IMPROVING REACHABILITY ON REASONING PUZZLES

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent work has observed the accuracy collapse of large language and reasoning models on moderately large puzzles. Apriori, it is unclear if this failure is due to the inability to find an algorithm or the inability to execute a given algorithm faithfully or simply due to the large number of execution steps. Many puzzles such as Tower of Hanoi are essentially graph reachability tasks, whose graph size increases exponentially. However, symmetry and structure of these graphs allow simple low-complexity algorithms to generate the solution, compared to general-purpose algorithms for reachability that require more compositions, recursion, memory. We theoretically show that faithful execution of certain low-complexity algorithms for reachability are easier to express for small transformer models, compared to general-purpose algorithms for reachability. We empirically observe that explicitly providing such algorithmic hints to language and reasoning models leads to substantially improved performance and pushes the frontier of their accuracy collapse.

1 INTRODUCTION

Recent years have witnessed the emergence of a new class of models, termed Large Reasoning Models (LRMs), which are explicitly designed to handle complex reasoning tasks (OpenAI, 2024; Guo et al., 2025a; Anthropic, 2025). These models typically employ Chain-of-Thought (CoT) reasoning, enabling them to explore multiple intermediate reasoning paths before producing a final answer. Despite achieving state-of-the-art performance on a wide range of reasoning benchmarks, recent evidence suggests that LRMs struggle with long-horizon reasoning, including comparatively simple reasoning puzzles whose solutions require coherent answers with a large number of steps (Shojaee* et al., 2025).

In this work, we study the algorithmic structure underlying reasoning tasks and argue that the algorithm, rather than task complexity that is believed to be the limiting factor, plays a decisive role in whether reasoning models succeed or fail. We focus on puzzles and reachability-style problems whose state spaces grow exponentially with input size, such as the Tower of Hanoi, river crossing puzzles, and checker-jumping games. These problems admit multiple correct solution strategies: some rely on recursive or sequential reasoning, while others admit explicit, closed-form descriptions that compute the desired outcome directly from the input. We naturally use results on transformer expressibility as the lens through which we differentiate various solution strategies.

1.1 RELATED WORKS

There has been substantial effort to analyze the expressive power of transformer architectures through a classical theoretical lens (Strobl et al., 2024). Recent works have shown that augmenting transformers with additional computational resources such as Chain-of-Thought Merrill & Sabharwal (2024), padding tokens Merrill & Sabharwal (2025a), and increased depth Merrill & Sabharwal (2025b) can significantly boost their expressive power.

Additionally, numerous works have focused on classifying transformer architectures in terms of classical complexity classes, with Strobl (2023) showing that with $O(\log n)$ bits of floating-point precision, average-hard attention transformers (AHATs) can be approximated in L-uniform TC^0. Later, Merrill & Sabharwal (2025c) showed that softmax-attention transformers (SMATs) can be approximated in DLOGTIME-uniform TC^0. These results were further improved by Chiang (2025),

who showed that SMATs with $O(\text{poly}(n))$ bits of floating-point precision, and AHATs with no approximations, are in DLOGTIME-uniform TC^0.

While expressibility is important from a theoretical standpoint, learnability is crucial for real-world applications. Many studies have attempted to bridge this gap withYang et al. (2025) showing that a 2-layered decoder only transformer not only expresses but also learns tree reachability. Ye et al. (2025) show that the learned strategy for graph reachability hinges on whether most training instances are within the models capacity or not.

Shojaee* et al. (2025) Schnabel et al. (2025) introduce the bounded attention prefix oracle (BAPO) model, a new computational framework that models bandwidth constraints on attention heads, the mechanism for internal communication in LLMs to show that several important reasoning problems like graph reachability require high communication bandwidth for BAPOs to solve.

1.2 Our Contributions

We view reasoning on puzzle graphs through the lens of reachability in the associated configuration graph, and observe that many such graphs exhibit a structure that can be exploited to have *parallelizable* algorithms (more formally, TC^0 circuits). Under this perspective, we focus on the Tower of Hanoi, a canonical puzzle that has been shown to be challenging for large reasoning models (LRMs) in recent work.

1. We theoretically prove that a simple 1-layer, 2-head transformer can express the solution for Tower of Hanoi 3.1 when given the configuration graph as input. In addition, we also compile a transformer object using RASP that realizes the closed form TC^0 solution.
2. Guided by insights on transformer expressibility through TC^0 computation, we perform experiments on LRMs. Specifically, for Tower of Hanoi, we showcase an increase in accuracy using two techniques (3): (i) passing the configuration graph to the LRM, (ii) using the non-sequential TC^0. A detailed account of the experiments can be found in section 4.

2 Preliminaries

2.1 Relevant Complexity Classes

A Boolean circuit is a directed acyclic graph whose internal nodes are gates and whose leaves are input variables or constants. The size of a circuit is the number of gates, and the depth is the length of the longest path from an input to the output. A circuit family $\{C_n\}_{n\in\mathbb{N}}$ is *uniform* if there exists a logspace Turing machine that, on input 1^n, outputs a description of C_n. We define here all the complexity classes referred to in this paper.

1. AC^0 consists of all Boolean functions computable by families of polynomial-size, depth-$O(1)$ circuits with unbounded fan-in $\{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ gates. Negations are allowed only at the inputs.
2. TC^0 is defined similarly, except that circuits may also use unbounded fan-in *threshold gates*. A threshold gate outputs 1 if the weighted sum of its inputs exceeds a given threshold.
3. NC^1 consists of all Boolean functions computable by families of polynomial-size, depth-$O(\log n)$ circuits with $\{\mathsf{AND}, \mathsf{OR}, \mathsf{NOT}\}$ gates with fan-in 2. Negations are allowed only at the inputs.

It is known that $\mathsf{AC}^0 \subsetneq \mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{L}$. Intuitively, L captures problems that can be solved while storing only a logarithmic number of bits of information about the input. This amount of memory is sufficient to maintain a constant number of pointers or counters into the input, but not enough to store large intermediate results.

2.2 Theoretical Limitations of Transformers

Chiang (2025) showed that softmax-attention transformers (SMATs) with $O(\text{poly}(n))$ bits of floating-point precision, and average-hard attention transformers (AHATs) with no approximations are in

DLOGTIME-Uniform TC^0, improving over the previous results from Strobl (2023) and Merrill & Sabharwal (2025c) who respectively showed that with $O(\log n)$ bits of floating-point precision AHATs can be approximated in L-uniform TC^0, and SMATs can be approximated in DLOGTIME-uniform TC^0.

SMATs are shown to exist between AC^0 and TC^0, i.e.:

$$\mathsf{NC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{SMAT} \subseteq \mathsf{TC}^0 \subseteq \mathsf{NC}^1.$$

2.3 Graph Reachability

Given a graph $G = (V, E)$ and vertices $s, t \in V$, the *reachability problem* asks whether there exists a path from s to t in G.

Reachability in Special Graphs The computational difficulty of reachability depends crucially on the structure of the underlying graph. In general directed graphs, the reachability problem is complete for nondeterministic logspace (NL Jones (1975). However, imposing structural constraints on the graph can dramatically reduce complexity. A classic example would be that the reachability of the undirected graph is L completeReingold (2005), which is spatially more efficient (under the conjecture that NL $\neq$ L). Allender et al. (2006) shows that reachability on a special case of grid graphs is TC^0 complete and we know that $\mathsf{TC}^0 \subsetneq \mathsf{NC}^1 \subseteq \mathsf{L}$ making these much easier to solve.

