UNDERSTANDING ALGORITHMIC LEARNING THROUGH TRANSFORMER SHORTCUTS

Yash Chauhan
School of Physical Sciences
National Institute of Science Education and Research
Bhubaneswar, Odisha 752050
yash.chauhan@niser.ac.in

Harisankar B
School of Mathematical Sciences
National Institute of Science Education and Research
Bhubaneswar, Odisha 752050
harisankar.b@niser.ac.in

April 18, 2025

ABSTRACT

Transformers are known for their impressive performance on tasks involving language, code, and symbolic reasoning—but how they handle sequential, algorithmic logic without any recurrence is still a puzzling question. This report explores how Transformers are able to simulate the behavior of finite-state automata using shallow, non-recurrent architectures. Instead of processing inputs step by step like a loop, they learn so-called "shortcut solutions" that compress the entire sequence into a few parallel layers. These shortcuts are often surprisingly efficient and are explained using ideas from automata theory and circuit complexity. While this approach works well within training settings, it also comes with limitations like poor generalization to new sequence lengths or distributions. The report is based on the paper *"Transformers Learn Shortcuts to Automata"* by Bingbin Liu et al [1].

1 Introduction

Neural networks have become increasingly capable of performing tasks that resemble algorithmic reasoning, such as parsing code, evaluating arithmetic expressions, or generating structured language. Many of these tasks can be naturally described using classical models of computation, such as finite-state machines or Turing machines, which rely on step-by-step transitions through a sequence of states. In contrast, Transformer models [2], which dominate modern sequence modeling, are non-recurrent and shallow—often having a depth that is far smaller than the length of the input sequence.

This contrast raises an intriguing question: how do Transformers manage to perform computations that seem to require sequential processing, without explicitly following a sequence of steps? Rather than simulating each transition in a step-by-step fashion, Transformers often learn what are called *shortcut solutions*—parallel strategies that replicate the overall effect of sequential logic using far fewer layers. These shortcuts compress computation into a smaller number of operations by reparameterizing or composing the underlying state transitions in clever ways.

A key setting for studying this phenomenon is the simulation of *finite-state automata*, particularly semiautomata, which update an internal state based on a sequence of inputs. Transformers are able to emulate these systems, not by iterating over transitions one at a time, but by learning global transformations that encode the entire sequence behavior. Remarkably, theoretical results show that any semiautomaton can be simulated with a Transformer of logarithmic depth, and that many important cases admit constant-depth solutions. These constructions draw on ideas from algebraic automata theory—especially Krohn-Rhodes decomposition—as well as results in circuit complexity.

The existence of such shortcut solutions has practical implications as well. Transformers trained using standard gradient descent can often discover these efficient encodings, even without any explicit supervision about the underlying automaton. At the same time, these shortcuts can be brittle, failing to generalize well to longer sequences or distribution shifts. Understanding when and how these shortcut mechanisms arise sheds light on the internal logic of Transformer models and their strengths and limitations when it comes to algorithmic learning.

2 Background

To understand how Transformers can simulate automata using shortcut solutions, it is important to first introduce the fundamental concepts involved in automata theory, algebraic structures like semigroups, circuit complexity, and the Transformer architecture itself.

2.1 Semiautomata and Finite-State Models

A fundamental model of sequential computation is the **finite-state automaton** (FSA). These systems process an input sequence one symbol at a time, updating an internal state via a transition function.

Definition 1. *A **finite-state automaton** is a tuple $(Q, \Sigma, \delta, q_0, F)$ where:*

- *Q is a finite set of states,*
- *Σ is a finite input alphabet,*
- *$\delta : Q \times \Sigma \to Q$ is a transition function,*
- *$q_0 \in Q$ is the initial state,*
- *$F \subseteq Q$ is the set of accepting states.*

FSAs are used to define regular languages, recognize patterns, and model bounded-memory computations. For the purposes of this study, we are interested in a simplified variant called a **semiautomaton**.

Definition 2. *A **semiautomaton** is a triple (Q, Σ, δ), where Q, Σ, and δ are defined as above. It evolves deterministically without designated accept states or outputs.*

Given an input string $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_T) \in \Sigma^T$ and initial state q_0, the system computes a trajectory $(q_1, q_2, \ldots, q_T) \in Q^T$ by recursive application of the transition function:

$$q_t = \delta(q_{t-1}, \sigma_t), \quad \text{for } t = 1, \ldots, T.$$

Semiautomata describe the core mechanism behind automata, parsers, counters, and deterministic finite-control systems. They appear across computing theory, symbolic processing, and even in models of reinforcement learning environments.

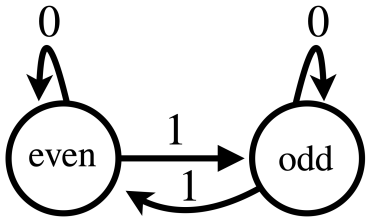

Figure 1: Example: A 2-state parity automaton that toggles between `even` and `odd` states on each input 1.

2.2 Groups & Semi-groups

To understand the algebraic structures underlying semiautomata, we need to introduce some basic concepts from group theory and semigroups.

Definition 3. *A **group** is a set G equipped with a binary operation $\cdot : G \times G \to G$ satisfying the following axioms:*

- ***Closure:** For all $a, b \in G$, $a \cdot b \in G$.*
- ***Associativity:** For all $a, b, c \in G$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.*
- ***Identity:** There exists an element $e \in G$ such that for all $a \in G$, $e \cdot a = a \cdot e = a$.*
- ***Inverses:** For each $a \in G$, there exists an element $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$.*

Definition 4. *A **semigroup** is a set S equipped with an associative binary operation $\cdot : S \times S \to S$, i.e., for all $a, b, c \in S$, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$. Unlike groups, semigroups do not necessarily have an identity element or inverses.*

Definition 5. *A subgroup N of a group G is called a **normal subgroup** if it is invariant under conjugation, i.e., for all $g \in G$ and $n \in N$, $gng^{-1} \in N$. This is denoted as $N \trianglelefteq G$.*

Definition 6. *Given a group G and a normal subgroup $N \trianglelefteq G$, the **quotient group** G/N is the set of cosets of N in G, with the group operation defined as $(aN)(bN) = (ab)N$ for $a, b \in G$.*

Definition 7. *A **simple group** is a nontrivial group whose only normal subgroups are the trivial group $\{e\}$ and the group itself. Simple groups are the building blocks of all finite groups, analogous to prime numbers in arithmetic.*

Definition 8. *A group G is **solvable** if it has a finite sequence of subgroups*

$$\{e\} = G_0 \trianglelefteq G_1 \trianglelefteq \cdots \trianglelefteq G_n = G,$$

such that each quotient G_{i+1}/G_i is abelian (i.e., commutative).

