Z. Hu. Reasoning with language model is planning with world model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173, Singapore, Singapore, 2023.

[10] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[11] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

[12] S. Kambhampati, K. Valmeekam, L. Guan, M. Verma, K. Stechly, S. Bhambri, L. Saldyt, and A. Murthy. Position: LLMs can't plan, but can help planning in LLM-Modulo frameworks. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 22895–22907, Vienna, Austria, 2024.

[13] M. Khalifa, R. Agarwal, L. Logeswaran, J. Kim, H. Peng, M. Lee, H. Lee, and L. Wang. Process reward models that think. *arXiv preprint arXiv:2504.16828*, 2025.

[14] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, pages 282–293, Berlin, Germany, 2006.

[15] M. Laskin, L. Wang, J. Oh, E. Parisotto, S. Spencer, R. Steigerwald, D. Strouse, S. S. Hansen, A. Filos, E. Brooks, M. Gazeau, H. Sahni, S. Singh, and V. Mnih. In-context reinforcement learning with algorithm distillation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Kigali, Rwanda, 2023.

[16] L. Lehnert, S. Sukhbaatar, D. Su, Q. Zheng, P. McVay, M. Rabbat, and Y. Tian. Beyond A*: Better planning with Transformers via search dynamics bootstrapping. In *Proceedings of the Conference on Language Modeling (COLM)*, Philadelphia, Pennsylvania, USA, 2024.

[17] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. Let's verify step by step. In *Proceedings of the International Conference on Learning Representations (ICLR)*, pages 39578–39601, Vienna, Austria, 2024.

[18] Llama Team, AI @ Meta. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[19] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Vancouver, British Columbia, Canada, 2018.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[21] N. Nolte, O. Kitouni, A. Williams, M. Rabbat, and M. Ibrahim. Transformers can navigate mazes with multi-step prediction. *arXiv preprint arXiv:2412.05117*, 2024.

[22] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[23] OpenAI. Introducing GPT-5. `https://openai.com/index/introducing-gpt-5/`, 2025.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, pages 8026–8037, Vancouver, British Columbia, Canada, 2019.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[26] J. Pérez, J. Marinković, and P. Barceló. On the turing completeness of modern neural network architectures. In *Proceedings of the International Conference on Learning Representations (ICLR)*, New Orleans, Louisiana, USA, 2019.

[27] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition, 2010.

[28] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[29] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587): 484–489, 2016.

[30] C. Snell, J. Lee, K. Xu, and A. Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, pages 10131–10165, Singapore, Singapore, 2025.

[31] D. Su, S. Sukhbaatar, M. Rabbat, Y. Tian, and Q. Zheng. Dualformer: Controllable fast and slow thinking by learning with randomized reasoning traces. In *Proceedings of the International Conference on Learning Representations (ICLR)*, pages 95080–95117, Singapore, Singapore, 2025.

[32] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo, and Y. Liu. RoFormer: Enhanced Transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

[33] Z. Sun, L. Yu, Y. Shen, W. Liu, Y. Yang, S. Welleck, and C. Gan. Easy-to-hard generalization: Scalable alignment beyond human supervision. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pages 51118–51168, Vancouver, British Columbia, Canada, 2024.

[34] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[35] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[36] K. Valmeekam, M. Marquez, A. Olmo, S. Sreedharan, and S. Kambhampati. PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 38975–38987, New Orleans, Louisiana, USA, 2023. Datasets and Benchmarks Track.

[37] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati. On the planning abilities of large language models: a critical investigation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 75993–76005, New Orleans, Louisiana, USA, 2023.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, pages 5998–6008, Long Beach, California, USA, 2017.

[39] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272, 2020.

[40] G. Weiss, Y. Goldberg, and E. Yahav. Thinking like Transformers. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 11080–11090, Virtual, 2021.

[41] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of Thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pages 11809–11822, New Orleans, Louisiana, USA, 2023.

[42] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, Kigali, Rwanda, 2023.

[43] D. Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural Networks*, 94: 103–114, 2017.

[44] C. Yun, S. Bhojanapalli, A. S. Rawat, S. J. Reddi, and S. Kumar. Are Transformers universal approximators of sequence-to-sequence functions? In *Proceedings of the International Conference on Learning Representations (ICLR)*, Virtual, 2020.

[45] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 62138–62160, Vienna, Austria, 2024.

A Related Work

In this section, we briefly review the background and related work relevant to this paper.

Blind and Uninformed Search Blind or uninformed search refers to algorithms that explore a search space without heuristic guidance or prior knowledge of its structure [27]. Classical methods such as Breadth-First Search (BFS), Depth-First Search (DFS), and uniform-cost search systematically enumerate nodes, guaranteeing completeness and, in some cases, optimality, but scale poorly in large or sparse environments due to their inability to incorporate feedback during search [27]. Modern extensions introduce adaptivity via stochastic sampling and online feedback. MCTS [34], particularly upper confidence bounds applied to trees (UCT) [14], balances exploration and exploitation by applying bandit principles to tree expansion and has proven highly effective in diverse domains such as Go [29] and Atari games [20]. Bandit-based tree search has been further studied in the context of regret minimization and sample complexity [3].

Learning-Based Planning It denotes training models to acquire planning abilities from data, allowing them to generate action sequences or policies for solving tasks under uncertainty without hand-crafted search. MuZero learns both dynamics model and search policy directly from interaction, achieving strong performance without access to game rules [28]. Valmeekam et al. [36] propose benchmarks for evaluating LLMs on planning and reasoning, and given existing pretrained LLMs such as GPT-3.5 and GPT-4 [22], Valmeekam et al. [37] analyze their planning abilities. Chen et al. [4] predict actions given the history of states, actions, and rewards, using the Transformer architecture. In addition, historical episodes are collected from a source RL algorithm and then they are used to autoregressively train a Transformer model [15], which is called algorithmic distillation. Recent studies [16, 21, 31] propose intriguing methods to train the Transformer-based model on the traces of A* search. In particular, the method by Nolte et al. [21] is capable of controlling Transformers' outputs using either fast or slow mode. This line of research provides the entire environment to the models, unlike our problem formulation. Beyond a framework with reasoning and acting [42], Zhou et al. [45] make use of the MCTS algorithm in decision-making with LLMs. Kambhampati et al. [12] introduce the LLM-Modulo framework, in which LLMs generate ideas and serve as external critics.

Reasoning Abilities of LLMs Popular LLMs [5, 7, 23] enhance its performance using a reasoning mechanism. Lightman et al. [17] compare outcome-supervised and process-supervised reward models for guiding a pretrained LLM's chain-of-thought reasoning, showing that step-level supervision yields more reliable reasoning. Dziri et al. [6] find that Transformers encounter limitations in compositional multi-step reasoning, leading to performance degradation as task complexity increases. Sun et al. [33] show that the reward models trained on easier problems can generalize to supervise harder ones, enabling scalable alignment without direct human feedback. Snell et al. [30] investigate that allocating inference-time compute effectively can improve LLM performance, comparing larger models under equal budgets. Han et al. [8] introduce a framework that dynamically considers chain-of-thought token budgets to the complexity of each problem, reducing the number of tokens with minimal loss in accuracy. Khalifa et al. [13] propose ThinkPRM, a generative chain-of-thought process reward model that, with limited step-level supervision, surpasses both LLM-as-a-judge and discriminative PRMs on several benchmarks.

B Problem Formulation and Model Interfaces

Each problem instance is defined by a finite, rooted search tree $\mathcal{T} = (\mathcal{S}, N)$ with maximum depth D, where $\mathcal{S}$ is a finite set of states and $N : \mathcal{S} \to 2^{\mathcal{S}}$ is a successor function mapping each state to its children. We assume a bounded reward function $r : \mathcal{S} \to [0, 1]$ and define the set of *goal states* as $\mathcal{S}_{\text{goal}} = \{s \in \mathcal{S} : r(s) > 0\}$, requiring each goal state to be a leaf node in $\mathcal{T}$. Goal states are assumed to be sparse, i.e., $|\mathcal{S}_{\text{goal}}| \ll |\mathcal{S}|$, with most states yielding zero reward, reflecting realistic scenarios where solutions are rare. Importantly, the tree $\mathcal{T}$ constitutes the underlying search space, which remains hidden from the search agent.

We define the *value* of a state s under a uniform random rollout policy as the following:

$$V(s) = \mathbb{E}_{\pi_{\text{unif}}}[r(s_{\text{leaf}}) \mid s_{\text{start}} = s], \tag{1}$$

where π_{unif} denotes a uniform random traversal policy through the tree, and s_{leaf} represents the leaf state reached after traversing from the initial state $s_{\text{start}} = s$. To simulate real-world scenarios where ground-truth feedback is unavailable, we approximate $V(s)$ by a random variable $\widehat{V}(s)$. Each realization of $\widehat{V}(s)$ is obtained via a Monte Carlo estimate of $V(s)$, generated by performing k independent rollouts using π_{unif}. These realizations serve as noisy bandit feedback signals received by the search agent and act as proxies for real-world situations where the agent, such as an LLM, must internally estimate the value of each searched state.