Tower of Hanoi as reachability problem

2.4 Restricted Access Sequence Processing (RASP)

Restricted Access Sequence Processing (RASP) Weiss et al. (2021) is a small domain-specific language (DSL) designed to describe computations that are *native* to Transformer architectures. Rather than providing general-purpose control flow, RASP restricts the programmer to a handful of primitives that mirror common Transformer building blocks: elementwise transformations over token-wise features, comparisons that induce attention-style relations, and aggregation operators that combine information across sequence positions. As a result, a RASP program can often be viewed as a high-level, architecture-aligned specification of what a Transformer *could* compute.

Programs written in RASP can be compiled by tools such as Tracr Lindner et al. (2023) into concrete Transformer objects whose forward pass implements the program's logic. Consequently, exhibiting a RASP program for a task serves as an existence proof: there exists a Transformer (possibly large and carefully constructed) that solves the task. Since expressing programs in RASP is a sufficient but not necessary requirement, RASP $\subset$ SMAT.

3 Reachability Problems with and without Sequential Structure

3.1 Solving Puzzles is Solving Reachability

Many combinatorial puzzles, such as the Tower of Hanoi, can be modeled as reachability problems in an implicitly defined graph, often called a configuration graph or a state graph. For example, the Tower of Hanoi puzzle can be formalized as a reachability problem on a directed graph $G_n = (V_n, E_n)$. The vertex set V_n consists of all legal configurations of n disks distributed among a fixed set of towers. There is a directed edge $(u, v) \in E_n$ if and only if configuration v can be obtained from u by a single legal move of one disk. Given a designated start configuration $s \in V_n$ and a target configuration $t \in V_n$, the Tower of Hanoi problem asks whether t is reachable from s in G_n. Other puzzle graphs such as River Crossing, Checker Jumping etc as well translate naturally in graph problems.

As we have discussed in the preliminaries solving reachability in the directed graph is NL complete, but special structural properties of the graph allow for easier algorithms. For our puzzle graphs, where usually the configuration graph is exponential in the size of the input, graph reachability would appear very hard to solve. In fact, Hearn & Demaine (2005) actually shows that some puzzle problems, such as the sliding block puzzle, are PSPACE-complete, making them very hard to compute. However,

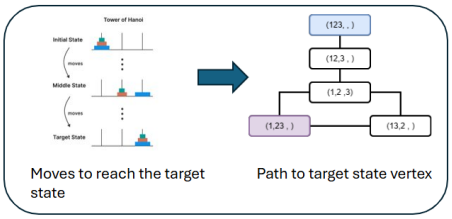

Figure 1: Reformulation of Tower of Hanoi

unlike general puzzle reachability problems captured by the above problem, although $|V_n|$ for Tower of Hanoi grows exponentially with n, the graph G_n has a highly regular and recursive structure, making it a canonical example of reachability in an implicitly defined and structurally constrained graph.

3.2 Tower of Hanoi : Recursive vs TC^0 algorithm

The classical recursive algorithm for solving the Tower of Hanoi produces an optimal sequence of moves but is inherently sequential: each move depends on the previously executed moves.

Surprisingly, there also exists a closed-form arithmetic description of individual moves in the optimal solution. This allows the computation of any constant number of moves using highly parallel computation, placing this task within the complexity class TC^0.

Algorithm 1 Tower-of-Hanoi (Arithmetic / TC^0)

Require: Move index i, number of disks n

$d \leftarrow v_2(i) + 1$
$t \leftarrow \left\lfloor \frac{i}{2^{d-1}} \right\rfloor$
if $n - d$ is even **then**
  $\delta_d \leftarrow +1$
else
  $\delta_d \leftarrow -1$
end if
$\mathrm{src}(i) \leftarrow \delta_d \cdot (t-1) \bmod 3$
$\mathrm{dst}(i) \leftarrow \delta_d \cdot t \bmod 3$
Move disk d from peg $\mathrm{src}(i)$ to peg $\mathrm{dst}(i)$

Algorithm 2 Tower-of-Hanoi (Recursive)

Require: Number of disks n, pegs (s, a, t)

if $n = 1$ **then**
  Move disk 1 from s to t
else
  Tower-of-Hanoi$(n-1, s, t, a)$
  Move disk n from s to t
  Tower-of-Hanoi$(n-1, a, s, t)$
end if

TC^0 **Complexity Class** This captures computations that can be performed by highly parallel, constant-depth circuits that allow threshold gates. From an algorithmic perspective, this characterization emphasizes that such algorithms perform operations such as indexing, thresholding, parity, and constant-size arithmetic. In contrast, TC^0 circuits are generally not believed to support unbounded sequential processes such as recursion or general graph exploration.

3.3 Transformer Expressivity of Tower of Hanoi

Prior literature has mapped the transformer architecture to classical models of computation Strobl et al. (2024), within which Boolean circuits are the most natural due to the inherent parallelizability of attention, and also analyzed how adding computation resources like padding tokens Merrill & Sabharwal (2025a), depth scaling Merrill & Sabharwal (2025b), etc. increase the expressive power of these models. Recent expressibility results Merrill & Sabharwal (2025b) Merrill & Sabharwal (2024) place a class of transformers with logarithmic CoT within uniform TC^0. We adopt constant-depth threshold circuits (TC^0) as a convenient abstraction for the algorithmic capabilities of small

transformer models. Consequently, problems whose solutions can be expressed by direct closed-form arithmetic computations are well aligned with the expressive power of these transformers, while tasks that are inherently sequential would not.

We show the explicit construction of a simple transformer that expresses the path for Tower of Hanoi graph given the configuration graph G_n

Theorem 3.1. *There exists a 2-head 1-layer decoder only transformer with embedding dimension $O(\exp(n))$ that given the Tower of Hanoi graph G can output the path needed to move the configuration with all disks on 1 tower to all disks on another*

ToH in RASP We show that the TC^0 closed form algorithm can be expressed in RASP and compile it to a Transformer object with 4 heads and 7 layers, where the width of the MLP blocks scales quadratically with n, thus proving that the algorithm is expressible with the attention model. The compiled model also only requires attention in the first layer, followed by a series of MLP operations.

Other architectural and implementation details can be found in Section D.1, D.2 and the code repository.

4 Experiments

Building on the distinction between sequential and parallelizable computation discussed above, we design experiments that probe the limits of large reasoning models under structured algorithmic tasks. Compared to prior work such as the Shojaee* et al. (2025), our evaluations explicitly leverage closed-form and TC^0-style computations as guiding principles. We examine whether this alignment improves robustness and increases the failure threshold beyond which model performance degrades.