2.3 Transformation Semigroups

The behavior of a semiautomaton can be captured algebraically. Each input symbol $\sigma \in \Sigma$ defines a function:

$$\rho_\sigma : Q \to Q, \quad \rho_\sigma(q) = \delta(q, \sigma).$$

These functions compose naturally: $\rho_{\sigma_2} \circ \rho_{\sigma_1}(q) = \delta(\delta(q, \sigma_1), \sigma_2)$.

Definition 9. *The **transformation semigroup** of a semiautomaton (Q, Σ, δ) is the semigroup $\mathcal{T} = \langle \rho_\sigma \mid \sigma \in \Sigma \rangle \subseteq Q^Q$, generated under composition.*

Semigroups abstract the temporal dynamics of automata by viewing input sequences as composed maps. Some important structural cases:

- If every ρ_σ is a bijection, then $\mathcal{T}$ is a *permutation group*.
- If ρ_σ maps all states to a single state, it is called a *reset map*.

These transformation structures can be studied using tools from algebra. The complexity of simulating a semiautomaton is directly related to the complexity of its associated semigroup.

2.4 Examples of Transformation Semigroups

To illustrate the concept of transformation semigroups, we provide a few concrete examples that connect automata to algebraic structures 2:

Parity Automaton: The parity automaton provides a simple yet illustrative example of how automata can be related to algebraic structures. Specifically, the parity automaton, which toggles between two states (`even` and `odd`) based on the input, is isomorphic to the cyclic group C_2. The group C_2 consists of two elements, typically denoted $\{0, 1\}$, where 0 is the identity element and $1 + 1 = 0$ (addition modulo 2). In the context of the parity automaton, the state transition induced by an input of 1 corresponds to the group operation 1, while an input of 0 corresponds to the identity operation 0. This correspondence highlights the algebraic simplicity of the parity automaton and its connection to fundamental group-theoretic structures.

1-Bit Memory Unit: Similarly, the 1-bit memory unit, which stores a binary value and can be reset or toggled, is isomorphic to the flip-flop monoid. The flip-flop monoid consists of two operations: a toggle operation that switches the stored value and a reset operation that sets the value to a fixed state (e.g., 0). These operations form a monoid because they are associative and include an identity operation (doing nothing). The flip-flop monoid captures the essence of a minimal memory unit, demonstrating how even simple automata can be understood through algebraic abstractions.

2.5 Circuit Complexity

Circuit complexity provides a framework to classify problems based on the computational resources needed by Boolean circuits—abstract models of computation using logic gates.

Definition 10. ***A Boolean circuit** is an acyclic, directed graph in which the edges carry unidirectional logical signals and the vertices compute elementary logical functions. [3]*

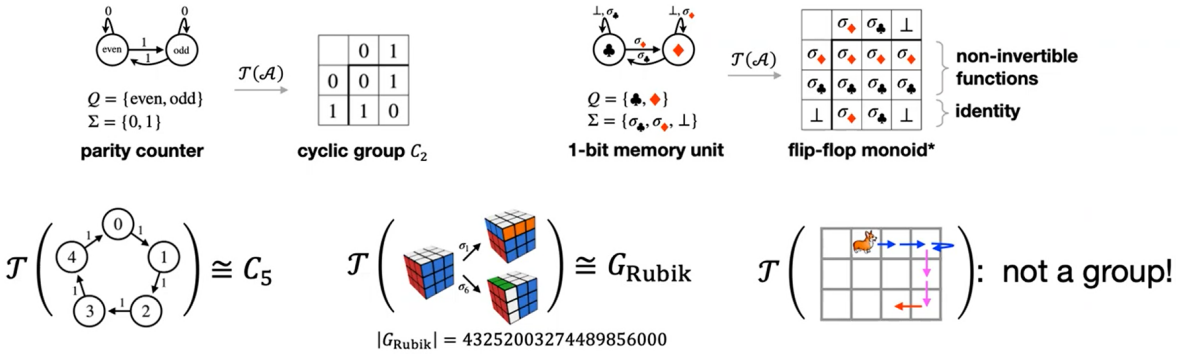

Figure 2: Some Examples of Transformation Semigroups

These classes capture families of circuits with specific constraints:

- **NC**^{0}: Very limited computational power.
  - Constant depth
  - Only 2-input AND, OR, NOT gates
  - Polynomial number of gates
  - Each output depends on only a constant number of inputs
- **AC**^{0}: Generalizes NC^0 by allowing more expressive gates.
  - Unbounded fan-in (i.e., n-input) AND, OR gates
  - Still constant depth and polynomial size
- **ACC**^{0}: Extends AC^0 with arithmetic operations.
  - Includes `mod-`p gates that compute sums modulo a fixed integer p
- **TC**^{0}: Allows even more complex behavior.
  - Adds *threshold gates*, such as majority functions
  - Powerful enough to simulate basic arithmetic like addition and comparison
- **NC**^{1}: Increases the circuit depth logarithmically.
  - Depth $O(\log T)$
  - 2-input AND, OR, NOT gates
  - Polynomial number of gates
  - Can compute problems like balanced parentheses (Dyck language)

Classes such as L, P, and NP represent even broader notions of computation not necessarily tied to circuits.