In our experiments involving pretrained LLMs on "real-world" search trees, we directly utilize this realistic scenario by employing an alternative formulation for $\widehat{V}(s)$, in which the LLM directly estimates the state values (e.g., by prompting the LLM to produce a score between 0 and 1).

Given the above setup, the agent-environment interaction occurs over T steps, where $T \ll |\mathcal{S}|$. Starting from the root state s_0, at each step t, the environment reveals the children $N(s_t)$ and a sampled rollout value $v_t \sim \widehat{V}(s_t)$. Defining the frontier of unvisited child states as the following:

$$F_t = (\bigcup_{i=0}^{t} N(s_i)) \setminus \{s_0, s_1, \ldots, s_t\}, \tag{2}$$

the search agent selects the state for the next step, $s_{t+1} \in F_t$, according to its policy $\pi(\cdot \mid s_0, v_0, N(s_0), \ldots, s_t, v_t, N(s_t))$. After T steps, we obtain a completed search trajectory τ:

$$\tau = [(s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))]. \tag{3}$$

We evaluate τ by the reward achieved at its best (highest-reward) state, $r(s_{\text{best}}) = \max_{s_i \in \tau} r(s_i)$. For later convenience, we collect the individual state rewards along the trajectory into the set:

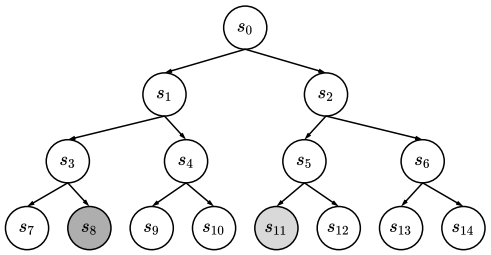
$$\mathcal{R} = \{r(s_i) \mid (s_i, v_i, N(s_i)) \in \tau\}. \tag{4}$$

With this notation, the previous expression can be written simply as $r(s_{\text{best}}) = \max \mathcal{R}$. In Section F.1, we provide additional diverse metrics, which are computed as functions of $\mathcal{R}$, to provide a more nuanced understanding of each search strategy.

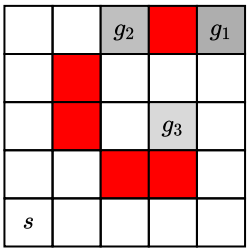
B.1 Specific Instances with Tree-Structured Search Spaces

Given the general setup, we detail two concrete problems with tree-structured search spaces; their illustrations are shown in Figure 4. The settings of these synthetic problems can be easily controlled to adjust their difficulty, which enables us to conduct the analysis of our Transformer models.

Multi-Reward Tree Search This problem is a straightforward implementation of our setup, by directly generating a randomized search tree. It is defined with three parameters: (i) a branching factor at each intermediate state B; (ii) a tree depth D; (iii) the number of goal states K. The number of accessible states is $(B^{D+1} - B)/(B - 1)$, where a root state s_0 is excluded since it is initially given. All goal states are located in the leaf nodes of the tree. In Figure 4a, s_0, s_1, …, s_{14} are all states in this binary tree, and s_8 and s_{11} are goal states where $r(s_8) > r(s_{11})$.



(a) Multi-reward tree search



(b) Multi-reward navigation

Figure 4: Two environments investigated in this work, where darker cells represent higher reward values and red cells denote cells that are impassable.

Multi-Reward Navigation We define a multi-reward maze traversal task within our problem setup. The objective of this task is to find paths from a designated start vertex v_{start} to one of several goal vertices. Specifically, consider an undirected graph $\mathcal{G} = (V, E)$ representing a two-dimensional grid of size $w \times h$, where $|V| = wh$ and the edges E connect neighboring vertices (i.e., up, down, left, right). Some of wh vertices, which are called *walls*, cannot be passed. Here, we define a wall density n_{wall}/wh, where n_{wall} is the number of wall vertices. The graph contains a designated start vertex $v_{\text{start}} \in V$ and a set of goal vertices $G = \{g_1, \ldots, g_k\} \subseteq V$, each associated with a strictly decreasing positive reward:

$$R(g_1) > R(g_2) > \cdots > R(g_k) > 0, \tag{5}$$

and $R(v) = 0$ for all non-goal vertices $v \notin G$. The branching factor B is naturally bounded by 4 due to the grid structure.

To formulate this problem as a search problem within our framework, we define the search space $\mathcal{S}$ as the set of all valid paths $\rho = (v_0, \ldots, v_\ell)$ in $\mathcal{G}$, satisfying the following constraints: (i) the initial vertex is fixed as $v_0 = v_{\text{start}}$, (ii) each path has length at most T, and (iii) each path visits at most one goal vertex, which if visited, must be the terminal vertex of the path. The successor function is defined by $N(\rho) = \{\rho \circ v \in \mathcal{S} : v \in V\}$, where $\rho \circ v$ denotes appending vertex v to the path ρ. Finally, the reward function is defined as $r(\rho) = R(v_\ell)$, assigning the reward of the terminal vertex v_ℓ to the entire path. This construction naturally yields a tree representation suitable for our general search framework.

More specifically, in Figure 4b, an agent is started from a start vertex s and then it can select one of neighbor vertices (i.e., up, down, left, and right vertices of the vertex of interest) sequentially. Red cells indicate impassible vertices. Similar to the binary tree example, $r(g_1) > r(g_2) > r(g_3)$. It is noteworthy that we express states as paths from s to a particular vertex which is shown in Figure 4b, so that it can straightforwardly create a tree-structured search space.

B.2 Reference Search Algorithms

Here, we introduce several reference search algorithms to solve the problem discussed above.

Uniform Leaf Sampling This method chooses one of F_{t-1} uniformly at random at iteration t: $s_t \sim \mathcal{U}(F_{t-1})$, where $\mathcal{U}$ is a uniform distribution. Refer to Algorithm 2 for details.

Greedy Leaf Sampling This algorithm selects one of F_{t-1} whose parent state has the highest reward value: $s_t = \arg\max_{s \in F_{t-1}} \hat{v}(s)$, where $\hat{v}(s)$ is the estimated value of the parent state of s. Refer to Algorithm 3 for details.

Uniform Path Sampling This strategy uniformly samples the next node at each depth level starting from a root state s_0 until it meets an instance in F_{t-1}. See Algorithm 4 for details.

Policy-Guided Path Sampling These methods traverse an underlying tree using one of tree traversal polices: pure exploration, greedy, and UCT policies; see Algorithm 5 for details. It is analogous to the conventional MCTS algorithm [34], but differs in that it excludes fully-explored sub-trees; refer to Algorithm 1 and its corresponding description.

These search strategies are employed to analyze the abilities of Transformers to navigate structured yet unknown environments. The pseudocode of these algorithms is shown in Section C. For the sake of brevity, uniform leaf sampling and greedy leaf sampling are categorized as *leaf-based sampling*, and uniform path sampling and policy-guided path sampling are categorized as *path-based sampling*.

B.3 Model Interfaces to Perform Search

Our problem setup can be viewed as a multi-turn interaction between an agent and an environment: at each step, the agent sends the environment a message containing the node s it wants to visit, and the environment replies with the node's value v and neighbors $N(s)$ (or unvisited states F).

This formulation allows Transformers or potentially any autoregressive next-token predictor to serve as search policies by predicting the agent's messages in the conversation conditioned on the chat history. We analyze this search capability through two complementary lenses:

- Theoretically demonstrating the existence of Transformer's parameters that implement search algorithms;
- Training Transformers on algorithmic traces.

See Section D for detailed implementation of the tokenization schemes for each of these setups.

C Details and Pseudocode of Reference Search Algorithms

In this section, we provide the pseudocode of reference search algorithms.

Algorithm 1 Fully-Explored Sub-Tree Exclusion

Input: The current state s, unvisited child states F, a hidden search tree $\mathcal{T} = (\mathcal{S}, N)$.
Output:
Obtain all visitable child states of s, denoted as $\mathcal{C}$.
if $\mathcal{C} \cap F = \emptyset$ **then**
    return `True`
else
    return `False`
end if

Algorithm 1 verifies if all possible child states of a state s are all explored. This algorithm is denoted as $E(s \mid F, \mathcal{T})$, where F is unvisited child states and $\mathcal{T}$ is a hidden search tree. If $E(s \mid F, \mathcal{T})$ is `True`, a search algorithm should exclude this sub-tree of which the root is s from further search. Importantly, with this process, underlying tree structures are not given to search algorithms, and we only provide unvisited states to the algorithms.