4.1 Initial Experiments to Improve Failure Threshold

While Shojaee* et al. (2025) reports that providing explicit algorithms to reasoning models does not lead to substantial accuracy gains, we show that modification such as prompt refinements, use of in-context examples, improved algorithm selection, and modifications to the output format can result in significant performance improvements on certain puzzles. A detailed description of the experimental setup is provided in Appendix **??**

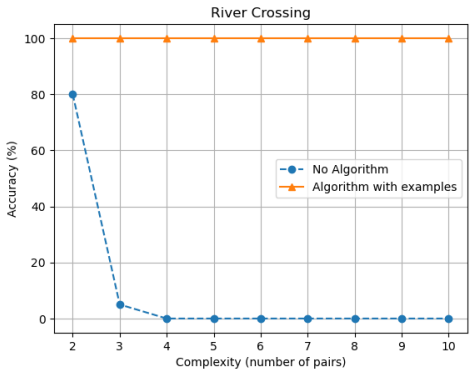

Figure 2: River Crossing with Algorithm

4.2 Graph as input

Since the tower of Hanoi problem can be represented as a graph reachability instance, we programmatically construct a graph which captures the given problem. We when pass the graph, along with the start and end node to the LRM. The following plot indicates the improvement in the performance.

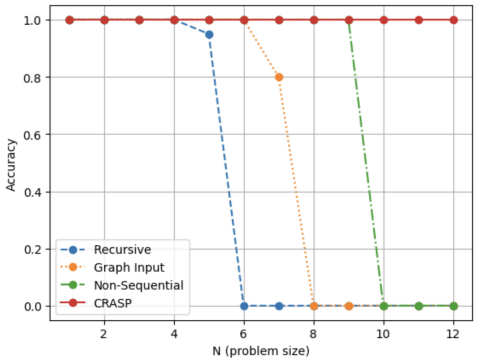

Figure 3: Accuracy across algorithms

Although the LRM is able to solve the problem for instances less than 8, due to the exponential increase in the size of the graph with the increase in n, it becomes impractical to feed the entire graph to the model.

4.3 Non-sequential (TC^0) Algorithm

To combat the issue arising due to the increase in graph size, we use the efficient TC^0 algorithm to split the long horizon reasoning task into non-sequential sub-tasks. Due to the independence of the current step from the previous step we are able to prevent both the accumulation of error as well as increase the accuracy significantly. Using this method, the accuracy now shoots up to a 100% till $n = 9$ (figure 3).

Other Puzzle Problems with TC^0 algorithms We also note that the checkers jumping problem also has a TC^0 algorithm that improves the failure threshold (figure 4)

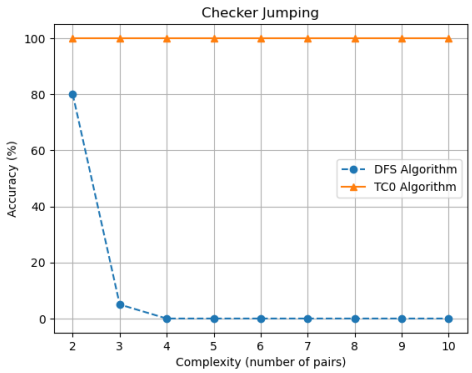

Figure 4: Checker Jumping with different algorithms

4.4 Training a Transformer

We evaluate the transformer on the Tower of Hanoi reachability task with training instances up to $n = 10$ disks and test instances up to $n = 13$. The model reliably learns to predict the identity of the disk moved at each time step, generalizing almost perfectly to larger n well beyond the training range. In contrast, predicting the source and destination pegs is significantly harder: while accuracy is high for small n, it degrades rapidly as n increases. This gap suggests that the model captures the structure governing which disk moves but struggles to infer the finer-grained tower dynamics required to determine where the disk moves for longer horizons.

Architectural details We use a decoder-only Transformer with 4 layers, 8 self-attention heads per layer, and a model dimension of 512, with a feedforward dimension of 2048. Inputs are tokenized representations of problem size n, move index t, and disk index, embedded using learned embeddings and combined with learned positional embeddings. The encoder outputs are pooled by selecting the final token representation, which is then fed into task-specific linear heads. One head predicts the disk moved, while two separate heads predict the source and destination towers. All components are trained jointly using cross-entropy loss with AdamW optimization.

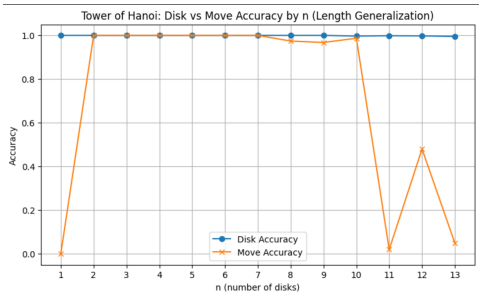

Figure 5: Performance of a simple transformer

5 Conclusion

We investigated the failure modes of large reasoning models (LRMs) on combinatorial puzzle problems through the lens of transformer expressibility. Although prior work attributes performance degradation to exponential growth of the underlying state space, our results suggest that the algorithmic structure of the solution plays a decisive role.

By reframing puzzles such as Tower of Hanoi and Checker Jumping as reachability problems on configuration graphs, we distinguished between inherently sequential solution strategies and closed-form, highly parallelizable algorithms. We construct small constant-depth transformers can express the closed-form solution for the Tower of Hanoi problem. Empirically, we demonstrated that explicitly providing this TC^0 algorithm substantially improves the failure threshold of LRMs.

Two natural directions follow. First, it would be valuable to explore this expressibility gap across a broader class of puzzle reasoning problems. Second, beyond expressibility, an important open question is learnability: how architectural biases and structural alignment affect what models can reliably acquire under gradient-based trainingYang et al. (2025).

References

Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Grid graph reachability problems. In *Proceedings of the 21st Annual IEEE Conference on Computational Complexity*, CCC '06, pp. 299–313, USA, 2006. IEEE Computer Society. ISBN 0769525962. doi: 10.1109/CCC.2006.22. URL https://doi.org/10.1109/CCC.2006.22.

Anthropic. Claude 3.7 sonnet, Feb 2025. https://www.anthropic.com/index/claude-3-7-sonnet.

David Chiang. Transformers in uniform tc^0, 2025. URL https://arxiv.org/abs/2409.13629.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint*, arXiv:2501.12948, 2025a.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms

via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025b. URL https://arxiv.org/abs/2501.12948.

Robert A. Hearn and Erik D. Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1):72–96, 2005. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2005.05.008. URL https://www.sciencedirect.com/science/article/pii/S0304397505003105. Game Theory Meets Theoretical Computer Science.

Andreas M. Hinz, Sandi Klavžar, Uroš Milutinović, and Ciril Petr. *The Tower of Hanoi – Myths and Maths*. Birkhäuser, 2013 (2nd ed. 2018).

Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(75)80050-X. URL https://www.sciencedirect.com/science/article/pii/S002200007580050X.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Thomas McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability, 2023. URL https://arxiv.org/abs/2301.05062.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought, 2024. URL https://arxiv.org/abs/2310.07923.

William Merrill and Ashish Sabharwal. Exact expressive power of transformers with padding, 2025a. URL https://arxiv.org/abs/2505.18948.

William Merrill and Ashish Sabharwal. A little depth goes a long way: The expressive power of log-depth transformers, 2025b. URL https://arxiv.org/abs/2503.03961.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers, 2025c. URL https://arxiv.org/abs/2210.02671.

OpenAI. Introducing openai o1, Jan 2024. https://openai.com/research/introducing-openai-o1.