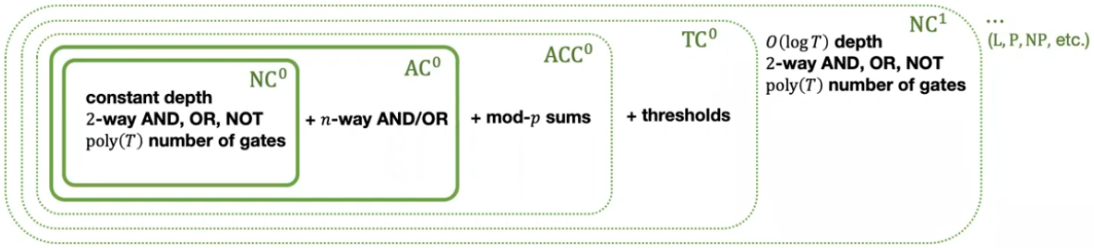

Figure 3: Hierarchy of Circuit Complexity Classes

Theorem 1 (Barrington's Theorem [4])**.** *Any language in* $\mathbf{NC^1}$ *can be recognized by a bounded-width branching program (i.e., a restricted automaton).*

This bridges finite automata and low-depth circuits. It also establishes that simulating arbitrary automata may require logarithmic depth, which becomes relevant when analyzing the theoretical limits of Transformers.

2.6 Transformer Architecture

The Transformer architecture [2] is a highly expressive model for sequence processing that operates without recurrence. Instead of updating internal states step by step as in recurrent networks, Transformers process all positions in parallel using layers of self-attention and feedforward computations.

Let $X \in \mathbb{R}^{T \times d}$ represent a sequence of T token embeddings, where each row $x_t \in \mathbb{R}^d$ encodes the t-th token. Since the architecture is invariant to input order, positional information is added using a positional encoding matrix $\text{PE} \in \mathbb{R}^{T \times d}$, which can be learned or fixed (e.g., sinusoidal). The input to the first layer is then $X + \text{PE}$.

The core operation is self-attention, which allows each position to attend to all others in the sequence. From X, one constructs queries, keys, and values as

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V,$$

where $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_k}$ are learned parameter matrices, and d_k is typically set such that $d_k = d/h$ for h attention heads.

The self-attention mechanism computes attention scores using scaled dot products:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

This results in a weighted sum of values, where each query attends to all keys and aggregates their corresponding values. Multiple attention heads operate in parallel, each with independent projections. The results are concatenated and linearly transformed to yield:

$$\text{MultiHead}(X) = \text{Concat}(H_1, \ldots, H_h)W^O,$$

where each H_i is a separate attention output and $W^O \in \mathbb{R}^{hd_k \times d}$ is a learned output projection.

Following self-attention, each position is independently passed through a position-wise feedforward network:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2,$$

where $W_1 \in \mathbb{R}^{d \times d_{ff}}$ and $W_2 \in \mathbb{R}^{d_{ff} \times d}$. This increases the representational capacity of the model by enabling nonlinear transformations after attention mixing.

Each layer also includes residual connections and layer normalization to stabilize training. Formally, for layer l with input $X^{(l)}$, the output is computed as:

$$Z^{(l)} = \text{LayerNorm}(X^{(l)} + \text{MultiHead}(X^{(l)})),$$
$$X^{(l+1)} = \text{LayerNorm}(Z^{(l)} + \text{FFN}(Z^{(l)})).$$

A Transformer encoder consists of L such layers applied in sequence, producing the final representation $X^{(L)}$. Importantly, all computations are parallel across positions, which allows efficient GPU execution and enables the model to model long-range dependencies in a small number of steps.

Shortcut Solutions

Definition 11 (Shortcut solution)**.** *Let $\mathcal{A}$ be a semiautomaton. For every $T \geq 1$, let f_T be a sequence-to-sequence neural network which simulates $\mathcal{A}$ at length T. Then, we call this sequence $\{f_T\}_{T \geq 1}$ a* shortcut *to $\mathcal{A}$ if the sequence of network depths $D := \{D(f_T)\}_{T \geq 1}$ satisfies $D \leq o(T)$.*

A key reason Transformers are well-suited for shortcut solutions lies in how they perform computations. Unlike Recurrent Neural Networks (RNNs), which operate strictly in a sequential manner across time, Transformer layers compute in parallel across all positions. That is, each layer processes the entire sequence simultaneously, while the overall depth of the computation corresponds to the number of layers stacked.

To make this contrast more concrete, consider the task of computing parity over a sequence of bits. There are two natural ways to solve this:

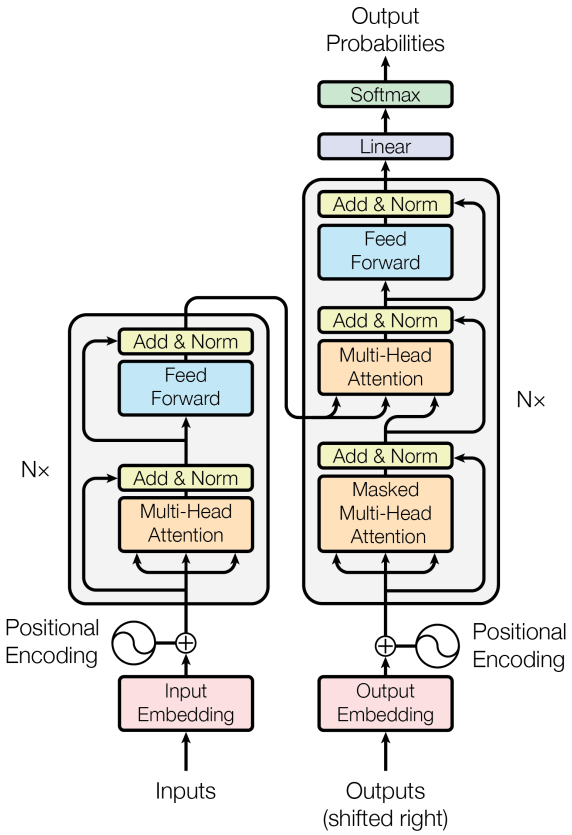

Figure 4: Transformer architecture: position embeddings, multi-head self-attention, feedforward network. [2]

