

This thesis was submitted to the Chair of Machine Learning and Reasoning.

# **Search Transformers: Latent Reasoning via Differentiable Search**

Master Thesis

**Presented by**

Nabin Bhattarai

425181

**Supervised by** Carlos Núñez Molina, Ph.D.

**1<sup>st</sup> Examiner** Prof. Hector Geffner, Ph.D.

**2<sup>nd</sup> Examiner** Prof. Dr. rer. nat. Christopher Morris

Aachen, September 3, 2026



# Abstract

Solving reasoning problems often requires performing a sequence of dependent computations, and models trained on shallow instances may fail to extrapolate to problems requiring more computation steps. Depth-recurrent Transformers present a potential approach: by iteratively applying the same Transformer blocks, they can scale computation at inference time, which may support depth extrapolation. In this thesis, we investigate whether organizing this latent recurrence in a search-like manner improves task learning and extrapolation to unseen problem depths. Building on the conceptual Search Transformer (ST) proposal by Núñez-Molina et al. [22], we develop a concrete, trainable architecture that structures computation as a differentiable sequence of select-expand-append steps. The problem description and query remain fixed as input vectors, while the model appends one latent search node at each step. We compare ST with a similarly sized Universal Transformer (UT), which instead refines a fixed-size collection of representations, on logical-deduction and path exact-cover decision problems. Both models use predefined input encodings and are trained using only final decision labels.

ST’s mean out-of-distribution (OOD) accuracy exceeds that of UT by 5.7, 11.6, and 11.0 percentage points on the easy, medium, and hard logical-deduction datasets, respectively. On the hard logical-deduction dataset, ST shows better sample efficiency, requiring a median of 3.84 million training examples across runs to reach a sustained validation accuracy of at least 90%, compared to 53.76 million for UT. However, the performance advantage does not transfer to the path exact-cover datasets. On the Convergent and State-Alias path exact-cover profiles, ST’s OOD accuracy lags behind UT by 8.2 and 1.9 percentage points, respectively. Nevertheless, some low-accuracy ST runs maintain high AUROC, suggesting that the two classes remain well separated, but an OOD logit distribution shift degrades classification accuracy under the fixed decision threshold. On the Disjoint profile, all five ST runs reach the final training stage and achieve 89.7% mean OOD accuracy, whereas only one of the five UT runs reaches that stage but achieves a higher OOD accuracy of 95.0%. Thus, ST shows more reliable convergence while settling on a poorer solution than the converged UT run. Overall, the results show that ST’s search-structured recurrence can provide an advantage on logical deduction tasks, but the same performance advantage does not carry over to path exact-cover.

# Contents

|          |                                                              |           |
|----------|--------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                          | <b>1</b>  |
| <b>2</b> | <b>Background</b>                                            | <b>4</b>  |
| 2.1      | Problem Solving and Reasoning as Search . . . . .            | 4         |
| 2.1.1    | Solving Decision Problems . . . . .                          | 4         |
| 2.2      | Transformer Architecture . . . . .                           | 7         |
| 2.2.1    | Transformer Blocks and Attention . . . . .                   | 8         |
| 2.2.2    | Sequential Computation in fixed-depth Transformers . . . . . | 10        |
| 2.2.3    | Latent Recurrent Transformers . . . . .                      | 11        |
| 2.3      | Multiple Instance Learning (MIL) . . . . .                   | 13        |
| 2.3.1    | Instance Space (IS) Paradigm . . . . .                       | 14        |
| 2.3.2    | Embedded Space (ES) Paradigm . . . . .                       | 15        |
| 2.4      | Adaptive Computation . . . . .                               | 15        |
| 2.4.1    | PonderNet . . . . .                                          | 16        |
| 2.5      | Memory Pointers . . . . .                                    | 17        |
| 2.6      | Curriculum Learning . . . . .                                | 18        |
| 2.7      | Reasoning Tasks . . . . .                                    | 19        |
| 2.7.1    | Logical Deduction . . . . .                                  | 19        |
| 2.7.2    | Edge-Labeled Path Exact Cover . . . . .                      | 20        |
| <b>3</b> | <b>Related Work</b>                                          | <b>21</b> |
| 3.1      | Recurrent Computation with Transformers . . . . .            | 21        |
| 3.2      | Adaptive Computation and Learned Halting . . . . .           | 23        |
| 3.3      | Learning to Plan . . . . .                                   | 24        |
| 3.4      | Reasoning with Large Language Models (LLMs) . . . . .        | 25        |
| 3.5      | Summary . . . . .                                            | 26        |
| <b>4</b> | <b>Methods</b>                                               | <b>28</b> |
| 4.1      | Problem Formulation . . . . .                                | 28        |
| 4.1.1    | Problem Representation . . . . .                             | 28        |
| 4.2      | Overview of Notations Used . . . . .                         | 30        |
| 4.3      | Search Transformer Architecture . . . . .                    | 30        |
| 4.3.1    | Memory Layout . . . . .                                      | 30        |
| 4.3.2    | Search Loop . . . . .                                        | 32        |