Omer Reingold. Undirected st-connectivity in log-space. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pp. 376–385, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139608. doi: 10.1145/1060590.1060647. URL https://doi.org/10.1145/1060590.1060647.

Tobias Schnabel, Kiran Tomlinson, Adith Swaminathan, and Jennifer Neville. Lost in transmission: When and why llms fail to reason globally. In *NeurIPS 2025*, December 2025. URL https://www.microsoft.com/en-us/research/publication/lost-in-transmission-when-and-why-llms-fail-to-reason-globally/.

Parshin Shojaee*, Iman Mirzadeh*, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. In *NeurIPS*, 2025. URL https://arxiv.org/abs/2506.06941.

Lena Strobl. Average-hard attention transformers are constant-depth uniform threshold circuits, 2023. URL https://arxiv.org/abs/2308.03212.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024. ISSN 2307-387X. doi: 10.1162/tacl_a_00663. URL http://dx.doi.org/10.1162/tacl_a_00663.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers, 2021. URL https://arxiv.org/abs/2106.06981.

Tong Yang, Yu Huang, Yingbin Liang, and Yuejie Chi. Multi-head transformers provably learn symbolic multi-step reasoning via gradient descent. 2025. URL https://api.semanticscholar.org/CorpusID:280566397.

Qilin Ye, Deqing Fu, Robin Jia, and Vatsal Sharan. When do transformers learn heuristics for graph connectivity?, 2025. URL https://arxiv.org/abs/2510.19753.

A TOWER OF HANOI IN L - UNIFORM TC^0

A.1 INDEX OF WHICH DISK TO MOVE IN THE OPTIMAL SOLUTION PATH AT STEP i ALGORITHM

Theorem A.1. *In the optimal Tower of Hanoi solution with n disks, the disk moved at step i (for $1 \le i \le 2^n - 1$) is the disk whose index equals the position of the least significant 1 bit in the binary representation of i. Equivalently,*

$$\mathrm{disk}(i) = v_2(i) + 1,$$

where $v_2(i)$ denotes the largest power of 2 dividing i (i.e., $v_2(i) = \max\{k : 2^k \mid i\}$).

Lemma A.2. *Let $n \ge 1$ and let $1 \le j \le 2^{n-1} - 1$. Then*

$$v_2(2^{n-1} + j) = v_2(j).$$

Proof. Write $j = 2^r \cdot t$ with t odd and $r = v_2(j) \ge 0$. Since $1 \le j \le 2^{n-1} - 1$ and j is a positive integer less than 2^{n-1}, we must have $r \le n - 2$. (If $r \ge n - 1$, then $j \ge 2^{n-1}$, contradicting $j \le 2^{n-1} - 1$.)

Therefore,

$$2^{n-1} + j = 2^{n-1} + 2^r \cdot t = 2^r(2^{n-1-r} + t).$$

Since $r \le n - 2$, we have $n - 1 - r \ge 1$, which means 2^{n-1-r} is even. Because t is odd, the sum $2^{n-1-r} + t$ is odd (even + odd = odd).

Thus, 2^r is the exact power of 2 dividing $2^{n-1} + j$, and so

$$v_2(2^{n-1} + j) = r = v_2(j). \qquad \square$$

Proof of the theorem. We proceed by induction on n.

Base case ($n = 1$). For $n = 1$ there is exactly one move $i = 1$, and $v_2(1) = 0$, so $\mathrm{disk}(1) = v_2(1) + 1 = 1$.

Inductive hypothesis. Assume the statement holds for $n - 1$: for all $1 \le j \le 2^{n-1} - 1$, $\mathrm{disk}(j) = v_2(j) + 1$ in the recursive solution for $n - 1$ disks.

Inductive step. The recursive solution for n disks is

$$S_n = S_{n-1}, \quad \text{move disk } n, \quad S_{n-1},$$

with total moves $1, 2, \ldots, 2^n - 1$.

1. If $1 \le i \le 2^{n-1} - 1$, the i-th move is in the first copy of S_{n-1}, so by the inductive hypothesis $\mathrm{disk}(i) = v_2(i) + 1$.
2. If $i = 2^{n-1}$, this is the single move of disk n. Since $v_2(2^{n-1}) = n - 1$, we get $\mathrm{disk}(i) = v_2(i) + 1 = n$.
3. If $2^{n-1} + 1 \le i \le 2^n - 1$, write $i = 2^{n-1} + j$. Then necessarily

   $$1 \le j \le 2^{n-1} - 1.$$

   (Indeed, the maximum possible j in this decomposition is $j = 2^{n-1} - 1$, and $2^{n-1} - 1 < 2^n - 1$, so j is strictly less than $2^n - 1$; this ensures i remains within the range $1, \ldots, 2^n - 1$.)

   By the recursive structure the i-th move of S_n is the same disk as the j-th move of S_{n-1}. By the lemma above we have

   $$v_2(i) = v_2(2^{n-1} + j) = v_2(j).$$

Applying the inductive hypothesis to S_{n-1} gives

$$\mathrm{disk}(i) = v_2(i) + 1 = v_2(j) + 1 = \mathrm{disk}(j),$$

which is exactly the disk moved in the j-th move of S_{n-1}.

Thus the formula holds for n. By induction it holds for all $n \geq 1$. □

Theorem A.3. *In the optimal Tower of Hanoi solution with n disks, the disk moved at step i ($1 \leq i \leq 2^n - 1$) is exactly the disk whose index equals the position of the least significant 1 bit of i. Equivalently,*

$$\mathrm{disk}(i) = v_2(i) + 1,$$

where $v_2(i) = \max\{k : 2^k \mid i\}$. Moreover, the predicate $\mathrm{disk}(i) = d$ is definable in uniform TC^0.

Proof. The classical recursive solution to the Tower of Hanoi problem has the following well-known characterization: disk d is moved exactly once every 2^{d-1} steps, and it is never moved on any step divisible by 2^d. Hence disk d is moved at step i if and only if

$$2^{d-1} \mid i \quad \text{and} \quad 2^d \nmid i.$$

Equivalently, $d - 1 = v_2(i)$, and therefore $\mathrm{disk}(i) = v_2(i) + 1$.

We now show that the predicate $\mathrm{disk}(i) = d$ is definable in uniform TC^0.

Represent i in binary as $i = (b_{m-1} \dots b_1 b_0)_2$ with $m = O(n)$. Then $v_2(i) = k$ holds exactly when

$$b_0 = b_1 = \cdots = b_{k-1} = 0 \quad \text{and} \quad b_k = 1.$$

Thus $\mathrm{disk}(i) = d$ holds iff

$$b_0 = \cdots = b_{d-2} = 0 \ \wedge \ b_{d-1} = 1.$$

This predicate is computable by a constant-depth, polynomial-size threshold circuit: we check with an AND gate that the first $d - 1$ bits are all 0, and with a single wire that bit $d - 1$ equals 1. Such bit tests and bounded conjunctions are in $\mathrm{AC}^0 \subseteq \mathrm{TC}^0$, uniformly constructible from n and d.