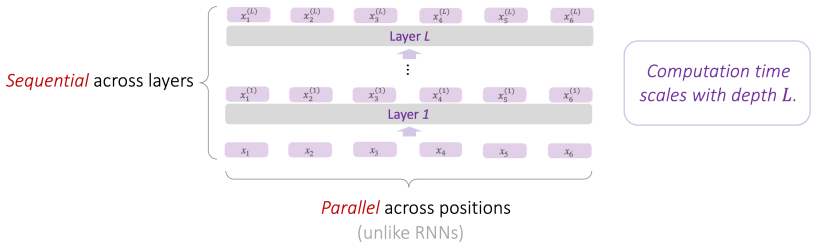

Figure 5: Illustration of sequential vs parallel computation in Transformers. Each layer is computed sequentially, but positions within the same layer are processed in parallel.

1. The **iterative solution**, defined by the recurrence:

$$q_t = \delta(q_{t-1}, \sigma_t) = q_{t-1} \oplus \sigma_t$$

This corresponds to a computation graph where each q_t depends on q_{t-1}, forming a chain of T sequential steps.

2. The **parallel solution**, which computes:

$$q_t = \left(\sum_{\tau \leq t} \sigma_\tau \right) \mod 2$$

This version requires computing the prefix sum followed by a modulo operation. All inputs can be processed in parallel to produce the result, as the graph has only one sequential step and depth is constant.

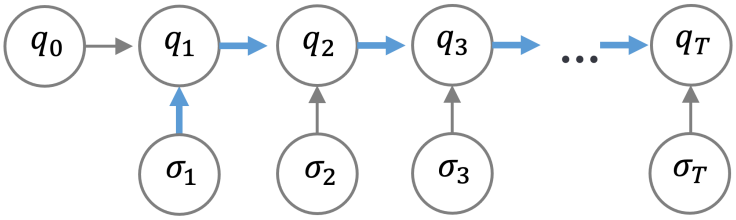

Figure 6: The iterative computation graph requires T sequential steps.

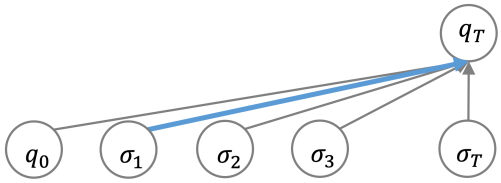

Figure 7: The parallel solution computes parity in one step using sum and mod operations.

The longest path in the computation graph for the iterative solution has length T, while in the parallel version it is only 1. This motivates the notion of a shortcut: a solution to $\mathcal{A}$ that reduces the number of sequential operations from $\sigma(T)$ to something sublinear, like $\mathcal{O}(\log T)$ or even $\mathcal{O}(1)$.

While the word "shortcut" is sometimes associated with spurious correlations or brittle heuristics, in this context, it refers to a desirable and mathematically grounded reduction in computation depth. For automaton simulation tasks, shortcuts are not only allowed but encouraged — they signify more efficient representations that Transformers are naturally capable of exploiting.

3 Theoretical Results

Can shallow Transformers really simulate deep algorithmic reasoning by learning the right computational shortcuts? To explore this, it helps to treat automata not just as state machines, but as algebraic objects — semiautomata defined over transition monoids. What follows is a sequence of formal results that establish how and when Transformer circuits, with limited depth and no recurrence, can simulate these systems.

Some of these results come from general parallelization properties that hold for any associative computation, while others rely on deep structural decompositions from algebra — revealing that even seemingly complex sequential behavior can sometimes be flattened into shallow circuits. Along the way, examples like parity, modular counters, and gridworld dynamics help illustrate how these ideas translate into actual network architectures.

Theorem 2 (Simulation is generically parallelizable; informal)**.** *Transformers can simulate all semiautomata $\mathcal{A} = (Q, \Sigma, \delta)$ at length T, with depth $O(\log T)$, embedding dimension $O(|Q|)$, attention width $O(|Q|)$, and MLP width $O(|Q|^2)$.*

To simulate a semiautomaton over an input sequence $\sigma_{1:T} \in \Sigma^T$, the state evolution can be expressed as a composition of transition functions:

$$q_T = \rho_{\sigma_T} \circ \cdots \circ \rho_{\sigma_1}(q_0)$$

where $\rho_\sigma : Q \to Q$ is defined by $\rho_\sigma(q) = \delta(q, \sigma)$.

Each ρ_σ can be represented as a $|Q| \times |Q|$ matrix acting on a one-hot encoded state. For example, in a parity automaton with $Q = \{0, 1\}$ (even/odd parity) and $\Sigma = \{0, 1\}$:

$$M_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Applying a 1 flips the parity; applying a 0 leaves it unchanged.

The final state is given by:

$$e_{q_T} = M_{\sigma_T} \cdots M_{\sigma_1} e_{q_0}$$

Matrix multiplication is associative, so the full computation can be reorganized as a binary tree of depth $O(\log T)$.

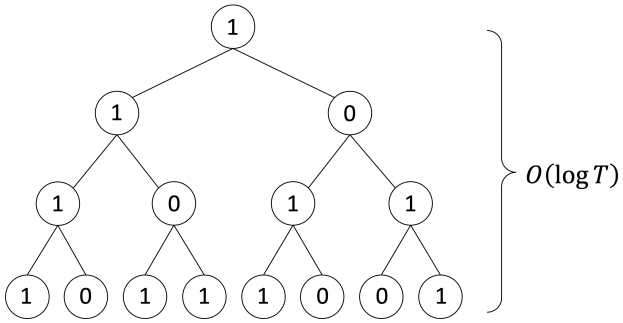

Figure 8: Associativity enables divide-and-conquer computation in $O(\log T)$ depth

A Transformer circuit can simulate each level of this tree using one layer. At each step, the model composes pairs of transition functions. After $\log T$ layers, the full composition is computed. Because no step-by-step unrolling is needed, the simulation avoids recurrence entirely.