|          |                                                     |           |
|----------|-----------------------------------------------------|-----------|
| 4.3.3    | Start Node Initialization . . . . .                 | 32        |
| 4.3.4    | Node Selection . . . . .                            | 32        |
| 4.3.5    | Action Selection as Memory Pointer . . . . .        | 33        |
| 4.3.6    | Candidate Expansion and Aggregation . . . . .       | 35        |
| 4.3.7    | Update Action Usage . . . . .                       | 37        |
| 4.3.8    | Readout and Pooling Operation . . . . .             | 39        |
| 4.4      | Adaptive Computation for ST . . . . .               | 40        |
| 4.4.1    | Halting Head . . . . .                              | 41        |
| 4.4.2    | KL Regularization . . . . .                         | 42        |
| 4.4.3    | Prediction Using Search Prefixes . . . . .          | 42        |
| 4.4.4    | Halting During Inference . . . . .                  | 42        |
| 4.5      | Learning Objectives . . . . .                       | 43        |
| 4.5.1    | Task-Level Supervision . . . . .                    | 43        |
| 4.5.2    | Dynamic Halting Objective . . . . .                 | 44        |
| 4.6      | Training . . . . .                                  | 44        |
| 4.6.1    | Curriculum Learning . . . . .                       | 44        |
| 4.6.2    | Scaling Search Horizon . . . . .                    | 45        |
| 4.6.3    | Entropy Regularization for Node Selection . . . . . | 45        |
| <b>5</b> | <b>Results</b>                                      | <b>46</b> |
| 5.1      | Experimental Setup . . . . .                        | 46        |
| 5.1.1    | Benchmark Configuration . . . . .                   | 47        |
| 5.1.2    | Compared Models . . . . .                           | 48        |
| 5.1.3    | Evaluation Metrics . . . . .                        | 48        |
| 5.1.4    | Training and Optimization . . . . .                 | 49        |
| 5.1.5    | Hyperparameters . . . . .                           | 50        |
| 5.2      | Performance Comparison . . . . .                    | 50        |
| 5.2.1    | Overall ID and OOD Performance . . . . .            | 50        |
| 5.2.2    | Logical Deduction: LD Hard . . . . .                | 51        |
| 5.2.3    | Path Exact-Cover . . . . .                          | 56        |
| 5.3      | Effect of Inference-Time Computation . . . . .      | 62        |
| 5.3.1    | Fixed-Step Search Budget . . . . .                  | 62        |
| 5.3.2    | Adaptive Computation . . . . .                      | 64        |
| 5.4      | Ablation Studies . . . . .                          | 65        |
| 5.4.1    | Node Selection and Action Representation . . . . .  | 65        |
| 5.4.2    | Readout Paradigm . . . . .                          | 68        |
| <b>6</b> | <b>Conclusion</b>                                   | <b>69</b> |
| 6.1      | Summary . . . . .                                   | 69        |
| 6.2      | Limitations and Future Work . . . . .               | 70        |