Without loss of generality, we can define a modified successor function N^*, which only provides immediate child states that have unvisited states:

$$N^*(s) = \{s' \in N(s) \mid \neg E(s' \mid F, \mathcal{T})\}, \tag{6}$$

where $\neg$ is a logical not operation.

C.1 Uniform Leaf Sampling

Algorithm 2 Uniform Leaf Sampling

Input: A root state s_0, a value estimation function $\widehat{V}(s)$, a step budget T, a hidden search tree $\mathcal{T} = (\mathcal{S}, N)$.
Output: A search trajectory $\tau = [(s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))]$.
Initialize a trajectory $\tau = [(s_0, v_0, N(s_0))]$.
Update unvisited child states F_0.
for $t = 1, 2, \ldots, T$ **do**
    Choose the next state s_t from F_{t-1} uniformly at random.
    Evaluate s_t to obtain its value v_t using $\widehat{V}(s_t)$.
    Update τ with $(s_t, v_t, \widehat{V}(s_t))$.
    Update unvisited child states F_t.
end for

Algorithm 2 shows the pseudocode of uniform leaf sampling. This algorithm can be considered as the randomized algorithm of DFS.

C.2 Greedy Leaf Sampling

Algorithm 3 Greedy Leaf Sampling

Input: A root state s_0, a value estimation function $\widehat{V}(s)$, a step budget T, a hidden search tree $\mathcal{T} = (\mathcal{S}, N)$.
Output: A search trajectory $\tau = [(s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))]$.

Initialize a trajectory $\tau = [(s_0, v_0, N(s_0))]$.
Update unvisited child states F_0.
for $t = 1, 2, \ldots, T$ **do**
    Choose a state s'_t from the parent states of F_{t-1} which has the highest reward estimated value.
    Sample the next state s_t from child states $N(s'_t)$ uniformly.
    Evaluate s_t to obtain its value v_t using $\widehat{V}(s_t)$.
    Update τ with $(s_t, v_t, \widehat{V}(s_t))$.
    Update unvisited child states F_t.
end for

Algorithm 3 demonstrates the pseudocode of greedy leaf sampling. It considers the reward values of the parent states of unvisited states.

C.3 Uniform Path Sampling

Algorithm 4 Uniform Path Sampling

Input: A root state s_0, a value estimation function $\widehat{V}(s)$, a step budget T, a hidden search tree $\mathcal{T} = (\mathcal{S}, N)$.
Output: A search trajectory $\tau = [(s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))]$.

Initialize a trajectory $\tau = [(s_0, v_0, N(s_0))]$.
Update unvisited child states F_0.
for $t = 1, 2, \ldots, T$ **do**
    Assign a state as $s'_t = s_0$.
    while $s'_t \notin F_{t-1}$ **do**
        Choose one of $N^*(s'_t)$ uniformly at random, ensuring that fully-explored sub-trees are not considered.
        Update s'_t using the sampled state.
    end while
    Assign the next state as $s_t = s'_t$
    Evaluate s_t to obtain its value v_t using $\widehat{V}(s_t)$.
    Update τ with $(s_t, v_t, \widehat{V}(s_t))$.
    Update unvisited child states F_t.
end for

Algorithm 4 presents a search algorithm of uniform path sampling. In contrast to uniform leaf sampling, this algorithm can be viewed as a randomized variant of BFS. The definition of N^* is shown in (6).

C.4 Path Sampling with Pure Exploration, Greedy, or UCT Policy

Algorithm 5 Path Sampling with a Specific Tree Traversal Policy

Input: A root state s_0, a value estimation function $\widehat{V}(s)$, a tree traversal policy $\mathcal{P}$, a step budget T, a hidden search tree $\mathcal{T} = (\mathcal{S}, N)$.
Output: A search trajectory $\tau = [(s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))]$.

```
Initialize a trajectory τ = [(s_0, v_0, N(s_0))].
Update unvisited child states F_0.
for t = 1, 2, ..., T do
    Assign a state as s'_t = s_0.
    while s'_t ∉ F_{t-1} do
        Determine unvisited immediate child states F̃_{t-1} = {s : s ∈ N(s_t), s ∈ F_{t-1}}.
        if |F̃_{t-1}| > 0 then
            Choose one of F̃_{t-1} uniformly at random.
        else
            Choose one of N*(s'_t) with a specific tree traversal policy T; see Algorithm 6, ensuring
            that fully-explored sub-trees are not considered.
        end if
        Update s'_t using the chosen state.
    end while
    Assign the next state as s_t = s'_t
    Evaluate s_t to obtain its value v_t using V̂(s_t).
    Update τ with (s_t, v_t, V̂(s_t)).
    Update unvisited child states F_t.
end for
```

Algorithm 5 shows a search strategy of path sampling with a specific tree traversal policy such as a pure exploration, greedy, or UCT policy. N^* is defined in (6).

Algorithm 6 Tree Traversal Policies

Input: A tree traversal policy $\mathcal{P}$, the current state s, modified successor function N^*.
Output: A selected next state s^*.

Pure Exploration Policy

$s^* = \arg\min_{s' \in N^*(s)} \mathrm{count}(s')$, where count returns the number of visit counts.

Greedy Policy

$s^* = \arg\max_{s' \in N^*(s)} \mathrm{value}(s')$, where value returns the summation of all values of the child nodes of s'.

UCT Policy

$s^* = \arg\max_{s' \in N^*(s)} \mathrm{value}(s')/\mathrm{count}(s') + c\sqrt{\log(2\,\mathrm{count}(s))/\mathrm{count}(s')}$, where c is a balancing hyperparameter.

Algorithm 6 presents the details of tree traversal polices. The pure exploration policy only considers the number of visit counts and chooses the least visited state as the next state. The greedy policy takes into account the values of child states in order to choose the next state. The UCT policy [14] balances exploration and exploitation considering both the number of visits and the estimated values. Across all experiments, we use $c = 0.1$. See this reference [34] for more details.

D Trace Formats

This section presents the trace formats used for theoretical constructions and Transformers trained from scratch.

D.1 Trace Format for Theoretical Constructions

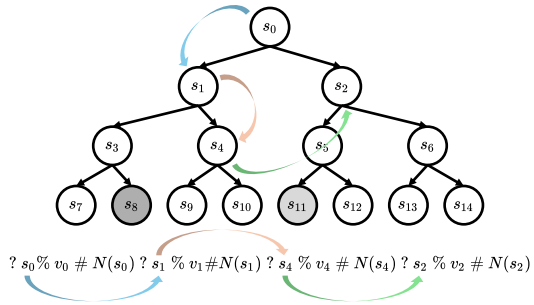


Figure 5: Visualization of our tokenization scheme. Same colors indicate same state transitions.

In our theoretical analysis, we represent each search trace as a sequence of discrete tokens. Given a search step budget T and a maximum branching factor B, the trace of the search trajectory can contain at most $TB+1$ distinct states as there is also the root node. For simplicity, we assume exactly $TB+1$ states in total.

Leaf-Based Tokenization We define three token types for leaf-based search methods:

- State tokens S_i for each unique state i;
- Continuous value tokens V_α, where $\alpha \in [0,1]$ is stored in a dedicated coordinate in the embedding;
- Structural markers `%`, `#`, and `?`.

Let a trajectory be

$$\tau = ((s_0, v_0, N(s_0)), (s_1, v_1, N(s_1)), \ldots, (s_T, v_T, N(s_T))),$$

where $N(s_t) = \{s_{t,1}, s_{t,2}, \ldots\}$ is the set of s_t's child states. We then construct the tokenized trace as the following:

$$\texttt{?}\ S_{s_0}\ \texttt{\%}\ V_{v_0}\ \texttt{\#}\ S_{s_{0,1}}\ S_{s_{0,2}}\ \ldots\ \texttt{?}\ S_{s_1}\ \texttt{\%}\ V_{v_1}\ \texttt{\#}\ S_{s_{1,1}}\ S_{s_{1,2}}\ \ldots\ \texttt{?}\ \ldots$$

for $t = 0, 1, \ldots, T$. Here:

- `%` separates the selected state from its value embedding.
- `#` precedes the list of child states.
- `?` terminates each search iteration.

Note that `%` is not required for the leaf-based sampling theoretical construction and it is added to ensure consistency with the tree-based sampling methods below. We will construct a Transformer model to predict the next state token S_{s_t} immediately after each `?`.