Theorem 3 (Transformer Krohn-Rhodes; informal)**.** *Transformers can simulate all solvable semiautomata $\mathcal{A} = (Q, \Sigma, \delta)$, with depth $O(|Q|^2 \log |Q|)$, embedding dimension $2^{O(|Q| \log |Q|)}$, attention width $2^{O(|Q| \log |Q|)}$, and MLP width $|Q|^{O(2^{|Q|})} + 2^{O(|Q| \log |Q|)} \cdot T$.*

A powerful insight from algebraic automata theory is that even seemingly complex automata can often be broken into smaller, composable parts. This idea is made precise by the Krohn–Rhodes theorem.

Theorem 4 (Krohn–Rhodes)**.** *Every finite transformation semigroup can be expressed as a cascade product of permutation groups and reset semigroups.*

In the context of semiautomata, this means that any solvable automaton can be simulated by interleaving two basic types of behavior: modular counting and memory resets. Each component corresponds to an algebraically simple operation and can be simulated efficiently using a Transformer.

The modular counter component is used to track cyclic quantities such as parity or modulo-p arithmetic. Consider a parity automaton with state space $Q = \{0, 1\}$ and binary inputs $\sigma_t \in \{0, 1\}$. Each input either flips or preserves the state:

$$M_0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad M_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The state after t steps is given by:

$$q_t = M_{\sigma_t} \cdots M_{\sigma_1} q_0$$

Since matrix multiplication is associative, this can be parallelized. In Transformers, this is implemented using uniform attention, where each input contributes equally to the output:

$$s_t = \sum_{i=1}^{t} \sigma_i, \quad q_t = s_t \bmod 2$$

The sum is computed using uniform attention weights and passed through a depth-1 MLP to reduce mod p.

The second component — a memory unit — simulates a resettable storage that holds the most recent value satisfying a condition. Let $x_1, x_2, \ldots, x_T$ be a sequence of inputs, and suppose we define a binary signal r_i to indicate a "reset" at position i. The memory at time t should return the last input where $r_i = 1$:

$$m_t = x_{i^*}, \quad \text{where } i^* = \max\{i \leq t \mid r_i = 1\}$$

Transformers achieve this behavior using sparse attention. The query vector at time t is designed to match strongly with the last "reset" key seen earlier in the sequence. This forces the attention weights to place nearly all mass on a single relevant position. In practice, this can be implemented with positional bias or a learned key-query system that prioritizes recency and event-type matches.

Each of these modules — the modular counter and the memory unit — requires only a constant number of Transformer layers. Since a cascade of such modules can simulate any solvable semiautomaton, the entire automaton can be simulated in $O(\log T)$ depth. The depth and width of the network depend on the number of such components needed and on the size of the state space $|Q|$, but not on the input length T.

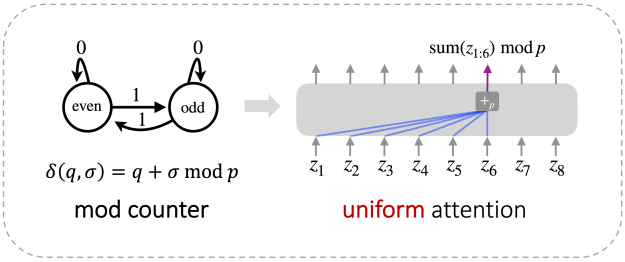

Figure 9: Uniform attention computes aggregate quantities like sums or parity. The attention weights are equal across positions, allowing prefix sums to be computed efficiently. These are then reduced modulo p through a feedforward MLP. This implements the modular counter component of the Krohn–Rhodes decomposition.

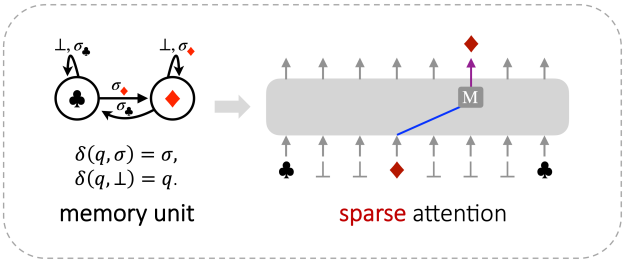

Figure 10: Sparse attention routes information from specific locations in the past. In a memory unit, the Transformer learns to attend to the last time a reset condition occurred. This attention pattern enables overwriting behavior consistent with reset monoids in the Krohn–Rhodes structure.

Theorem 5 (Depth-2 shortcut for gridworld; informal)**.** *For all positive integers n, T, Transformers can simulate* Grid_n *at length T, with depth 2, embedding dimension $O(1)$, attention width $O(n)$, and MLP width $O(T)$.*

Consider an automaton where a car moves along a circular track with four discrete positions, labeled $\mathbb{Z}_4 = \{0, 1, 2, 3\}$. The state of the system at any time is given by a pair (d, p), where $p \in \mathbb{Z}_4$ is the current position and $d \in \{\leftarrow, \rightarrow\}$ is the current direction of motion.

The input alphabet is $\Sigma = \{\texttt{D}, \texttt{U}\}$:

$\texttt{D}$ (drive) : move forward by one position in the current direction, $\texttt{U}$ (U-turn) : reverse direction

The key property of this automaton is that the transitions are non-commutative. Applying a drive followed by a U-turn leads to a different result than performing them in the reverse order:

$$\texttt{D} \circ \texttt{U} \neq \texttt{U} \circ \texttt{D}$$

This non-abelian behavior makes the full automaton appear sequential. Nonetheless, the final state can be computed in constant depth by decomposing the dynamics into two simpler and independent subroutines: one tracking the direction and the other tracking the signed displacement.