|                                                                         |            |
|-------------------------------------------------------------------------|------------|
| <b>A Appendix</b>                                                       | <b>71</b>  |
| A.1 Reasoning Problems . . . . .                                        | 71         |
| A.1.1 Logical Deduction . . . . .                                       | 71         |
| A.1.2 Path Exact-Cover Problem . . . . .                                | 75         |
| A.2 Supplementary Experimental Setup . . . . .                          | 81         |
| A.2.1 Model Configuration . . . . .                                     | 81         |
| A.2.2 Implementation Environment . . . . .                              | 81         |
| A.3 Supplementary Experimental Results . . . . .                        | 83         |
| A.3.1 Depth-wise Logical-Deduction Results . . . . .                    | 83         |
| A.3.2 Additional Fixed-Budget Results for LD Hard . . . . .             | 83         |
| A.3.3 Additional Fixed-Budget Results for Path Exact-Cover . . . . .    | 84         |
| A.3.4 Seed Sensitivity on the Convergent Profile . . . . .              | 84         |
| A.3.5 Additional Per-Seed Results for the State-Alias Profile . . . . . | 84         |
| A.3.6 Node-Selection Ablation on the State-Alias Profile . . . . .      | 88         |
| A.4 Search Transformer with Token Inputs . . . . .                      | 89         |
| A.4.1 General ST Architecture . . . . .                                 | 89         |
| A.4.2 Model Configuration . . . . .                                     | 90         |
| A.4.3 Dataset . . . . .                                                 | 90         |
| A.4.4 Training Objective . . . . .                                      | 91         |
| A.4.5 Results . . . . .                                                 | 93         |
| <b>List of Acronyms</b>                                                 | <b>95</b>  |
| <b>List of Symbols</b>                                                  | <b>96</b>  |
| <b>List of Figures</b>                                                  | <b>99</b>  |
| <b>List of Tables</b>                                                   | <b>100</b> |
| <b>List of Algorithms</b>                                               | <b>101</b> |
| <b>List of References</b>                                               | <b>102</b> |

# 1 Introduction

*“I do not see how thinking could fundamentally be anything other than a kind of trial action carried out only in the brain, in an imagined space.”*

— Konrad Lorenz, *Die Rückseite des Spiegels* (1973)

Many problems require composing information over multiple sequential steps to make a decision. Examples include applying multiple logical rules to determine whether a fact is provable. We use the term “depth” to refer to a task-specific proxy for this sequential complexity. In this thesis, we study a neural network architecture for solving decision problems. Importantly, we study whether a model trained on shallower problems can reuse its learned computation when extrapolating to unseen and larger problem depths. Since problems with larger depths generally require more sequential computation, we focus on parameter-shared recurrent architectures for which the amount of sequential computation can be increased without increasing their parameter count.

Parameter-shared recurrent architectures like Universal Transformers (UTs) [7] and Looped Transformers (LTs) [11] have emerged as parameter-efficient alternatives to standard Transformers for solving reasoning problems [28, 34]. Generally referred to as latent recurrent models, these architectures iteratively refine a collection of hidden representations using parameter-shared Transformer blocks. The information accumulated across the recurrent steps remains in the model’s continuous hidden state. This differs from the Chain-of-Thought (CoT) [33] reasoning approach, which represents the intermediate outputs, such as partial results, as discrete tokens.

This thesis studies an alternative latent recurrent architecture, called Search Transformer (ST), based on the proposal by Núñez-Molina et al. [22]. The proposal defines a high-level search organization, but leaves several architectural choices unspecified and provides no empirical evaluations. Unlike Universal Transformer (UT), which jointly refines a collection of representations, Search Transformer (ST) maintains a fixed encoding of the problem and an expanding collection of latent nodes. Conceptually, each recurrent step selects a previous node, retrieves an action from problem encodings, generates a successor from the resulting parent–action pair, and appends the successor as a new node. Node selection can use global information, but we assume the transition to have Markov property: each candidate successor depends only on its parent node and action, and not on the preceding search trajectory. This gives ST a stricter search-like inductive bias than UT. For problems whose solutions can be represented as sequences of local transitions, we hypothesize that this inductive bias makes

learning the solution easier from data, potentially improving sample efficiency and training reliability. Furthermore, reusing the learned computation for more steps may also improve extrapolation to problems at unseen depths. UT serves as the primary baseline since it performs latent reasoning using shared parameters but differs in how these internal computations are organized. This comparison therefore tests whether the additional search-like inductive bias of ST improves learning and depth extrapolation.