Tree-Based Tokenization For path-based search algorithms, we extend the leaf-based trace with the start-of-sequence marker `[BOS]` and the path separator `>`. At each iteration, we encode the full tree-policy path leading to the chosen state. The resulting sequence is as follows:

$$\texttt{[BOS]}\ \texttt{?}\ \pi^{(0)}_{\text{tree}}\ \texttt{\%}\ V_{v_0}\ \texttt{\#}\ S_{s_{0,1}}\ S_{s_{0,2}}\ \ldots\ \texttt{?}\ \pi^{(1)}_{\text{tree}}\ \texttt{\%}\ V_{v_1}\ \texttt{\#}\ S_{s_{1,1}}\ S_{s_{1,2}}\ \ldots\ \texttt{?}\ \ldots$$

where

$$\pi^{(t)}_{\text{tree}} = S_{s_0}\ \texttt{>}\ S_{s_{0,i_1}}\ \texttt{>}\ \cdots\ \texttt{>}\ S_{s_t}$$

denotes the sequence of states selected by the tree policy, whose terminal state S_{s_t} is chosen for expansion at step t. Our constructed model will generate the complete path $\pi^{(t)}_{\text{tree}}$ at each step, conditioned on the previous trace. The Transformer terminates the generation by outputting the `%` token, which also indicates that the immediately preceding state token is the state that should be used for expansion in the next step.

D.2 Trace Format for Transformers Trained from Scratch

To simplify both training and inference of Transformer models trained from scratch, we provide the full set of unvisited states F_{t-1} at step t. Additionally, each unvisited state, denoted as $S_{t-1,1}, S_{t-1,2}, \ldots, S_{t-1,|F_{t-1}|}$, is accompanied by its 0-based index, denoted as $i_1, i_2, \ldots, i_{|F_{t-1}|}$. As a result, the Transformer models should predict these indices when predicting next states. It implies that the task becomes easier than the task of predicting states directly. The example of this trace format is as follows:

$$\cdots\ V_{t-1}\ \underbrace{\texttt{\#}\ i_1\ S_{t-1,1}\ i_2\ S_{t-1,2} \cdots i_{|F_{t-1}|}\ S_{t-1,|F_{t-1}|}\ \texttt{?}\ S_t\ V_t}_{\text{Tokens for a single step}}\ \texttt{\#}\ i_1\ S_{t,1}\ i_2\ S_{t,2} \cdots i_{|F_t|}\ S_{t,|F_t|}\ \cdots$$

where V_t is an estimated value of S_t at step t. In this setting, we discretize V_t to two decimal places and represent it using 101 tokens corresponding to the values $0.00, 0.01, 0.02, ..., 0.99, 1.00$.

```
start_of_iteration 0 r0d0>i0d1 1 r0d0>i1d1 2 r0d0>i2d1
selected_child_and_then_reward 2 0.00 start_of_iteration 0 r0d0>i0d1 1 r0d0>i1d1
2 r0d0>i2d1>i0d2 3 r0d0>i2d1>i1d2 4 r0d0>i2d1>i2d2 selected_child_and_then_reward
0 0.00 start_of_iteration 0 r0d0>i1d1 1 r0d0>i2d1>i0d2 2 r0d0>i2d1>i1d2
3 r0d0>i2d1>i2d2 4 r0d0>i0d1>i0d2 5 r0d0>i0d1>i1d2 6 r0d0>i0d1>i2d2
selected_child_and_then_reward 6 0.00 start_of_iteration 0 r0d0>i1d1
1 r0d0>i2d1>i0d2 2 r0d0>i2d1>i1d2 3 r0d0>i2d1>i2d2 4 r0d0>i0d1>i0d2 5
r0d0>i0d1>i1d2 6 r0d0>i0d1>i2d2>i0d3 7 r0d0>i0d1>i2d2>i1d3 8 r0d0>i0d1>i2d2>i2d3
selected_child_and_then_reward 4 0.40 start_of_iteration 0 r0d0>i1d1
1 r0d0>i2d1>i0d2 2 r0d0>i2d1>i1d2 3 r0d0>i2d1>i2d2 4 r0d0>i0d1>i1d2
5 r0d0>i0d1>i2d2>i0d3 6 r0d0>i0d1>i2d2>i1d3 7 r0d0>i0d1>i2d2>i2d3
8 r0d0>i0d1>i0d2>i0d3 9 r0d0>i0d1>i0d2>i1d3 10 r0d0>i0d1>i0d2>i2d3
selected_child_and_then_reward 9 1.00 start_of_iteration 0 r0d0>i1d1
1 r0d0>i2d1>i0d2 2 r0d0>i2d1>i1d2 3 r0d0>i2d1>i2d2 4 r0d0>i0d1>i1d2
5 r0d0>i0d1>i2d2>i0d3 6 r0d0>i0d1>i2d2>i1d3 7 r0d0>i0d1>i2d2>i2d3 8
r0d0>i0d1>i0d2>i0d3 9 r0d0>i0d1>i0d2>i2d3 selected_child_and_then_reward 9 0.00
```

Figure 6: Trace example of multi-reward tree search problems.

```
start_of_iteration 0 x0y0>x0y1 1 x0y0>x1y0 selected_child_and_then_reward 0 0.00
start_of_iteration 0 x0y0>x1y0 1 x0y0>x0y1>x0y2 2 x0y0>x0y1>x0y0 3 x0y0>x0y1>x1y1
selected_child_and_then_reward 1 0.10 start_of_iteration 0 x0y0>x1y0 1
x0y0>x0y1>x0y0 2 x0y0>x0y1>x1y1 3 x0y0>x0y1>x0y2>x0y3 4 x0y0>x0y1>x0y2>x0y1
selected_child_and_then_reward 3 0.10 start_of_iteration 0 x0y0>x1y0 1
x0y0>x0y1>x0y0 2 x0y0>x0y1>x1y1 3 x0y0>x0y1>x0y2>x0y1 4 x0y0>x0y1>x0y2>x0y3>x0y4
5 x0y0>x0y1>x0y2>x0y3>x0y2 selected_child_and_then_reward 4 0.00
start_of_iteration 0 x0y0>x1y0 1 x0y0>x0y1>x0y0 2 x0y0>x0y1>x1y1 3
x0y0>x0y1>x0y2>x0y1 4 x0y0>x0y1>x0y2>x0y3>x0y2 5 x0y0>x0y1>x0y2>x0y3>x0y4>x0y3
6 x0y0>x0y1>x0y2>x0y3>x0y4>x1y4 selected_child_and_then_reward
4 0.00 start_of_iteration 0 x0y0>x1y0 1 x0y0>x0y1>x0y0 2
x0y0>x0y1>x1y1 3 x0y0>x0y1>x0y2>x0y1 4 x0y0>x0y1>x0y2>x0y3>x0y4>x0y3
5 x0y0>x0y1>x0y2>x0y3>x0y4>x1y4 6 x0y0>x0y1>x0y2>x0y3>x0y2>x0y3 7
x0y0>x0y1>x0y2>x0y3>x0y2>x0y1 selected_child_and_then_reward 3 0.00
```

Figure 7: Trace example of multi-reward navigation problems.

Figures 6 and 7 show the trace examples of multi-reward tree search and navigation problems. These tokens are split by whitespaces, which implies that a single word corresponds to a single token. For the multi-reward navigation problem, to reduce the number of state tokens, we additionally split the state tokens with ">."

E Theoretical Constructions of Transformers

In this section, we provide the details of our theoretical results.

E.1 Theoretical Transformer Model

This section defines a simplified version of the Transformer architecture used in this work. We omit layer normalization and replace the fully connected layer with any arbitrary token-wise function (since MLPs are universal approximators). We also replace the conventional softmax attention mechanism with hard attention.

Transformer Block Let $X \in \mathbb{R}^{d\times n}$ be the input matrix representing a sequence of n token embeddings, each of dimension d. A single-head Transformer *block* consists of a residual self-attention mechanism followed by a position-wise feed-forward function f:

First, the self-attention mechanism updates the input X:

$$\mathrm{Attn}(X) \coloneqq X + VX\ \mathrm{hardmax}\big(X^\top K^\top QX\big) \quad \in \mathbb{R}^{d\times n},$$

Here, $Q, K, V \in \mathbb{R}^{d\times d}$ are the learnable query, key, and value weight matrices, respectively. The term $X^\top K^\top QX$ calculates the $n \times n$ attention weight matrix, where entry (i, j) represents the attention from token j to token i and we define the hardmax operator as the following:

$$\mathrm{hardmax}(z)_i \coloneqq \frac{\mathbb{1}[z_i = \max_k z_k]}{\sum_j \mathbb{1}[z_j = \max_k z_k]}.$$

This operator assigns a uniform probability distribution over the index (or indices) corresponding to the maximum value(s) in z, and zero otherwise. Next, a feed-forward function f is applied independently to each token's representation (each column vector):

$$\mathrm{Block}(X) \coloneqq \mathrm{Attn}(X) + \big[f\big(\mathrm{Attn}(X)_{:,j}\big)\big]_{j=1}^{n} \quad \in \mathbb{R}^{d\times n},$$

where:

- $\mathrm{Attn}(X)_{:,j}$ denotes the j-th column (token representation) after the attention step;
- $f : \mathbb{R}^d \to \mathbb{R}^d$ is any arbitrary piecewise continuous function. The output of f for each token is added residually to the output of the attention layer.