Therefore the function $\mathrm{disk}(i)$ is computable in uniform TC^0, and the stated formula holds. □

A.2 Peg number for a given disk at step i, i.e., state configuration at step i algo

Theorem A.4. *Let n be the number of disks and $1 \leq i \leq 2^n - 1$ a move index. Let*

$$d = \mathrm{disk}(i) = v_2(i) + 1.$$

Define

$$t(i, d) = \left\lfloor \frac{i}{2^{d-1}} \right\rfloor, \qquad \delta_d = \begin{cases} +1 & \text{if } n - d \text{ is even}, \\ -1 & \text{if } n - d \text{ is odd}, \end{cases}$$

with arithmetic modulo 3 on peg labels $\{0, 1, 2\}$.

Then the source and destination pegs of the move at step i are

$$\boxed{\begin{aligned} &\mathrm{src}(i) = \delta_d \cdot (t(i, d) - 1) \pmod 3, \\ &\mathrm{dst}(i) = \delta_d \cdot t(i, d) \pmod 3. \end{aligned}}$$

Moreover, the functions $\mathrm{disk}(i)$, $\mathrm{src}(i)$, and $\mathrm{dst}(i)$ are computable in uniform TC^0.

Proof. We use the standard structural properties of the optimal Tower of Hanoi solution.

Step 1: Identification of the moving disk. It is classical that disk d is moved exactly at those times i such that

$$2^{d-1} \mid i \quad \text{and} \quad 2^d \nmid i,$$

equivalently $d = v_2(i) + 1$. Thus $\mathrm{disk}(i) = d$.

Step 2: Frequency of motion. Disk d moves once every 2^{d-1} steps. Hence before step i, disk d has moved

$$t(i,d) - 1 = \left\lfloor \frac{i}{2^{d-1}} \right\rfloor - 1$$

times, and after step i it has moved $t(i,d)$ times.

Step 3: Direction of motion. In the optimal solution, each disk moves monotonically in a fixed cyclic direction around the three pegs. The direction depends only on the parity of $n - d$: if $n - d$ is even, disk d moves clockwise; if $n - d$ is odd, it moves counterclockwise. This direction is encoded by $\delta_d \in \{+1, -1\}$.

Step 4: Peg positions. Each move advances disk d by one peg in its fixed direction. Starting from peg 0, after k moves disk d is on peg

$$\delta_d \cdot k \pmod 3.$$

Therefore:

- Before step i, disk d is on peg
$$\delta_d \cdot (t(i,d) - 1) \pmod 3,$$
- After step i, disk d is on peg
$$\delta_d \cdot t(i,d) \pmod 3.$$

Thus the move at time i is exactly from

$$\mathrm{src}(i) = \delta_d(t(i,d) - 1) \pmod 3 \quad \text{to} \quad \mathrm{dst}(i) = \delta_d t(i,d) \pmod 3.$$

This is the unique legal optimal move, proving correctness. □

Complexity Analysis. We show that $\mathrm{disk}(i)$, $\mathrm{src}(i)$, and $\mathrm{dst}(i)$ are computable in uniform TC^0.

Disk index. The predicate $v_2(i) = k$ is definable by testing

$$b_0 = \cdots = b_{k-1} = 0 \ \wedge \ b_k = 1,$$

which is in $\mathrm{AC}^0 \subseteq \mathrm{TC}^0$. Hence $d = v_2(i) + 1$ is in uniform TC^0.

Move count. The quantity $t(i,d) = \lfloor i/2^{d-1} \rfloor$ is obtained by shifting i right by $d - 1$ bits. Bit selection and shifts by a TC^0-computable index are in uniform TC^0.

Direction bit. The parity of $n - d$ is computable in AC^0, hence $\delta_d \in \{+1, -1\}$ is definable in uniform TC^0.

Modulo 3 reduction. Multiplication by ± 1 and reduction modulo 3 are computable by constant-depth threshold circuits; $\mathrm{mod}3$ is in uniform TC^0.

Conclusion. All operations in the definitions of $\mathrm{src}(i)$ and $\mathrm{dst}(i)$ — bit tests, shifts, parity, multiplication by ± 1, and reduction modulo 3 — lie in uniform TC^0. Therefore the full transducer

$$i \mapsto (\mathrm{disk}(i), \mathrm{src}(i), \mathrm{dst}(i))$$

is computable in uniform TC^0. □

Since TC^0 is an upper bound on Softmax Attentions Transformers, we also prove that the aforementioned formula is computable by a transformer by writing it in RASP.

B OTHER PUZZLE PROBLEMS IN L - UNIFORM TC^0

B.1 RIVER CROSSING

Algorithm 3 RIVER-CROSSING($\{(a_i, A_i)\}_{i=1}^{n}$)

for $i = 1$ to n **do**
  Send (a_i, A_i) across the river
  Return a_i alone
  Send (a_i, a_{i+1}) across the river
  Return a_{i+1} alone
end for

B.2 CHECKERS JUMPING

Algorithm 4 Recursive Backtracking Solver for Checker Jumping

Require: Current board $board$, target configuration $goal$, list $moves$
Ensure: Sequence of moves reaching $goal$, or **null** if none exists
Solve$board, goal, moves$
if $board = goal$ **then**
  return $moves$
end if
for $i = 1$ to length($board$) **do**
  if $board[i] = R$ **then**
    $result \leftarrow$ TryMove$R, i, i+1, board, goal, moves$
    if $result \neq null$ **then**
      return $result$
    end if
    $result \leftarrow$ TryMove$R, i, i+2, board, goal, moves$
    if $result \neq null$ **then**
      return $result$
    end if
  end if
  if $board[i] = B$ **then**
    $result \leftarrow$ TryMove$B, i, i-1, board, goal, moves$
    if $result \neq null$ **then**
      return $result$
    end if
    $result \leftarrow$ TryMove$B, i, i-2, board, goal, moves$
    if $result \neq null$ **then**
      return $result$
    end if
  end if
end for
return null
TryMove$piece, from, to, board, goal, moves$
if to is a valid position AND $board[to] =$ _ **then**
  $new\_board \leftarrow$ copy of $board$
  $new\_board[from] \leftarrow$ _
  $new\_board[to] \leftarrow piece$
  $new\_moves \leftarrow moves \cup \{(from, to)\}$
  return Solve$new_board, goal, new_moves$
end if
return null

Algorithm 5 Move Type at Index i

Require: Move index i
Ensure: Symbol at move i
For all r in parallel:
  $d_r \leftarrow |pos_B[r] - pos_A[r]|$
Compute in parallel prefix sums:
  $S_r \leftarrow \sum_{t \le r} d_t$
  $S_{r-1} \leftarrow S_r - d_r$
For all r in parallel:
  $owner_r \leftarrow (S_{r-1} < i \le S_r)$
Let r^* be the unique r with $owner_r = 1$
$offset \leftarrow i - S_{r^*-1}$
if $offset \le \lfloor d_{r^*}/2 \rfloor$ **then**
  if piece r^* is R **then**
    return J
  else
    return j
  end if
else
  if piece r^* is R **then**
    return M
  else
    return m
  end if
end if

C Tower of Hanoi Transformer

C.1 Edge Pattern Algo

An iterative solution Hinz et al. (2013 (2nd ed. 2018) to the Tower of Hanoi problem can be described as the repeated execution of the following sequence of steps until the goal configuration is reached:

- Move one disk between pegs A and B, in whichever direction the move is legal.
- Move one disk between pegs A and C, in whichever direction the move is legal.
- Move one disk between pegs B and C, in whichever direction the move is legal.