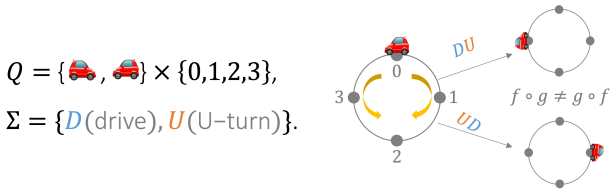

Figure 11: Car-on-a-circle automaton. The position evolves on $\mathbb{Z}_4$ and direction flips under $\texttt{U}$-turns. Transitions are non-commutative.

Let $\sigma_1, \ldots, \sigma_T \in \Sigma$ be the sequence of input symbols. Define:

$$U_t = \sum_{i=1}^{t} \mathbb{1}_{\sigma_i = \texttt{U}}$$

Then the direction at time t is:

$$d_t = d_0 \cdot (-1)^{U_t}$$

That is, the car flips direction with each U-turn. Determining the final direction only requires computing the parity of the number of U-turns, which is a modular counter over $\mathbb{Z}_2$.

Figure 12: Computing the parity of U-turns yields the final direction. This can be implemented using a uniform attention head followed by a mod-2 MLP.

With the final direction determined, the car's position depends only on how many `D` steps were taken and in which direction. Let $D_i = \mathbb{1}_{\sigma_i=\texttt{D}}$ and let $s \in \{-1, +1\}$ be the direction sign (negative for $\leftarrow$, positive for $\rightarrow$). The position is computed as:

$$p_T = \left(p_0 + s \cdot \sum_{i=1}^{T} D_i \right) \bmod 4$$

This is a signed modular sum over all drive instructions, scaled by the direction.

Figure 13: Given the direction, the final position is a signed sum of `D` steps modulo 4. This can also be computed using uniform attention followed by a mod-4 MLP.

Each of these computations — parity of U-turns and signed displacement — can be implemented with a single Transformer layer. The attention mechanism aggregates relevant tokens, and the MLPs perform the appropriate modular arithmetic. The full trajectory of this non-abelian automaton is therefore recoverable in $O(1)$ depth.

Theorem 6 (Transformer Barrington)**.** *Let $\mathcal{A}$ be a non-solvable semiautomaton. Then, for sufficiently large T, no $O(\log T)$-precision Transformer with depth independent of T and width polynomial in T can continuously simulate $\mathcal{A}$ at length T, unless $\mathsf{TC}^0 = \mathsf{NC}^1$.*

This lower bound follows by adapting Barrington's Theorem, which shows that width-5 branching programs of polynomial length can simulate any function in NC^1, but not beyond. As discussed earlier 1, this implies that simulating arbitrary non-solvable automata requires $\Omega(\log T)$ sequential steps in general.

In the context of Transformers, the result states that if $\mathcal{A}$ is non-solvable, then no shallow Transformer — specifically, one with constant depth and poly-size width — can continuously simulate $\mathcal{A}$ over inputs of length T, unless $\mathsf{TC}^0 = \mathsf{NC}^1$, which is widely conjectured to be false.

This shows that the shortcut constructions seen earlier fundamentally rely on the solvability of the automaton. For non-solvable cases (e.g., alternating groups like A_5), such compressions are impossible unless we allow logarithmic depth, or unless major complexity-theoretic collapses occur.

4 Empirical Results

What we've seen so far addresses the theoretical capabilities of Transformers for simulating semiautomata. The first construction showed that associativity enables a divide-and-conquer approach that leads to a $\log T$-depth simulator for any automaton. A more refined construction, based on the Krohn–Rhodes decomposition, gives an even more succinct simulator for solvable automata — built from modules for modular counting and memory.

However, these constructions assume a perfect circuit design. It is not obvious whether such shortcut solutions actually emerge when models are trained using standard optimization pipelines. This section explores that question empirically.

The experimental setup trains a Transformer to reproduce the state trajectory of a given semiautomaton, given an input sequence $\sigma_{1:T} \in \Sigma^T$. The model is trained to predict each q_t from the sequence $\sigma_{1:t}$ using a causally-masked

Transformer, as used in autoregressive models like GPT-2. Attention is masked to ensure that position t can only attend to earlier positions $\leq t$, preserving the causal structure.

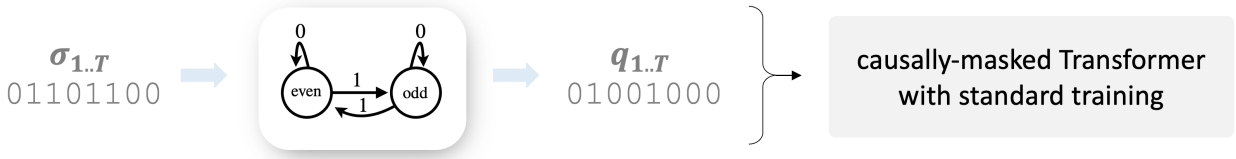

Figure 14: A causally-masked Transformer is trained on automaton trajectories. Each input sequence $\sigma_{1:T}$ is mapped to the corresponding state sequence $q_{1:T}$ via the automaton's transition rule.

4.1 Can shortcuts be found in practice?

The main empirical result is shown in Figure 15, which reports the test accuracy across 19 different automata and 16 different Transformer depths. The lighter the color, the better the accuracy. Each value represents the maximum performance across 20 random runs.

The upper rows of the heatmap correspond to automata with simpler structure, such as Dyck languages or cyclic groups, which can often be simulated with $O(1)$ layers. For these, high accuracy is achieved even with shallow models. In contrast, more complex automata such as the quaternion group Q_8, the alternating group A_5, and the symmetric group S_5 require significantly deeper Transformers. Notably, S_5 is a non-solvable group, and its simulation is provably hard under the assumptions stated in Theorem 6.

Another interesting observation is that the number of layers needed in practice appears to loosely correlate with the theoretical complexity of the automaton. However, the match isn't perfect. Even for automata where $O(1)$-layer shortcuts exist in theory, shallow models sometimes fail to reach high accuracy. This highlights issues like training instability or misalignment with the shortcut representation.