We retain the high-level select–expand–append search organization from [22], but we make changes to the action, transition, retry, and readout mechanisms. We represent each action as a soft pointer to problem memory. We simplify the transition function using a gated residual update: a Multilayer Perceptron (MLP) computes an update from the parent–action pair, which is modulated by a sigmoid gate and added to the parent node to obtain the candidate successor. We also incorporate an availability mechanism that reduces the availability of previously used node–memory pairs, encouraging the model to select a different memory element when a node is revisited. We also modify the readout function to first score each generated node against the query and then pool these scores to obtain the final prediction. We implement the node selection as a differentiable relaxation in which we expand all frontier nodes and append the weighted combination of their candidate successors using the selection weights. Thus, the computation is organized as a search without constructing a discrete search tree.

To study the advantage of ST, especially the search-like internal computations, we restrict our study to structured binary decision problems. The problem context and query are presented as vector embeddings using a predefined encoder, and the models are trained using only the final decision label, without supervision of intermediate states or search trajectories. This enables us to compare the latent computation performed by ST and UT without additionally requiring the models to parse the natural language inputs or generate answers autoregressively.

This thesis addresses the following questions:

1. How does ST compare with a similarly sized UT in terms of in-distribution performance and extrapolation to unseen problem depths?
2. How do fixed and adaptive computation affect the performance of ST, and how does its learned halting behavior vary with problem depth?
3. How do the action representation, node-selection policy, and readout mechanism affect the learning and performance of ST?

We investigate these questions on two synthetically generated dataset families: logical deduction and path exact-cover (Path-EC). Logical deduction is monotonic: once an atom is derived, it remains available for subsequent deductions, and different derivation paths that produce the same set of atoms correspond to the same logical state. Path-EC is instead path-dependent: a search state must represent both the current graph vertex and the items collected along the path to that vertex. Thus, reaching the same vertex via different paths leads to different search

states. Hence, the two task families allow us to evaluate whether the effectiveness of ST extends across structurally different reasoning problems or depends on a specific problem structure.

Our findings indicate that ST is capable of learning tasks from both families, but does not consistently outperform UT across them. On all three logical-deduction datasets, ST achieves higher average OOD performance than UT. On LD Hard, ST demonstrates better sample efficiency under our training framework: it attains 90% validation accuracy after using a median of 3.84 million training instances, compared with 53.76 million for UT, a 14-fold difference. On Path-EC, ST learns non-trivial solutions across all three profiles; however, UT attains higher OOD performance in each case. In the Convergent and State-Alias profiles, several ST runs maintain high AUROC despite lower accuracy, suggesting that part of the OOD gap results from the fixed classification threshold being poorly calibrated on OOD examples. The Disjoint profile indicates a different trade-off: all five ST runs finish the curriculum and converge to similar but lower accuracies, whereas only one of the five UT runs completes the curriculum and achieves higher ID and OOD accuracy. Thus, ST is more reliable at completing training on the Disjoint profile, but the successful UT run learns a better solution. Increasing the inference-time search budget improves performance on LD Hard but not on Disjoint, indicating that the benefit of additional computation depends on the computation learned for the task.

This thesis makes the following contributions:

- A concrete and trainable adaptation of the conceptual ST proposal [22] for structured decision problems, with components such as an action pointer, a readout paradigm, and dynamic halting.
- Multiple controlled comparisons with UT across two task families.
- Ablation studies of the action representation, node selection, and readout, together with analyses of inference-time computation.

Chapter 2 formulates the decision problems as state-space search and introduces components and paradigms used in this work, such as Transformers, Multiple Instance Learning (MIL), and dynamic halting. We also define the two reasoning task families used for evaluation. In Chapter 3, we position Search Transformer relative to the original proposal, latent recurrent Transformers, learned planning methods, and CoT reasoning with LLMs. Chapter 4 then makes the architecture concrete by defining its search components, readout, learning objectives, and adaptive computation. Chapter 5 first compares ST with UT on problems at depths seen during training (ID) and at unseen depths (OOD), and then studies inference-time computation and architectural ablations. Finally, Chapter 6 presents the main findings relative to the research questions and discusses the limitations and possible directions for future work.

## 2 Background

This chapter introduces the core concepts needed to define Search Transformer and tasks used to evaluate it. We first formulate the studied decision problems as state-space reachability problems and outline an idealized bounded search procedure. This symbolic formulation motivates the structuring of ST, but its internal search process should be understood as a differentiable relaxation rather than an exact implementation. We then introduce Transformer-based latent recurrence and differentiate between the fixed-state and expanding-state models. The remaining sections cover the readout, adaptive halting, memory access, and curriculum mechanisms used in the thesis. Finally, we define the two task families used to assess the models.