Following this procedure, the stack of disks will end up on peg B if the number of disks is odd, and on peg C if the number of disks is even.

The reachability version of this uses the following algorithm when the moves $A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A$ are labelled as $\{1, 2, 3\}$

Algorithm 6 Follow Edge Pattern in a Colored Graph

Input: Graph $G = (V, E)$ with edge colorings $\{1, 2, 3\}$
Initialize: Start at the initial vertex v_0
while $v \neq$ final state **do**
  for color c in sequence $[1, 2, 3]$ **do**
    Follow the edge from current vertex v with color c
    Update current vertex v to the reached vertex
    if v is the final state **then**
      break
    end if
  end for
end while

Yang et al. (2025) shows explicit construction of a 2 layered transformer with multiple heads that can be trained to solve the reachability problem on trees. We show that a modified construction can be used to solve the Tower of Hanoi with the above algorithms.

C.1.1 Equivalence to patterned Graph reachability

The above algorithm shows that solving graph reachability where the path is just $(abc)^n$ with a, b, c being colored edges. Hence out path finding task for solving Tower of Hanoi is the following:

Path finding

Embedding For each node i, we represent by $a_i \in \mathbb{R}^{d_1}$ to be its vertex encoding. Suppose that $\{1,0,0\}, \{0,1,0\}, \{0,0,1\}$ is the color encoding of the 3 edge colors, we now represent the edge $e = (p(i), i)$ with color c_i by encoding it as $(a_{p(i)}^\top, 000, a_i^\top, c_i^\top)^\top$

C.1.2 Transformer Architecture

As in Yang et al. (2025) the architecture for a single layer transformer with $H \geq 1$ heads with trainable parameters θ is as follows:

Given the input embedding matrix $E_{d\times n}$ for every head $h \in [H]$ we have the query, key, value $W^K_{d\times d}, W^Q_{d\times d}, W^V_{d\times d}$ matrices that give the the head output:

$$\text{head}_h(E) = W^V E \cdot \text{softmax}\Big(E^\top W_h^{K\top} W_h^Q E\Big)$$

$\text{softmax}\Big(E^\top W_h^{K\top} W_h^Q E\Big)$ gives the attention matrix on the previous tokens (each of the n d-dimensional tokens).

$$f(E;\theta) = W^O \begin{pmatrix} \text{head}_1(E) \\ \vdots \\ \text{head}_H(E) \end{pmatrix},$$

We consider step-by-step reasoning in an autoregressive manner. At each reasoning step $k > 0$, given the input embedding $E^{(k)}(G)$, the output vector is taken as the last column of the output matrix, which is then concatenated with the input embedding to form the new input to the next reasoning step,

$$\widehat{o}^{(k+1)}(G,\theta) = f(E^{(k)}(G),\theta)_{:-1}, \quad E^{(k+1)}(G) = (E^{(k)}(G), \widehat{o}^{(k+1)}(T,\theta)) \tag{1}$$

C.1.3 Expressibility for Backward Reasoning

Goal to root:

For the backward reasoning task, we are interested in outputting the path from the goal node to the root node (g2r). Let the input embedding of the transformer be

$$E = E_{\text{g2r}}(\mathcal{G}) = \begin{pmatrix} X(\mathcal{G}) \\ Y(\mathcal{G}) \end{pmatrix} = \begin{pmatrix} x_1 & \cdots & x_{l(\mathcal{G})} & a_{g(\mathcal{G})} \\ 0 & \cdots & 0 & c_g \\ y_1 & \cdots & y_{l(\mathcal{G})} & a_0 \\ c_1 & \cdots & c_{l(\mathcal{G})} & 0 \end{pmatrix} \in \mathbb{R}^{2d_l \times (l(\mathcal{G})+1)}, \tag{2}$$

where $a_0 \in \mathbb{R}^{d_l}$ is used to fill the empty slots. Note that c is the embedding of the color and this can be of size 3 since there are only 3 possible colors We set the ground label of $\mathcal{T}$ as the embedding of the reversed path:

$$O_{\text{g2r}}(\mathcal{G}) = \begin{pmatrix} a_{p^1(g(\mathcal{T}))} & a_{p^2(g(\mathcal{T}))} & \cdots & a^{r(\mathcal{T})} \\ c_{p^1(g(\mathcal{T}))} & c_{p^2(g(\mathcal{T}))} & \cdots & c_{r(\mathcal{T})} \\ a_{g(\mathcal{T})} & a_{p(g(\mathcal{T}))} & \cdots & a_{p_{m(\mathcal{T})-1}(g(\mathcal{T}))} \\ 0 & \cdots & 0 & 0 \end{pmatrix} \in \mathbb{R}^{2(d_l+3)\times m(\mathcal{T})}. \tag{3}$$

We consider a one-layer single-head transformer, where $H = 1$. We impose the following parameters: let $W_1^V = W_1^K = I_{2d_l}$, and set

$$W^O = \begin{pmatrix} I_{d_1} & 0_{d_1\times 3} & 0_{d_1\times d_1} & 0_{d_1\times 3} \\ 0_{3\times d_1} & 0_{3\times 3} & 0_{3\times d_1} & P_{3\times 3} \\ 0_{d_1\times d_1} & 0_{d_1\times 3} & I_{d_1} & 0_{d_1\times 3} \\ 0_{3\times d_1} & 0_{3\times 3} & 0_{3\times d_1} & 0_{3\times 3} \end{pmatrix}, P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \tag{4}$$

$$W_1^Q = \begin{pmatrix} 0_{(d_1+3)\times(d_1+3)} & 0_{(d_1+3)\times(d_1+3)} \\ B^* & 0_{(d_1+3)\times(d_1+3)} \end{pmatrix}, B^* = \begin{pmatrix} B_{d_1\times d_1} & 0_{d_1\times 3} \\ 0_{3\times d_1} & I_3 \end{pmatrix} \tag{5}$$

where $B \in \mathbb{R}^{d_l\times d_l}$ is trainable. In this case, $\theta = \{B\}$.

Multi-step reasoning. We set the first step $E_{\text{g2r}}^{(0)} = E_{\text{g2r}}(\mathcal{T})$, and let

$$X^{(k-1)} := E_{\text{g2r}}^{(k-1)}(1 : d_l + 3, :) \in \mathbb{R}^{(d_l+3)\times(l(\mathcal{T})+k)}$$

$$Y^{(k-1)} := E_{\text{g2r}}^{(k-1)}(d_l + 4 : 2d_l + 6, :) \in \mathbb{R}^{(d_l+3)\times(l(\mathcal{T})+k)}.$$

Then according to the transformer architecture described above, we have the output $\hat{o}^{(k)}$ at reasoning step k as

$$\hat{o}^{(k)} = W^O \begin{pmatrix} X^{(k-1)} \\ Y^{(k-1)} \end{pmatrix} \cdot \text{softmax}\Big(Y^{(k-1)T} B x_{l-1}^{(k-1)}\Big), \tag{6}$$

where $x_{l-1}^{(k-1)}$ is the last column of $X^{(k-1)}$. Reasoning recursively for $m(\mathcal{T})$ steps according to (2), we obtain the output

$$\hat{O}_{\text{g2r}}(\mathcal{G}; \theta) = (\hat{o}^{(1)}, \ldots, \hat{o}^{(m(\mathcal{G}))}).$$

We make the following assumption on the node embeddings for the backward reasoning case.