One particularly illuminating example is the 1D gridworld automaton 16. It consists of n states arranged linearly, where the agent starts from an initial state q_0 and receives inputs from the alphabet $\Sigma = \{L, R\}$ indicating a move to the left or right. The transition rule is clipped at the boundaries, and takes the form:

$$q_t = \max(1, \min(n, q_{t-1} + \sigma_t)), \quad \text{where } \sigma_t \in \{-1, +1\}$$

The model does not track the position through all steps. Instead, it learns to identify the last time a boundary state was reached 17 — that is, the latest t for which $q_t = 1$ or $q_t = n$. After this point, there is no more clipping, and the state evolves linearly as

$$q_T = q_{t^*} + \sum_{t > t^*} \sigma_t, \quad \text{where } t^* = \max\{t \leq T \mid q_t \in \{1, n\}\}$$

This means the prediction task reduces to computing a prefix sum, starting from a known q_{t^*}. Both steps — locating t^* and summing σ_t afterward — are parallelizable. The first is learned through sparse attention focused on the boundary hit, and the second through uniform attention that aggregates the remaining steps.

4.2 When Are Shortcuts Enough?

Shortcuts are appealing because they promise fast, shallow, and highly parallelizable solutions to simulating automata. They reduce the need for deep computation and can often match the theoretical bounds with fewer layers. But these benefits raise an important question: are shortcuts truly robust? What happens when supervision becomes sparse, or when the test sequences drift slightly from the training distribution? Understanding their limitations requires going beyond exact match accuracy and looking at how these solutions behave in more challenging settings.

While shallow shortcut solutions can match the target automaton under ideal conditions, they can also be fragile. To test their robustness, two modifications to the supervision setup were considered.

First, indirect supervision replaces the full-state supervision signal q_t with some function of q_t, like a binary label or projection. Second, incomplete supervision introduces sparsity: the ground truth q_t is revealed only with some probability p_{reveal} at each step. As p_{reveal} decreases, the learner must rely more on internal structure, rather than external feedback.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Dyck	99.3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Grid_4	99.9	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Grid_9	92.2	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
C_2	77.6	99.8	99.9	100	100	99.5	100	99.7	100	100	100	100	100	100	100	100
C_3	54.6	94.6	96.7	99.4	100	100	99.8	100	99.9	100	100	100	100	100	99.8	100
C_4	95.1	92.3	84.2	99.9	99.7	99.9	100	100	100	100	100	100	100	100	100	100
C_5	89.0	99.1	99.9	100	100	100	100	100	100	100	100	100	100	100	100	100
C_6	59.8	98.7	75.5	99.9	99.8	99.9	99.9	100	100	100	99.8	99.9	100	99.8	99.9	99.9
C_7	90.9	95.0	99.9	99.9	100	99.9	100	100	100	100	100	99.8	100	100	100	100
C_8	79.6	96.2	99.8	99.8	99.9	100	99.9	99.9	100	99.4	99.9	99.9	99.9	100	99.9	99.9
C_2^2	90.5	98.8	99.9	100	100	99.9	100	100	99.9	99.9	100	100	100	100	100	100
C_2^3	65.0	77.9	99.9	97.9	100	99.8	98.2	99.9	100	100	91.9	95.9	91.7	90.6	87.5	80.6
D_6	25.4	27.2	47.4	75.2	100	100	100	100	100	100	100	100	100	100	100	100
D_8	45.6	98.0	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Q_8	31.6	49.2	59.6	60.4	73.5	99.3	100	100	100	100	100	100	100	100	100	100
A_4	25.0	35.4	49.1	59.3	62.6	82.3	90.9	98.0	98.0	99.1	99.8	100	99.7	100	100	100
A_5	12.5	23.1	32.5	46.7	71.2	98.8	100	100	100	100	100	100	100	100	100	100
S_4	11.3	17.6	22.0	27.1	37.7	44.8	50.8	72.5	91.3	97.1	97.9	98.7	99.9	100	99.8	99.9
S_5	7.9	11.8	14.6	19.7	26.0	28.4	32.8	51.8	86.3	94.8	90.2	97.2	99.3	99.1	99.9	99.9

Figure 15: Test accuracy (color-coded) for different automata $\mathcal{A}$ (rows) across Transformer depths (columns). Lighter shades correspond to better performance. Each result is the best of 20 random trials.

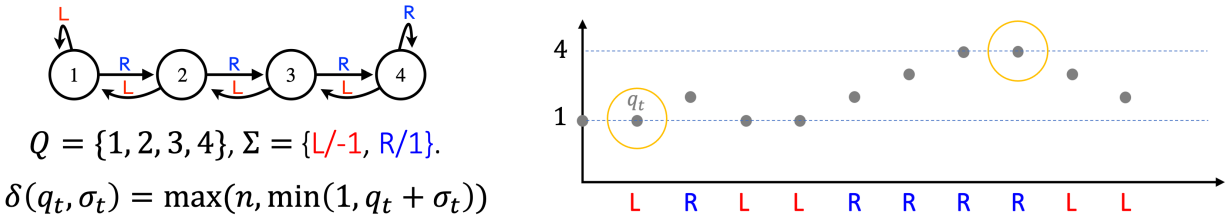

Figure 16: A learned $\mathcal{O}(1)$-layer shortcut to simulate gridworld. Despite the combinatorial complexity, the Transformer recovers a much simpler boundary-aware strategy.

Under indirect supervision, Transformers are able to learn stable and accurate simulations. However, when supervision becomes sparse, performance drops sharply, especially on harder tasks like symmetric group S_5. In contrast, Long Short-Term Memory (LSTMs) remain robust across all p_{reveal} values. This is likely due to the recurrent nature of LSTMs, which inherently propagate memory and structure over time.

Another axis of brittleness comes from distributional shift. When trained on sequences of fixed length and balanced statistics—such as parity tasks with $p(\sigma_t = 1) = 0.5$—Transformers perform well within that regime. But when tested on sequences with a different number of 1s (e.g., more or fewer than 20), performance degrades, particularly at the edges of the distribution. This suggests that the model leans on shallow positional correlations that break outside the training distribution.