Transformer Network A Transformer network is formed by stacking ℓ such blocks. Given an initial input sequence X and fixed positional embeddings $p_1, p_2, \ldots, p_n \in \mathbb{R}^d$, we first incorporate positional information:

$$X^{(0)} = X + [p_1, p_2, \ldots, p_n].$$

Then, we iteratively apply the blocks:

$$X^{(k)} = \mathrm{Block}\big(X^{(k-1)}\big), \quad \text{for } k = 1, \ldots, \ell.$$

A final linear layer (the "unembedding" matrix) $U \in \mathbb{R}^{v\times d}$ is applied to the final representation of the last token in the sequence, $X^{(\ell)}_{:,n}$, where v is the vocabulary size:

$$\mathrm{logits} = U\, X^{(\ell)}_{:,n} \in \mathbb{R}^v,$$

The predicted next token is the one corresponding to the highest logit value:

$$\text{next token} = \arg\max_i(\mathrm{logits}_i),$$

i.e., we do greedy decoding. We break ties uniformly at random.

Discussion on Model Assumptions The omission of layer normalization follows established practice in theoretical Transformer constructions including universal approximation proofs [44] and constructing programming languages that can be compiled into Transformers [40].

Our hard attention assumption is mild in that it can be approximated arbitrarily closely by softmax attention through logit scaling. It is also a standard assumption used in previous Transformer complexity analyses [26].

Our requirement for feed-forward networks to represent arbitrary piecewise-continuous functions aligns with classical universal approximation theorems. Specifically, ReLU network have been shown to be able to approximate any function in Sobolev spaces [43] with tight bounds on the parameters needed to achieve arbitrary closeness.

Lastly, we note, these assumptions match those in the RASP programming model [40], which similarly removes normalization layers and assumes full expressivity of feed-forward layers.

E.2 Proof of Theorem 1

Embedding Construction We embed each token into a vector whose coordinates ("registers") store interpretable features. Table 1 lists the registers, their meaning, and the initial values for each token type. The $\texttt{id}_i$ register is a collection of registers, each corresponding to one state token S_i. The last three registers (`isVisited`, `inhValue`, and `pos`) are initialized to zero and will be updated by the Transformer. Because, we require a register for each state, it follows that embedding dimension is size $\Theta(TB)$.

Notation We write $e_{\texttt{a}}$ for the standard basis vector with a 1 in the coordinate corresponding to register `a` and 0 elsewhere. For registers `a` and `b`, define

$$M_{\texttt{a}\to\texttt{b}} = e_{\texttt{b}}\, e_{\texttt{a}}^{\top},$$

which is the matrix that copies the value from register `a` into register `b`.

Table 1: Initial embeddings. Each row is a register (coordinate), and each column gives its initial value for the token types V_α (value), "`# ? %`" separators, and state tokens S_k.

Register	V_α	#	?	%	S_k
`value`	α	0	0	0	0
`#?`	0	$+1$	-1	0	0
`isValue`	1	0	0	0	0
`is#?`	0	1	1	0	0
`isState`	0	0	0	1	0
$\texttt{id}_i$	0	0	0	0	$\mathbb{1}[i=k]$
`bias`	1	1	1	1	1
Dynamically updated registers:					
`isVisited`	0	0	0	0	0
`inhValue`	0	0	0	0	0
`pos`	0	0	0	0	0

We also add a positional embedding by setting

$$X_t^{(0)} \leftarrow X_t + t\, e_{\texttt{pos}},$$

so that the `pos` register of the token at position t stores its index in the trace.

Layer 1: Marking Visited States We want each state token to know whether it follows a `?` (i.e., was selected by the Transformer for selection) or if it was generated by the environment to indicate the children states.

Choose

$$Q = M_{\texttt{bias}\to 1}, \quad K = M_{\texttt{is\#?}\to 1}, \quad V = M_{\texttt{\#?}\to\texttt{isVisited}}.$$

Here `bias` queries uniformly, K gives score 1 to separators (tokens with `is#?=1`), and V copies the separator's `#?` ($+1$ for `#`, -1 for `?`) into the `isVisited` register of the querying token. Since a state

immediately after ? sees one more -1 than $+1$, its isVisited becomes negative; after # it sees equal counts and remains zero. Thus $\texttt{isVisited} < 0$ exactly for visited states.

We then apply a feed-forward that scales the isValue register of value tokens by their index in the trace:

$$f^{(1)}(y) = \left(y_{\texttt{isValue}}\, y_{\texttt{pos}}\right) e_{\texttt{isValue}},$$

which will be used in subsequent layers state tokens attend to only the closest preceding value token.

Layer 2: Propagating Inherited Rewards Next we want each *frontier* state (states that are children of visited states but are themselves not yet visited) to inherit the rollout estimate value of its parent state. Parent values live in the most recent value token V_α before the state's position.

Set

$$Q = M_{\texttt{bias}\to 1}, \quad K = M_{\texttt{isValue}\to 1}, \quad V = M_{\texttt{value}\to\texttt{inhValue}}.$$

Each state token attends (via hard attention) to the immediately preceding value token (due to the feedforward in the previous layer), and the V matrix writes that value α into its inhValue register.

We then use a piecewise feed-forward on the $\texttt{id}_i$ registers:

$$f^{(2)}(y)_{\texttt{id}_i} = \begin{cases} y_{\texttt{inhValue}}, & \text{if,} y_{\texttt{isVisited}} = 0 \\ -1, & \text{otherwise,} \end{cases}$$

with all other coordinates set to zero. Non-state tokens and visited state tokens get a negative score, and each frontier state S_i has its inherited reward stored in $\texttt{id}_i$.

Layer 3: Selecting the Maximum Finally, we collect all frontier scores and choose the largest:

$$Q = M_{\texttt{bias}\to 1}, \quad K = M_{\texttt{isState}\to 1}, \quad V = \sum_i M_{\texttt{id}_i\to\texttt{id}_i}.$$

At embedding to corresponding to the last ? in the trace, the query attends to all preceding state tokens (isState=1) and V sums each state S_i's $\texttt{id}_i$ value into the querying token's $\texttt{id}_i$. We set $f^{(3)}(y) = 0$ as we do not need the feedforward in this layer. Now assuming tokens are indexed so that the embedding coordinate $\texttt{id}_i$ corresponds exactly to the vocabulary index of state token S_i, we choose the unembedding matrix U to be zero everywhere except on the submatrix mapping the $\texttt{id}_i$ coordinates to their corresponding token logits, where it is the identity. In this setup, the $\texttt{id}_i$ registers directly become the logits for each state token, so greedy decoding picks the state with the highest inherited reward.

Remark (Uniform-Leaf Sampling) If instead all V_α embeddings are set to the same constant vector, then every frontier state receives an identical score in Layer 2, yielding uniform sampling over leaves by the same construction.

E.3 Proof of Theorem 2

For this theorem, we analyze the conventional MCTS [34], which is a widely-used form, where the selected state s^* is determined by taking the $\arg\min$ over $N(s)$, rather than $N^*(s)$ as in Algorithm 6. In our empirical experiments, we used $N^*(s)$ due to the shallow nature of the search trees. However, for theoretical clarity and consistency with classical MCTS, the Transformer construction presented here uses $N(s)$.

Embedding Construction Tokens are again embedded into interpretable "registers." Table 2 defines the initial static register values; all dynamic registers are initialized to zero. Registers of type $\texttt{X}_i$ denote a set $\{\texttt{X}_i\}_{i=1}^{TB+1}$, with one register per state token. The $\texttt{oid}_i$ registers include two additional coordinates used specifically for the > and % tokens.

As in the previous construction, we apply a positional embedding via $X_t^{(0)} \leftarrow X_t^{(0)} + t\, e_{\texttt{pos}}$, so that each token's pos register holds its position in the trace.

For any coordinates where we do not explicitly how it is updated by a feedforward function, the output is defined to be zero.

Table 2: Initial embeddings for the MCTS-Transformer: static registers encode each token's fixed features, while dynamic registers are initialized to zero and updated by the network conditioned on the trace.