Assumption 1 (Linear independent embeddings (backward reasoning))**.** *Suppose $a_0 = 0_{d_l}$, and $\{a_i\}_{i\in[S]}$ are linearly independent.*

Let $A := (a_1, \ldots, a_S) \in \mathbb{R}^{d_l\times S}$ be the embedding matrix. The following theorem provides a construction of the transformer that solves backward reasoning.

Theorem C.1 (Construction for backward reasoning)**.** *Under Assumption 1, for any $\alpha \in \mathbb{R}$, there exists $B = B_\alpha \in \mathbb{R}^{d_l\times d_l}$ such that*

$$A^\top B A = \alpha I_S. \tag{7}$$

Let $\theta = \{B_\alpha\}$, then for any tree $\mathcal{T}$, we have

$$\hat{O}_{\text{g2r}}(\mathcal{G}; \theta) \to O_{\text{g2r}}(\mathcal{G}) \quad \text{as } \alpha \to +\infty.$$

The above when $\alpha \to +\infty$ ensures that

$$\begin{pmatrix} X^{(k-1)} \\ Y^{(k-1)} \end{pmatrix} \cdot \text{softmax}\Big(Y^{(k-1)T} B^* x_{:-1}^{(k-1)}\Big) = \begin{pmatrix} a_{\rho^{(k)}(g)} \\ 0 \\ a_{\rho^{(k-1)}(g)} \\ c_{\rho^{(k-1)}(g)} \end{pmatrix} \tag{8}$$

Note that W^O performs the affine transform needed to give

$$W^O \begin{pmatrix} a_{\rho^{(k)}(g)} \\ 0 \\ a_{\rho^{(k-1)}(g)} \\ c_{\rho^{(k-1)}(g)} \end{pmatrix} = \begin{pmatrix} a_{\rho^{(k)}(g)} \\ c_{\rho^{(k)}(g)} \\ a_{\rho^{(k-1)}(g)} \\ 0 \end{pmatrix} \tag{9}$$

D EXPERIMENTAL DETAILS

We provide the exact prompts used in all experiments to ensure full reproducibility. All experiments were conducted using DeepSeek-R1 Guo et al. (2025b).

Prompt for the Non-recursive Algorithm - Tower of Hanoi

You are a deterministic Python code execution engine.
Your task is to EXECUTE the Python code below exactly as written and return the FINAL OUTPUT.
STRICT RULES (must follow all):
- Do NOT explain the code.
- Do NOT add any commentary, text, or formatting.
- Do NOT modify the code.
- Do NOT abbreviate or partially print the output.
- Assume all required imports (e.g., math) are available.
- Follow Python operator precedence exactly.
- Return ONLY the final Python object produced by the function call.
- The output must be exactly what Python would produce.

Code:

```
def get_right_most_bit(n):
if n == 0:
return 0
res = n & (~n + 1)
return int(math.log2(res)) + 1

def to_h(start_step, end_step):
steps = []
for n in range(start_step, end_step + 1):
num_1 = get_right_most_bit(n)
num_2 = (n & (n - 1)) % 3
num_3 = ((n | (n - 1)) + 1) % 3
steps.append([num_1, num_2, num_3])
return steps
```

Function call:
`to_h(3, 4)`
$n = 3$
Binary: `11`

```
get_right_most_bit(3):
~3 + 1 = -3
3 & (-3) = 1
log2(1) + 1 = 1
num_1 = 1
num_2:
n & (n - 1) = 3 & 2 = 2
2 % 3 = 2

num_3:
n | (n - 1) = 3 | 2 = 3
(3 + 1) % 3 = 1
```

Result for $n = 3$:

```
[1, 2, 1]
```

Prompt for the Graph Algorithm - Tower of Hanoi

You are a BFS agent that continues a breadth-first search (BFS) algorithm.

You will be given a HISTORY in the following format:

{
"queue": [...], # frontier as (node, path) pairs,
"visited": [...], # nodes already visited,
"parent_map": {...}, # child → parent mapping (if any),
"adj_retrieved": {...} # adjacency lists fetched so far
}

You have access to TOOLS that can fetch adjacency lists of nodes.

YOUR JOB:
- Continue the BFS algorithm step-by-step.
- Use tools only when adjacency info is missing.
- Update the BFS state deterministically.
- When the goal node is found, reconstruct and output the path.

YOUR OUTPUT FORMAT RULES:

1. If BFS is still ongoing:
- Return ONLY the updated state JSON (no extra text).
- The format must be exactly:

{
"queue": [...],
"visited": [...],
"parent_map": {...},
"adj_retrieved": {...}
}

2. If BFS is finished and a path is found:
- Return ONLY:
FINAL_PATH:[v1, v2, v3, ...]

3. Do NOT include any explanations, reasoning, or commentary.

4. Do NOT output markdown, quotes, or prose — only raw JSON or FINAL_PATH line.

You must always return one of the two:
- The new BFS state JSON, or
- The final path (if done).

Prompt for River crossing

You are given a river crossing puzzle involving multiple agents $(A_1, A_2, \ldots)$ and their corresponding actors $(a_1, a_2, \ldots)$.

Your task is to compute a correct list of boat moves that solves the puzzle, following the rules below:

- Each agent A_i and actor a_i must end up on the right side of the river.
- The boat can carry at most two individuals at a time.
- The boat must always have at least one person on board to move.
- No actor must be left with agents other than its own without its own agent present, even if there are other actors present on either of the shores or the boat.
For example, a_1 cannot be left on the shore with (A_2, a_2, A_3) as A_1 is not present.
- Actors on the boat do not have to get down unless explicitly dropped off. For example, in the sequence
`[["A_2", "a_2"], ["a_2"], ["a_2", "a_1"], ...]`
no rule is violated as a_2 never gets off the boat after the second step.

2. Entity Naming

- Actors: $a_1, a_2, \ldots, a_n$
- Agents: $A_1, A_2, \ldots, A_n$

3. Output Format

Your final output must be a Python-style list of lists of strings. For example:

```
[["A_2", "a_2"], ["a_2"], ["a_2", "a_1"], ["a_1"], ["A_1",
"a_1"]]
```

4. Algorithm to Follow

```
 moves = [["A_1", "a_1"]]
for i in range(1, n):
moves.append([f"a_i"])
moves.append([f"a_i", f"a_i+1"])
moves.append([f"a_i+1"])
moves.append([f"A_i+1", f"a_i+1"])
return moves
```

This means the general sequence will look like:

```
 [["A_1", "a_1"], ["a_1"], ["a_1", "a_2"], ["a_2"], ["A_2",
"a_2"], ["a_2"], ["a_2", "a_3"], ["a_3"], ["A_3", "a_3"], ...]
```

FOLLOW THE ALGORITHM EXACTLY.

Prompt for Checkers

You are a helpful assistant. Solve this puzzle for me.