### 2.1 Problem Solving and Reasoning as Search

We adopt the state-space search as an abstraction for problem-solving and reasoning tasks that can be expressed with an initial state, operations, and a goal condition.

Formally, we define the reasoning problem as a tuple  $\langle \mathcal{S}, \mathcal{A}, s_0, S_g, \mathcal{T} \rangle$ :

- $\mathcal{S}$  (State Space): A set of states. A state contains the information required to determine its successor and whether the goal condition is satisfied.
- $\mathcal{A}$  (Actions): A set of available actions.
- $s_0 \in \mathcal{S}$  (Initial State): Initial state.
- $S_g \subseteq \mathcal{S}$  (Goal State): A set of goal states.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$  (Transition Function): A function that maps a state-action pair to a successor state.

#### 2.1.1 Solving Decision Problems

We adapt the state-space search formulation for modeling the decision problems. Starting from the initial state  $s_0$ , an action sequence  $a_0, \dots, a_{k-1}$  induces a state trajectory  $(s_0, s_1, \dots, s_k)$  defined by

$$s_{i+1} = \mathcal{T}(s_i, a_i), \quad i = 0, \dots, k-1.$$

Then, the decision label is:

$$y = \begin{cases} 1, & \text{if there exist } k \geq 0 \text{ and } a_0, \dots, a_{k-1} \in \mathcal{A} \text{ such that } s_k \in S_g, \\ 0, & \text{otherwise.} \end{cases}$$

Thus,  $y = 1$  if at least one search trajectory reaches a goal state, whereas  $y = 0$  means that no such trajectory exists.

**Search Process.** The decision problem formulation above defines whether a solution exists, but it does not specify how a search procedure finds a goal state. Here, we describe the general search process using a persistent frontier.

In classical search, the frontier generally contains states that have been generated but not yet expanded. Once a state is expanded, it is removed from the frontier. Here, we define a *persistent frontier*. In our search formulation, an expansion applies one selected action to one state. Therefore, expanding a state does not imply that all actions from that state have been tried. We keep the expanded state in the frontier so that it can be selected again and expanded with a different action. Throughout our search formulation, the term *frontier* refers to this *persistent frontier* unless otherwise stated.

Let  $N_t = [n_0, \dots, n_{t-1}]$  denote the ordered persistent frontier available at the beginning of search step  $t \geq 1$ , where  $n_i \in \mathcal{S}$  is the state stored at position  $i$ . Initially,

$$N_1 = [n_0], \quad n_0 = s_0, \quad U_1 = \emptyset.$$

At step  $t$ , the search process selects an unused node–action pair:

$$(i_t, a_t) \in (\{0, \dots, |N_t| - 1\} \times \mathcal{A}) \setminus U_t,$$

where  $i_t$  identifies the selected node and  $U_t$  contains the set of node index and action pairs already expanded.

It then generates and appends a successor:

$$\begin{aligned} n_t &= \mathcal{J}(n_{i_t}, a_t), \\ N_{t+1} &= N_t \parallel [n_t], \\ U_{t+1} &= U_t \cup \{(i_t, a_t)\}. \end{aligned}$$

Here,  $\parallel$  denotes sequence concatenation.

The above search procedure defines an idealized discrete search process. In ST, we represent the persistent frontier with a growing list of latent nodes, and replace the hard node–action selection with a differentiable relaxation.

**Running Example: Logical Deduction.** We use the following logical-deduction instance as a running example.

|        |                                                   |
|--------|---------------------------------------------------|
| Facts: | $A, C$                                            |
| Rules: | $A \rightarrow B, \quad B \wedge C \rightarrow D$ |
| Query: | $D?$                                              |
| Label: | 1                                                 |

The task is to decide whether the atom  $D$  can be derived from the given facts and rules. We map this logical-deduction instance to the state-space search formulation as follows.