Register	V_α	#	?	>	[bos]	%	S_k
value	α	0	0	0	0	0	0
#?	0	+1	−1	0	0	0	0
isValue	1	0	0	0	0	0	0
is#?	0	1	1	0	0	0	0
is>	0	0	0	1	0	0	0
is?	0	0	1	0	0	0	0
isBos	0	0	0	0	1	0	0
is#Bos	0	1	0	0	1	0	0
isState	0	0	0	0	0	0	1
$\texttt{id}_i$	0	0	0	0	0	0	$\mathbb{1}[i = k]$
bias	1	1	1	1	1	1	1
Dynamically updated registers:							
is? · pos	0	0	0	0	0	0	0
isValue · pos	0	0	0	0	0	0	0
isState · pos	0	0	0	0	0	0	0
closest?pos	0	0	0	0	0	0	0
parentpos	0	0	0	0	0	0	0
isVisited	0	0	0	0	0	0	0
wasVisited	0	0	0	0	0	0	0
$\texttt{vid}_i$	0	0	0	0	0	0	0
$\texttt{cid}_i$	0	0	0	0	0	0	0
$\texttt{pid}_i$	0	0	0	0	0	0	0
$\texttt{psid}_i$	0	0	0	0	0	0	0
$\texttt{nsid}_i$	0	0	0	0	0	0	0
$\texttt{oid}_i$	0	0	0	0	0	0	0
iter	0	0	0	0	0	0	0
pos	0	0	0	0	0	0	0

Layer 1: Iteration Counter and Positional Precomputation This layer precomputes positional dependent registers. The attention mechanism is defined by the following:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{is\#bos}\to 1}, \\ V &= M_{\texttt{isBos}\to\texttt{iter}}. \end{aligned}$$

This attention is used to compute the number of iterations that have passed which will be stored in the iter register. We apply the feed-forward function $f^{(1)}$:

$$\begin{aligned} f^{(1)}(y)_{\texttt{is?p}} &= y_{\texttt{is?}} y_{\texttt{pos}}, \\ f^{(1)}(y)_{\texttt{isState·pos}} &= y_{\texttt{isState}} y_{\texttt{pos}}, \\ f^{(1)}(y)_{\texttt{isValue·pos}} &= y_{\texttt{isValue}} y_{\texttt{pos}}, \\ f^{(1)}(y)_{\texttt{iter}} &= (1/y_{\texttt{iter}}) - y_{\texttt{iter}}, \quad \text{if } y_{\texttt{iter}} > 0 \text{ else } 0, \end{aligned}$$

which processes the iter register to hold the correct value and preproccess several other positionally dependent registers for later computation.

Layer 2: Compute closest?pos This layer will do computation so that when isState is 1, closest?pos will store the position of the closest preceding ? token in the trace. Otherwise, it is set to 0. The attention matrices are given by the following:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{is?·pos}\to 1}, \\ V &= M_{\texttt{pos}\to\texttt{closest?pos}}. \end{aligned}$$

Through the hard attention, each token attends to the preceding token with the largest $\texttt{is?} \cdot \texttt{pos}$ value (i.e., the value of the closest preceding `?` token). Its `pos` value is copied to the querying token's `closest?pos` register. The subsequent feed-forward function $f^{(2)}$ is:

$$f^{(2)}(y)_{\texttt{closest?pos}} = y_{\texttt{closest?pos}}\, y_{\texttt{isState}} - y_{\texttt{closest?pos}},$$

which zeros out `closest?pos` if the current token does not correspond to a state.

Layer 3: Compute Per-Iteration Reward and Count Statistics This layer computes the reward and tree policy visitation count statistics for each iteration, which will be stored in the respective V_α token at the end of its iteration. The attention is:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{closest?pos}\to 1}, \\ V &= \sum_i (M_{\texttt{id}_i \to \texttt{vid}_i} + M_{\texttt{id}_i \to \texttt{cid}_i}). \end{aligned}$$

For a V_α token, this attention sums statistics from state tokens s_k visited by the tree policy within the current iteration, identified as the tokens with the largest value in `closest?P`. The `id_i` values from these s_k tokens are copied and summed into the V_α token's `vid_i` and `cid_i` registers. The feed-forward function $f^{(3)}$, applied at V_α tokens, then sets:

$$\begin{aligned} f^{(3)}(y)_{\texttt{vid}_i} &= y_{\texttt{value}}\, \mathbb{1}[y_{\texttt{vid}_i} > 0] - y_{\texttt{vid}_i}, \\ f^{(3)}(y)_{\texttt{cid}_i} &= \mathbb{1}[y_{\texttt{cid}_i} > 0] - y_{\texttt{vid}_i}, \end{aligned}$$

storing 1 in $\texttt{cid}_i$ when state S_i is visited in this iteration (used to compute visitation counts).

Layer 4: Aggregate Statistics (Backpropagation) This layer aggregates reward and count statistics from all iterations. These accumulated statistics will be stored in the final V_α token of the trace. We set:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{isValue}\to 1}, \\ V &= \sum_i (M_{\texttt{vid}_i \to \texttt{vid}_i} + M_{\texttt{cid}_i \to \texttt{cid}_i}). \end{aligned}$$

The token intended for final aggregation attends to all previous V_α tokens (where $\texttt{isValue} = 1$). It sums up the $\texttt{vid}_i$ (values) and $\texttt{cid}_i$ (counts) from these iteration-specific V_α tokens into its own registers. We then use the feedforward layer to unormalize by the number of iterations (since the attention will average them):

$$\begin{aligned} f^{(5)}(y)_{\texttt{cid}_i} &= y_{\texttt{cid}_i}\, y_{\texttt{iter}} - y_{\texttt{cid}_i} \\ f^{(5)}(y)_{\texttt{vid}_i} &= y_{\texttt{vid}_i}\, y_{\texttt{iter}} - y_{\texttt{vid}_i}. \end{aligned}$$

Layer 5: Identify State Tokens Selected by Transformer This layer identifies which state tokens in the trace corresponds to those generated by the Transformer during the tree policy trace. We use attention:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{is?\#}\to 1}, \\ V &= M_{\texttt{?\#}\to \texttt{isSelected}}. \end{aligned}$$

Each state token sums the `?#` values from preceding relevant tokens. For states immediately following a `#`, this sum in `isSelected` will be 0. It will be nonzero otherwise. The feed-forward $f^{(5)}$ uses this fact to compute the register:

$$f^{(5)}(y)_{\texttt{isSelected}} = y_{\texttt{isState}} \cdot \mathbb{1}[y_{\texttt{isSelected}} \neq 0] - y_{\texttt{isSelected}}.$$

This sets `isSelected` to 1 for state tokens selected by the Transformer and 0 for the state token generated by the environment to indicate the child nodes.

Layer 6: Identify Parent Position for Neighbors This layer serves as preprocessing to identify the parent node of each neighbor token (i.e., those state tokens immediately succeeding the # token), by first storing the parent's position in the `parentpos` register of the parent token itself. No attention is needed, so we set all the attention weights to 0. The feed-forward $f^{(6)}$ is:

$$f^{(6)}(y)_{\texttt{parentP}} = y_{\texttt{isState}} \cdot y_{\texttt{isSelected}} \cdot y_{\texttt{pos}}.$$

This operation stores the current token's position into its own `parentP` register if it is a state token and corresponds to a state visited by the tree policy in some iteration.

Layer 7: Propagate Parent `id` to Its Children States Children state tokens use the information about the parent's position to retrieve the parent's `id`. The attention is:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{parentpos}\to 1}, \\ V &= \sum_i M_{\texttt{id}_i \to \texttt{pid}_i}. \end{aligned}$$

Each neighbor state token attends to the immediately preceding non-neighbor state (which will be it's parent state). It then copies the parent token's $\texttt{id}_i$ into its own $\texttt{pid}_i$ registers. No feed-forward $f^{(7)}$ is needed, so we set it to 0.

Layer 8: Store Previously Selected State `id` in Embedding of > Token The > token stores the `id` of the state that was just selected by the tree policy. We set:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{isState}\cdot\texttt{pos}\to 1}, \\ V &= \sum_i M_{\texttt{id}_i \to \texttt{psid}_i}. \end{aligned}$$

The > token attends to the immediately preceding state token and copies that state's $\texttt{id}_i$ into its own $\texttt{psid}_i$.

Layer 9: Collect Children States of Previously Selected State in Embedding of > Token The > token collects the `id` of all children of the previously selected state. The attention is:

$$\begin{aligned} Q &= \sum_i M_{\texttt{psid}_i \to i}, \\ K &= \sum_i M_{\texttt{pid}_i \to i}, \\ V &= \sum_i M_{\texttt{id}_i \to \texttt{nsid}_i}. \end{aligned}$$

The > token uses its $\texttt{psid}_i$ registers as the query. State tokens use their $\texttt{pid}_i$ registers as keys. If key and query match, the neighbor's $\texttt{id}_i$ is copied/summed into the > token's $\texttt{nsid}_i$ register. No feed-forward $f^{(9)}$ is needed, so we set it to 0.