On a one-dimensional board, there are red checkers ('R'), blue checkers ('B'), and one empty space ('_').

A checker can move by either:
1. Sliding forward into an adjacent empty space, or
2. Jumping over exactly one checker of the opposite color to land in an empty space.

The goal is to swap the positions of all red and blue checkers, effectively mirroring the initial state.

Example: If the initial state is `['R', '_', 'B']`, the goal is to reach `['B', '_', 'R']`.

Your output should be a string or list of move symbols, where each symbol corresponds to a valid move made:

`'S'` = red checker (to immediate left of blank) slides right.
`'s'` = blue checker (to immediate right of blank) slides left.
`'J'` = red checker (to the 2nd left of the blank) jumps right over blue.
`'j'` = blue checker (to the 2nd right of the blank) jumps left over red.

Example solution for the initial state `['R', '_', 'B']` might be:

```
moves = ['S', 'j', 'S']
```

This means:
`'S'`: Red checker steps into the blank on its right.
`'j'`: Blue checker jumps left over red into the new blank.
`'S'`: Red checker steps into the blank on its right.

Your goal is to output the full sequence of symbolic moves to achieve the mirrored configuration.

Ensure the solution is correct and minimal (i.e., optimal number of moves).

D.1 Architectural Details of the RASP Model

Table 1: Architectural details for the RASP compiled model at $n = 20$

Parameter	Value	Notes
num_layers	7	One active attention layer and six additional blocks implementing arithmetic in the MLPs.
num_heads	4	Four-head attention used to broadcast metadata tokens in layer 0.
key_size	27	Scales approximately linearly with max_disks ($\approx$ max_disks + 7).
mlp_hidden_size	1804	Dominant width term, scaling quadratically with max_disks.
residual_dim	190	Dimension of the joint residual feature space.

Key dimensions as `max_disks` grows (single-step compilation).

max_disks	Residual dim	key_size	mlp_hidden_size
1	61	8	28
2	63	9	40
3	70	10	70
5	85	12	154
10	120	17	504
20	190	27	1804

Closed-form trends (matching observed data for `max_disks` ≥ 2).

$$\text{residual\_dim}(n) \approx 7n + 50,$$
$$\text{key\_size}(n) = n + 7,$$
$$\text{mlp\_hidden\_size}(n) = 4n^2 + 10n + 4.$$

The quadratic MLP width dominates parameter growth; the residual dimension and attention key size increase only linearly to accommodate additional scratch slots and positional bases.

D.2 RASP CODE

```
"""Compile Tower of Hanoi peg configuration into a Tracr transformer.
Compiles into transformer with 1 attention layer with 4 heads and
6 MLP layers. No layer norm, dropout and non causal masking.

The RASP program implements the formula for determining the peg index of each disk
at a given step in the optimal solution to the Tower of Hanoi problem.

The formula is:
    s_d ← (ell/2^d + 1/2 × (j − i) × ((n − d) mod 2 + 1) + i) mod 3

Where:
    - s_d: peg index for disk d at step
    - ell: step index
    - d: disk number
    - n: total number of disks
    - i: starting peg index
    - j: target peg index
    - x: floor function
"""
    BOS_TOKEN = "BOS"
# Positions 0-4 hold metadata tokens; everything from index 5 onward is scratch
# space used to write the peg index for each disk.
SCRATCH_OFFSET = 5

def _broadcast_value(position: int) -> rasp.SOp:
  """Broadcast a token at `position` across the entire sequence."""

  selector = rasp.Select(
    rasp.indices,
    rasp.indices,
    lambda key, unused_query, pos=position: key == pos)
  return rasp.Aggregate(selector, rasp.tokens)


def _disk_number_sop() -> rasp.SOp:
  """Return disk numbers (1-indexed) for scratch-pad positions, else None."""

  indices = rasp.indices
  return rasp.Map(
    lambda idx: (idx - (SCRATCH_OFFSET - 1)) if idx >= SCRATCH_OFFSET else None,
```

```
    indices)


def build_hanoi_program() -> rasp.SOp:
  """Construct a RASP program that writes peg indices for each disk."""

  n_tokens = _broadcast_value(0).named("n")
  start_peg = _broadcast_value(1).named("start_peg")
  target_peg = _broadcast_value(2).named("target_peg")
  step_index = _broadcast_value(3).named("step_index")

  disk_numbers = _disk_number_sop().named("disk_number")

  # Compute round(step_index / 2^disk) modulo 3 using bit operations.
  upper_bits_mod3 = rasp.SequenceMap(
    lambda ell, disk: ((int(ell) >> int(disk)) % 3)
    if isinstance(ell, int) and isinstance(disk, int) else None,
      step_index,
      disk_numbers,
  ).named("upper_bits_mod3")

  half_threshold_bit = rasp.SequenceMap(
    lambda ell, disk: ((int(ell) >> (int(disk) - 1)) & 1)
    if isinstance(ell, int) and isinstance(disk, int) and int(disk) > 0 else None,
      step_index,
      disk_numbers,
  ).named("half_threshold_bit")

  rounded_mod3 = rasp.SequenceMap(
      lambda upper, half: (upper + half) % 3 if upper is not None and half is not No
      upper_bits_mod3,
      half_threshold_bit,
  ).named("rounded_mod3")

  peg_delta_mod3 = rasp.SequenceMap(
      lambda target, start: (int(target) - int(start)) % 3
      if isinstance(target, int) and isinstance(start, int) else None,
      target_peg,
      start_peg,
  ).named("peg_delta_mod3")

  n_minus_d_parity = rasp.SequenceMap(
      lambda n_val, disk: (int(n_val) - int(disk)) & 1
      if isinstance(n_val, int) and isinstance(disk, int) else None,
      n_tokens,
      disk_numbers,
  ).named("n_minus_d_parity")

  parity_plus_one_mod3 = rasp.Map(
      lambda parity: (int(parity) + 1) % 3 if isinstance(parity, int) else None,
      n_minus_d_parity,
  ).named("parity_plus_one_mod3")

  parity_term_mod3 = rasp.SequenceMap(
      lambda base, start: (int(base) + int(start)) % 3
      if isinstance(base, int) and isinstance(start, int) else None,
      parity_plus_one_mod3,
      start_peg,
  ).named("parity_term_mod3")
```

```
    first_product_mod3 = rasp.SequenceMap(
        lambda a, b: (int(a) * int(b)) % 3
        if isinstance(a, int) and isinstance(b, int) else None,
        rounded_mod3,
        peg_delta_mod3,
    ).named("first_product_mod3")

    mod_three = rasp.SequenceMap(
        lambda a, b: (int(a) * int(b)) % 3
        if isinstance(a, int) and isinstance(b, int) else None,
        first_product_mod3,
        parity_term_mod3,
    ).named("mod_three")

    is_scratch = rasp.Map(
        lambda idx: idx >= SCRATCH_OFFSET, rasp.indices).named("is_scratch")

    scratch_values = rasp.SequenceMap(
        lambda scratch, value: value if scratch else None,
        is_scratch,
        mod_three,
    ).named("scratch_values")
    final_output = rasp.SequenceMap(
        lambda new, original: new if new is not None else original,
        scratch_values,
        rasp.tokens,
    ).named("hanoi_configuration")

    return final_output
```