- State space:  $\mathcal{S} = 2^{\{A,B,C,D\}}$
- Actions:  $\mathcal{A} = \{A \rightarrow B, B \wedge C \rightarrow D\}$
- Start state:  $s_0 = \{A, C\}$
- Goal States:  $S_g = \{s \in \mathcal{S} \mid D \in s\}$
- Transition function: For a state  $s \in \mathcal{S}$  and an action  $a \in \mathcal{A}$ , we define  $\mathcal{T}(\cdot, \cdot)$  as:

$$\mathcal{T}(s, a) = \begin{cases} s \cup \{B\}, & \text{if } a = (A \rightarrow B) \text{ and } A \in s, \\ s \cup \{D\}, & \text{if } a = (B \wedge C \rightarrow D) \text{ and } \{B, C\} \subseteq s, \\ s, & \text{otherwise.} \end{cases}$$

Here, a state is represented by the set of atoms currently known to be true. For simplicity, each action is denoted by the rule it applies. If the rule's premises are satisfied in the state, the transition function adds its conclusion; otherwise, the state remains unchanged.

Now, we can solve the problem by performing search in the state space. Below, we show one possible search trajectory that reaches the goal.

1. Initialize the frontier as  $N_1 = [s_0]$ , where  $s_0 = \{A, C\}$ .
2. Select  $(s_0, A \rightarrow B)$  and generate  $s_1 = \mathcal{T}(s_0, A \rightarrow B) = \{A, B, C\}$ . Then,  $N_2 = N_1 \parallel [s_1]$ .
3. Select  $(s_1, B \wedge C \rightarrow D)$  and generate  $s_2 = \mathcal{T}(s_1, B \wedge C \rightarrow D) = \{A, B, C, D\}$ .  
Then,  $N_3 = N_2 \parallel [s_2]$ .
4. Since  $s_2 \in S_g$ , this trajectory proves  $y = 1$ .

The trajectory above shows that the goal is reachable, but it leaves out the detail of the search strategy used. For example, given the frontier  $N_2 = [s_0, s_1]$ , how is the next state-action pair selected for expansion? This strategy becomes important when the search has to operate under a limited computation budget.

**Bounded Search.** For a finite state space, an exhaustive search over all reachable states can always determine if the goal is reachable from  $s_0$ . However, exploring all reachable states may require more computation than is available. Therefore, we consider search procedures that

operate under a predefined search budget  $B$ , which sets an upper bound on the search effort. The nature of this budget depends on the search procedure. For example, it may limit the number of state–action expansions or the search iterations. In ST, the budget is represented by the maximum search horizon  $T$ , measured in recurrent search steps.

Let  $V_B$  denote the set of states generated using a fixed search strategy before the search budget is exhausted. Then, we define the output of the bounded search as:

$$\hat{y}_B = \mathbb{1}[V_B \cap S_g \neq \emptyset].$$

Then

$$\hat{y}_B = 1 \Rightarrow y = 1, \quad \hat{y}_B = 0 \not\Rightarrow y = 0,$$

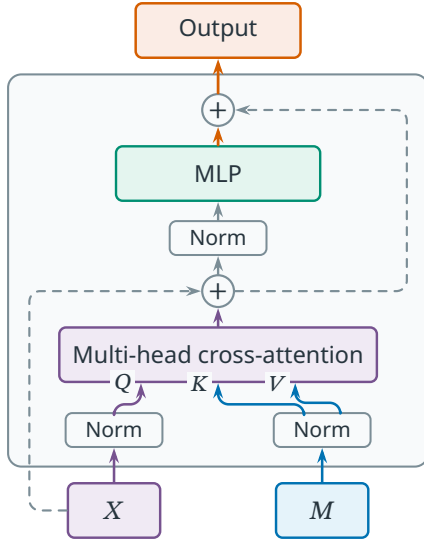
So, finding a goal state proves that the problem has a positive label. However, failing to find one only means that the search did not reach a goal state within its budget; but does not prove that the goal is unreachable. Also, the above result only holds when the transition and goal test are exact. ST instead learns a latent transition and goal test, so the above guarantee does not fully translate to ST.

**Heuristic Search.** The set of states visited within a given budget depends on which state–action pairs the search chooses to expand. Since bounded search can consider only a limited number of expansions, these choices determine whether a reachable goal is found before the budget is exhausted. For example, the logical-deduction formulation above contains  $2^n$  possible states for  $n$  logical atoms. Although only a fraction of these states may be reachable, exhaustively exploring them can still require more computation than the available budget permits.