Layer 10: Final UCT Computation for > Token At the > token, the next state to select is calculated using UCT scores. First, it gathers aggregated statistics using the following attention weights:

$$\begin{aligned} Q &= M_{\texttt{bias}\to 1}, \\ K &= M_{\texttt{isValue}\cdot\texttt{pos}\to 1}, \\ V &= \sum_i (M_{\texttt{cid}_i \to \texttt{cid}_i} + M_{\texttt{vid}_i \to \texttt{vid}_i}). \end{aligned}$$

The > token attends to the token holding globally aggregated statistics (computed in layer 4) into its own registers. The feed-forward function $f^{(10)}$ then does the UCT computation:

$$f^{(10)}(y)_{\texttt{oid}_i} = \begin{cases} y_{\texttt{vid}_i} + c\sqrt{\log(2\sum_k y_{\texttt{cid}_k}\, y_{\texttt{psid}_k})/y_{\texttt{cid}_i}} & \text{if } y_{\texttt{is>}} = 1,\ y_{\texttt{nsid}_i} = 1 \text{ and } y_{\texttt{cid}_i} \neq 0, \\ \infty, & \text{if } y_{\texttt{is>}} = 1,\ y_{\texttt{nsid}_i} = 1 \text{ and } y_{\texttt{cid}_i} = 0, \\ 0, & \text{otherwise.} \end{cases}$$

Layer 11: Generate Start State S_0 After ? This layer ensures the Transformer generates S_0 (the root node) when it sees the initial ? token. The attention matrices are

$$\begin{aligned} Q &= 0, \\ K &= 0, \\ V &= M_{\texttt{is?} \to \texttt{oid}_0}. \end{aligned}$$

We set $f^{(11)}$ to the 0 function in this layer.

Layer 12: Generate > or % After State Selection This layer determines whether to generate > or % after a state token has been chosen by the tree policy. Specifically > is generated if the tree policy has not reached a new undiscovered node. The % token is generated otherwise. For this layer, we set the attention matrices to the following:

$$\begin{aligned} Q &= \sum_i M_{\texttt{id}_i \to i}, \\ K &= \sum_i M_{\texttt{id}_i \to i}, \\ V &= M_{\texttt{isSelected} \to \texttt{wasSelected}}. \end{aligned}$$

Intuitively for the current state token in the tree policy path, it will attend to all previous state tokens that represent the same states. It aggregates the `isSelected` values into the `wasSelected` register. Thus `wasSelected` > 0 if and only if the state was previously visited in another iteration. Thus we let the feed forward layer be

$$\begin{aligned} f^{(12)}(y)_{\texttt{oid}_>} &= y_{\texttt{isState}} \mathbb{1}[y_{\texttt{wasVisited}} > 0], \\ f^{(12)}(y)_{\texttt{oid}_\%} &= y_{\texttt{isState}} \mathbb{1}[y_{\texttt{wasVisited}} = 0]. \end{aligned}$$

Final Unembedding Assuming without loss of generality that $\texttt{oid}_i$ corresponds to the i'th state token, it again suffices to let U be the matrix that is zero everywhere except on the submatrix mapping the $\texttt{oid}_i$ to the corresponding token. The token with the highest logit value is chosen by greedy decoding.

Note on on Different Tree Policies For path sampling with a greedy policy and uniform path sampling, Layer 10 is modified. For path sampling with a greedy policy, the feed-forward $f^{(10)}$ at the > token computes:

$$f^{(10)}(y)_{\texttt{oid}_i} = y_{\texttt{vid}_i}\, y_{\texttt{is>}}\, y_{\texttt{nsid}_i}.$$

For uniform path sampling, $f^{(10)}$ would set:

$$f^{(10)}(y)_{\texttt{oid}_i} = y_{\texttt{is>}}\, y_{\texttt{nsid}_i}.$$

F Details of Empirical Analysis on Transformers' Search Abilities

In this work, we use NumPy [10], SciPy [39], PyTorch [24], Scikit-learn [25], and other scientific packages in Python. Moreover, as shown in the main article, we employ a variety of LLM APIs such as OpenAI GPT, Google Gemini, and Alibaba Qwen.

To run our experiments, we utilize commercial Intel and AMD CPUs such as AMD EPYC 9374F and Intel Xeon Platinum 8352Y, and NVIDIA GPUs such as NVIDIA L40S.

F.1 Evaluation Metrics

We use the following evaluation metrics to analyze search algorithms:

- Max-reward hit rate: The fraction of runs that ever attain the global-best reward $r(s_{\text{best}})$ calculates the probability of achieving $r(s_{\text{best}})$ over multiple runs (i.e., the hit rate): $N_{\text{hit}}/N_{\text{total}}$, where N_{hit} and N_{total} are the number of runs that hit s_{best} and the total number of runs, respectively;
- Discounted cumulative gain: This metric, based on discounted cumulative gain [11], quantifies how quickly a search algorithm finds s_{best} in each run: $(1/N_{\text{total}}) \sum_{k=1}^{N_{\text{total}}} 1/\log_2(i_k + 1)$, where i_k is the 1-indexed max-reward hit iteration index at run k. If a particular run fails, $i_k = \infty$;
- Normalized path length: This metric regarding the shortest path to s_{best} computes a metric value of $(1/N_{\text{total}}) \sum_{k=1}^{N_{\text{total}}} \exp(\ell_{k,\text{truth}} - \ell_k)$, where $\ell_{k,\text{truth}}$ is the ground-truth shortest path length to s_{best} and ℓ_k is the shortest path length predicted by the Transformer model. If it fails to find s_{best}, $\ell_k = \infty$, which makes $\exp(\ell_{k,\text{truth}} - \ell_k)$ zero;
- Highest/cumulative reward: A highest reward is calculated by $(1/N_{\text{total}}) \sum_{k=1}^{N_{\text{total}}} \max(\mathcal{R}_k)$ and a cumulative reward is calculated by $(1/N_{\text{total}}) \sum_{k=1}^{N_{\text{total}}} \sum_{r \in \mathcal{R}_k} r$, where $\mathcal{R}_k$ is the rewards achieved at run k. Compared to the max-reward hit rate, these account for both suboptimal and global-best rewards.
- Normalized jump distance: This is the arithmetic mean of jump distances where a jump distance is defined by $(\text{jump}(s_{t-1}, s_t) + \text{jump}(s_t, s_{t+1}))/2$. Note that $\text{jump}(s_i, s_j)$ indicates the shortest distance between s_i and s_j on a tree-structured space.

These metrics are selected to evaluate search algorithms across criteria such as efficiency, robustness, and solution quality. Note that higher values indicate better performance across all metrics except for normalized jump distance.

F.2 Experimental Details of Transformers Trained from Scratch

The architecture of Llama models [35, 18] is adopted to our Transformer models trained from scratch. Unless otherwise specified, the Transformer models are defined with 8 blocks, 8 heads, and 512 hidden dimensions. The number of intermediate dimensions is set as 1024. Minimum and maximum learning rates are 5×10^{-5} and 5×10^{-4}. It uses a learning rate scheduling technique with a warm-up. The AdamW optimizer [19] with $\beta_1 = 0.9$ and $\beta_2 = 0.99$ is utilized, and the rotary positional embedding [32] with $\theta = 10{,}000$ is used.

As offline training and validation datasets, we generate 200 different problem instances and 100 traces per instance, for a total of 20,000 traces. These datasets are split to 70% of training datasets and 30% of validation datasets. On the other hand, we report online test performance for all experiments. These test experiments use 10 different problem instances and 100 traces per instance, for a total of 1,000 traces. These results with 1,000 traces are used to calculate the mean and standard deviation of each experiment set. We verify that training and test problem instances are mutually exclusive. Moreover, we plot 1.96 standard errors for all figures.

To calculate the KL divergence, we should obtain the next-state probabilities of reference algorithms. Uniform leaf sampling assigns equal probabilities to all possible next states, and uniform path sampling uses depth-level equal probabilities to calculate the probabilities of possible next states. Greedy and policy-guided path sampling follow their respective tree traversal rules, while also considering all tied states when selecting among equally preferred options. As a result, we empirically estimate the probabilities of the reference algorithms by repeating the sampling process 100 times with the same trace, varying only the random seed.