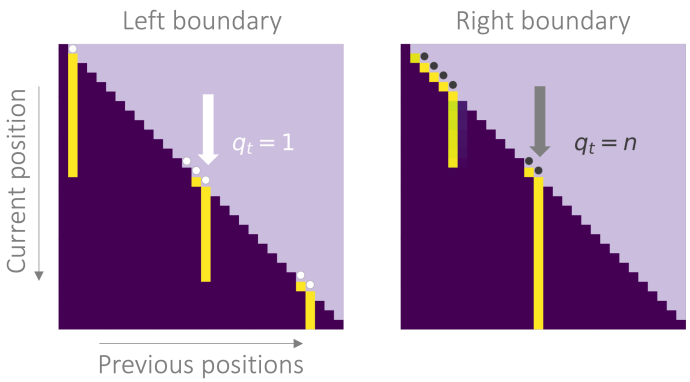

Figure 17: Attention map from the final layer. The yellow region shows that the Transformer sharply attends to the last boundary state, where $q_t = 1$ or $q_t = n$. This enables exact computation of the final position.

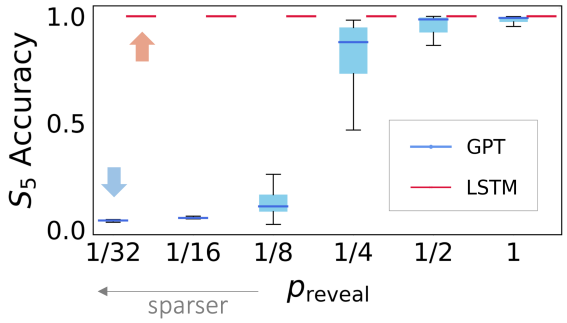

Figure 18: Transformer accuracy drops as supervision becomes sparse; LSTM stays robust due to recurrent structure.

By contrast, shifted positional encodings—like jittered embeddings or learned positions—help stabilize the output, showing that part of the failure is due to overly rigid reliance on position encoding.

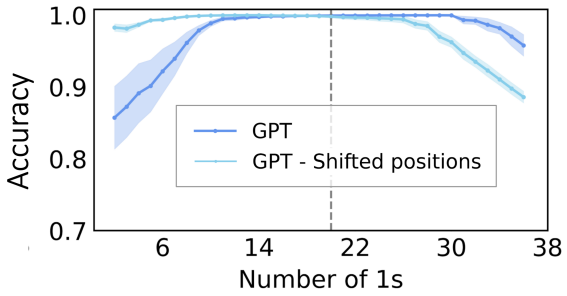

Figure 19: Accuracy on the parity task drops for sequences with out-of-distribution number of 1s.

5 Discussion

The results explored throughout this report highlight both the expressive power and the brittleness of Transformer circuits when used to simulate automata. The theoretical constructions show that deep-seeming behaviors can often be flattened into shallow, parallelizable architectures — particularly when the automaton admits an algebraic shortcut such as associativity or Krohn–Rhodes decomposition.

Transformers are especially good at exploiting structure when it exists. In the case of parity, mod counters, gridworld navigation, or even carefully structured group actions, attention and MLPs can simulate the entire state evolution with $\mathcal{O}(1)$ to $\mathcal{O}(\log T)$ depth. These results suggest that the underlying architecture, although non-recurrent, is well-matched to sequential reasoning when the problem is compositional and compactly representable.

At the same time, the empirical section reveals important caveats. Models trained in standard ways don't always converge to the clean, theoretically optimal shortcuts — particularly when supervision is sparse or biased, or when input distributions shift. In the parity task, even a small deviation from training distribution (e.g., changing the number of 1s) leads to significant performance drops. This reflects a sensitivity to positional encodings, and perhaps an over-reliance on statistical regularities present in training data rather than structural generalization.

In terms of architectural robustness, LSTMs tend to generalize more gracefully under limited feedback, owing to their built-in recurrence and inductive bias. In contrast, Transformer shortcuts appear brittle unless trained with full supervision and good positional diversity. This highlights an important distinction: while Transformers can learn shortcuts, they don't necessarily prefer them unless incentivized to do so.

The authors of the original paper also point to several future directions that remain open:

- Understanding the inductive biases that cause Transformers to prefer or avoid shortcut solutions under different training regimes.
- Exploring whether architectural modifications, such as memory-augmented Transformers or hybrid recurrent-attentive models, could stabilize generalization under distribution shift.
- Investigating whether pre-training on broader data distributions encourages more robust automaton simulation across tasks.
- Studying how shortcut-learning behaviors relate to compositional generalization in more complex domains like programming, language, and planning.

These are promising areas, especially given how relevant automata-theoretic reasoning is for problems like parsing, logic, and formal verification.

6 Conclusion

This report was a review of the work *Transformers Learn Shortcuts to Automata*. The central idea was that although simulating automata appears sequential, many such computations admit shallow parallel solutions that Transformers are theoretically capable of learning. We examined four core theorems that showed how associativity, solvability, and structural decomposition enable depth-efficient simulation.

We also looked at how such solutions arise during training, identifying when they are recovered in practice and when they fail. From brittle behavior under limited supervision to failure under OOD conditions, the results showed that architectural capability alone doesn't guarantee robust learning.

Nonetheless, the overall takeaway is optimistic: when guided correctly, Transformers can serve not just as sequence models, but as shortcut solvers for deep algorithmic tasks. The intersection of automata theory and deep learning continues to offer deep insight into what these models learn — and how far they can go.

References

[1] Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, volume 30, 2017.

[3] Raymond Greenlaw and H. James Hoover. Chapter 9 - circuit complexity. In Raymond Greenlaw and H. James Hoover, editors, *Fundamentals of the Theory of Computation: Principles and Practice*, pages 241–257. Morgan Kaufmann, Oxford, 1998.

[4] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 1–5, 1986.

[5] Bingbin Liu, Jordan Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata, 2022. `https://clarabing.github.io/shortcut_automata/`.