A heuristic estimates which state–action pairs are more promising and prioritizes them during search. With a fixed budget, a useful heuristic increases the likelihood of reaching the goal.

## 2.2 Transformer Architecture

In this section, we introduce the Transformer [31] components used to implement ST and explains how they support iterative computation. We first define attention and the Transformer block. We then examine the sequential computation available in a fixed-depth Transformer and describe how repeated application of shared components can extend it. Finally, we distinguish latent recurrent models based on whether their state size remains fixed or grows across recurrent steps.



(a) A cross-attention Transformer block.

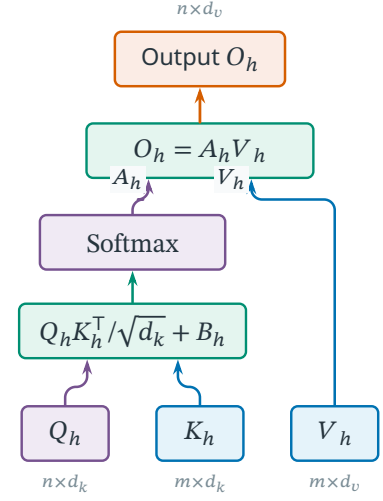
(b) Scaled dot-product attention computed at an attention head  $h$ .

Figure 2.1: The components of a Transformer. A Transformer block combines multi-head attention with residual connections, normalization, and a feed-forward network. Each attention head converts query-key compatibility scores into weights  $A_h$  used to combine the values.

### 2.2.1 Transformer Blocks and Attention

Let  $\mathcal{V} \subseteq \mathbb{R}^{d_m}$  denote the representation space of the Transformer, where  $d_m$  is the model dimension. We define a cross-attention Transformer block as a parameterized function

$$f_\theta : \mathcal{V}^n \times \mathcal{V}^m \rightarrow \mathcal{V}^n.$$

The Transformer block updates a sequence of input representations ( $X \in \mathcal{V}^n$ ) using a memory ( $M \in \mathcal{V}^m$ ) while preserving the sequence length and model dimension of  $X$ . Figure 2.1 provides an overview of the Transformer block and the attention operation computed within each attention head.

**Attention heads.** An attention head updates each input representation by assigning weights to the memory representations and combining their projected values. Let  $X \in \mathcal{V}^n$  contain  $n$  input representations and let  $M \in \mathcal{V}^m$  contain  $m$  memory representations. We denote the query-key dimension by  $d_k$  and the value dimension by  $d_v$ . Now, we define a single attention

head  $h \in \{1, \dots, H\}$ , where  $H$  is the number of attention heads, as:

$$\begin{aligned} Q_h &= XW_h^q && \in \mathbb{R}^{n \times d_k}, \\ K_h &= MW_h^k && \in \mathbb{R}^{m \times d_k}, \\ V_h &= MW_h^v && \in \mathbb{R}^{m \times d_v}, \\ S_h &= \frac{Q_h K_h^\top}{\sqrt{d_k}} + B_h && \in \mathbb{R}^{n \times m}, \\ A_h &= \text{Softmax}_{\text{row}}(S_h) && \in \mathbb{R}^{n \times m}, \\ O_h &= A_h V_h && \in \mathbb{R}^{n \times d_v}. \end{aligned}$$

where,

$$W_h^q, W_h^k \in \mathbb{R}^{d_m \times d_k}, \quad W_h^v \in \mathbb{R}^{d_m \times d_v}.$$

are the parameters of the attention head  $h$ .

The entry  $[S_h]_{r,j}$  measures the scaled compatibility between the representation of input at position  $r$  and memory at position  $j$ . The Softmax is applied row-wise, and it converts these scores into nonnegative weights  $[A_h]_{r,j}$  that sum to one over the memory positions. We obtain the output for each query position  $r$  as

$$[O_h]_{r,:} = \sum_{j=1}^m [A_h]_{r,j} [V_h]_{j,:},$$

which is a convex combination of the projected memory values where the weights  $[A_h]_{r,j}$  are input and memory dependent. Here, we used the notation  $[\cdot]_{i,j}$  to represent the element of row  $i$  and column  $j$  of the input matrix.