G Additional Empirical Results on Transformers' Behavior Cloning Abilities

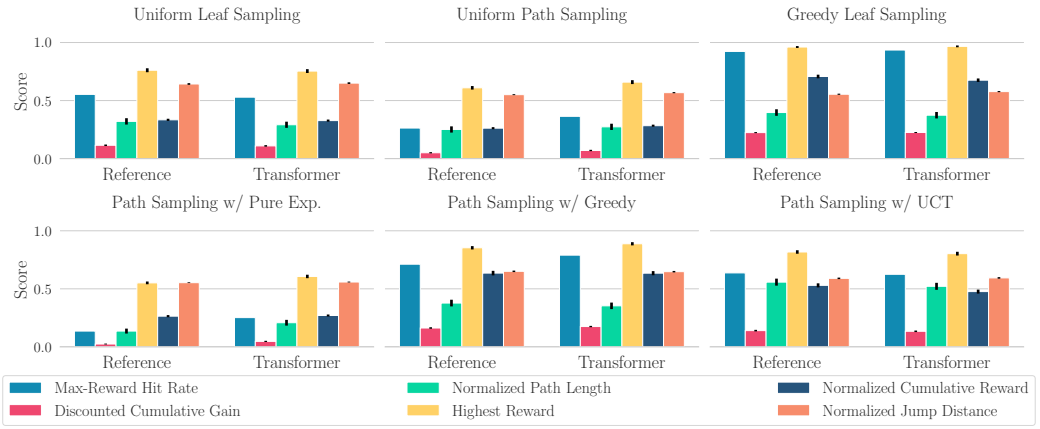


Figure 8: Behavior cloning results on the multi-reward navigation problem, where the size of each problem is 4×4, the wall density of each problem is 0.4, and a search step budget is 50.

In addition to Figure 1, Figure 8 demonstrates behavior cloning results on the multi-reward navigation problem. It shows that the Transformers trained from scratch successfully imitate our search trajectories by viewing their performance in terms of diverse evaluation metrics.

To understand the behavior cloning abilities of Transformers, we also test whether the algorithm mimics the underlying behavior, i.e., conditioned on the same trajectory, will the Transformer select the same state for expansion as the reference algorithm? To answer this question, we measure the KL divergence between the distribution over the next selected state of the reference algorithm and the behavior-cloned Transformer. This KL divergence is computed online by following the traces of the reference algorithms and comparing the predicted next-state probabilities generated by the reference strategies and the Transformer models. Eventually, these probabilities are used for calculating the KL divergence: $\sum p(x) \log(p(x)/q(x))$, where $p(x)$ and $q(x)$ is the next-token probabilities over a token x of a reference algorithm and Transformer model, respectively. Note that the next-state probabilities of the reference algorithms reflect uniform tie-breaking among equally preferred nodes; see Section F.2 for the details of how these probabilities are calculated.

As shown in Figure 3, the leaf-based sampling algorithms achieve low KL divergence with their reference counterparts, suggesting successful behavior alignment. In contrast, for path-based sampling algorithms, they exhibit significantly higher divergence. This suggests that they are unlikely to replicate the reference algorithms' behavior despite matching the performance on all of our metrics. This discrepancy arises because the training traces for path-based methods do not explicitly encode the tree-policy path for each step. Consequently, the Transformer must infer the tree-policy and select the next state in a single step, without intermediate outputs. This aligns with our theoretical analysis, where the trace format as described in Section D.1 explicitly requires the Transformer to output the tree-policy before selecting the next state.

H Additional Empirical Results on Multi-Reward Tree Search

In this section, we show additional empirical results on multi-reward tree search problems.

H.1 Additional Results on More Diverse Multi-Reward Tree Search Problems

In addition to the experiments shown in Figure 1, we present additional results on multi-reward tree search problems with more diverse settings.

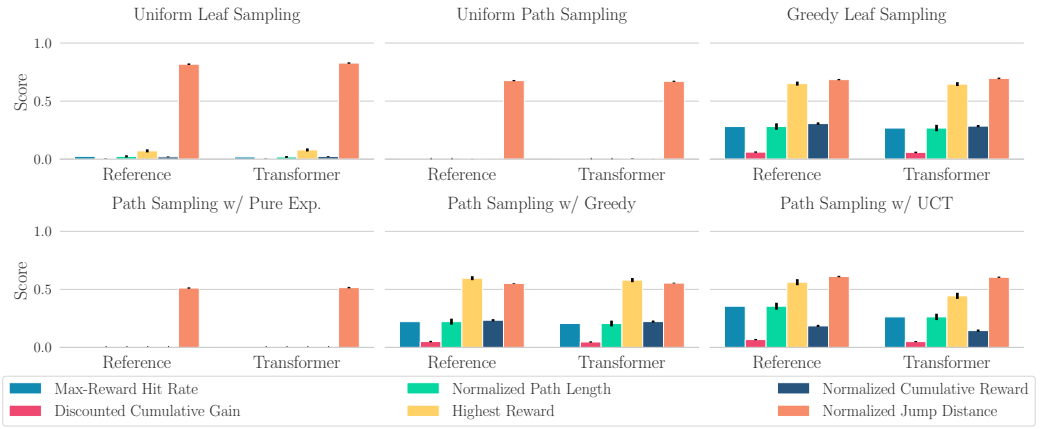


Figure 9: Behavior cloning results on the multi-reward tree search problem, where each binary tree of depth 8 has 8 different goals and a search step budget is 50.

Figure 9 shows the experimental results that test deeper multi-reward trees, which are depth 8. These results follow our observation that Transformers successfully mimic the behaviors of reference search algorithms. Since this setting is harder than the setting of Figure 1, the search performance of each algorithm is relatively poor.

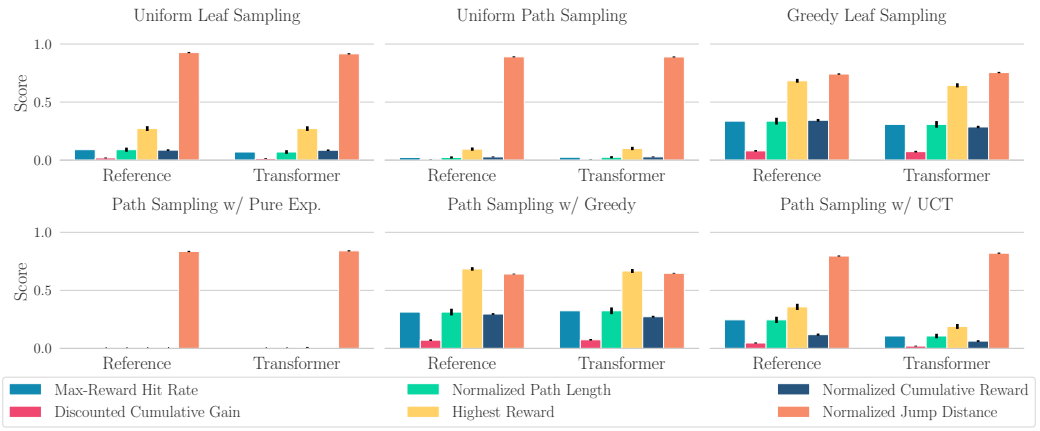


Figure 10: Behavior cloning results on the multi-reward tree search problem, where each quaternary tree of depth 4 has 8 different goals and a search step budget is 50.

Similarly, we conduct experiments on wider multi-reward trees in Figure 10. These trees have 4 child states for each parent state. Transformers successfully show similar performance of the respective search strategies, following our expectation.

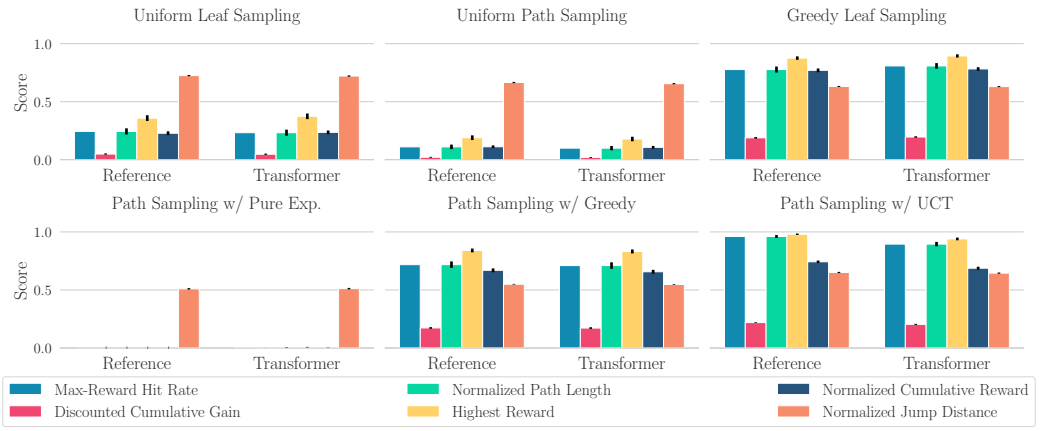


Figure 11: Behavior cloning results on the multi-reward tree search problem, where each binary tree of depth 6 has 4 different goals and a search step budget is 50.

We test a problem with a fewer number of goal sates. As shown in Figure 11, these results follow the findings observed in the previous experiments.