In the above equations,  $B_h \in \mathbb{R}^{n \times m}$  is an optional attention mask that assigns  $-\infty$  (or sufficiently negative value in practice) to query-key pairs that should not contribute to the output computation. If no mask is required, we set  $B_h = \mathbf{0}$ .

Since we obtain queries from  $X$  and key-values from  $M$ , this formulation describes cross-attention operation. We obtain self-attention operation for an input  $X$  by setting  $M = X$ .

**Multi-head attention.** A single attention head computes the input-dependent convex combination of memory representation in one learned projection space. Multi-head attention applies multiple independent attention heads in parallel where each head uses independently learned query, key, and value projections. It can therefore compare the input and memory in a different projection space and produce a different output. Here we define a *Multi-Head*

Attention with  $H$  heads as:

$$\begin{aligned} C &= \text{Concat}(O_1, \dots, O_H) && \in \mathbb{R}^{n \times Hd_v}, \\ \text{MHA}(X, M; \theta_{\text{attn}}) &= CW_o && \in \mathbb{R}^{n \times d_m}, \end{aligned}$$

with

$$W_o \in \mathbb{R}^{Hd_v \times d_m}, \quad \theta_{\text{attn}} = \{W_o, \theta_1, \dots, \theta_H\},$$

where

$$\theta_h = \{W_h^q, W_h^k, W_h^v\}.$$

We obtain  $C$  by concatenating the output of the attention heads, which is then projected to the model dimension  $d_m$  with learnable projection matrix  $W_o$ .

**Transformer block.** We now define a pre-norm variant of the Transformer block using multi-head attention and MLP where the input to each sublayer is normalized and each sublayer is followed by a residual addition:

$$\begin{aligned} G &= X + \text{MHA}(\text{N}_X(X), \text{N}_M(M); \theta_{\text{attn}}), \\ Y &= G + \text{MLP}(\text{N}_G(G); \theta_{\text{ffn}}), \\ f_\theta(X, M) &= Y. \end{aligned} \tag{2.1}$$

Here,  $G$  is the output of the attention sublayer and its residual connection. The operators  $\text{N}_X$ ,  $\text{N}_M$ , and  $\text{N}_G$  denote normalization operations, such as LayerNorm [2] or RMSNorm [36], applied to  $X$ ,  $M$ , and  $G$ , respectively. The MLP is applied independently to each position and preserves the sequence length and model dimension.

A Transformer of depth  $L$  stacks  $L$  such blocks. Within each block, attention processes the sequence positions in parallel, whereas information passes through the  $L$  blocks sequentially. Below we examine this distinction and its relevance for reasoning and search.

### 2.2.2 Sequential Computation in fixed-depth Transformers

Different branches of a search may be expanded in parallel, but along an individual trajectory, each successor  $s_{i+1} = \mathcal{T}(s_i, a_i)$  can only be generated after its predecessor  $s_i$ . Under this search procedure, reaching depth  $k$  requires  $k$  dependent state transitions. Constructing this explicit trajectory is part of our search formulation; other approaches may solve the same task without representing such a trajectory. Since the required trajectory length can vary between problems, we are interested in architectures that can scale their sequential computation depending on the problem instance.

Let us consider a Transformer  $f_\theta$  with a fixed number  $L$  of layers where  $L$  does not depend on input length. The Transformer  $f_\theta$  can process input of different sequence lengths, and its total computation grows with the sequence size because more representations and attention interactions need to be performed. However, a single forward pass still applies only  $L$  Transformer layers in sequence. So, increasing sequence length increases the computation performed at those layers, but does not introduce additional sequential computation. Under various assumptions including logarithmic numerical precision, Merrill and Sabharwal [21] show that fixed-depth Transformers can be simulated by a polynomial-size, constant-depth family of threshold circuits. This places its computation in  $\text{TC}^0$  under those assumptions. This provides a formal account of the bounded computation depth of a single forward pass.

One approach for scaling the Transformer’s sequential computation is with autoregressive decoding. During autoregressive decoding, a Transformer generates one token, appends it to the current sequence, and executes the same fixed-depth decoder again. Each generated token therefore adds another dependent decoder call. LLM-based reasoning systems such as OpenAI o1[15] and DeepSeek-R1[6] use autoregressive decoding to generate intermediate reasoning tokens before producing the final answer. The resulting generation process thus can be viewed as a growing-state recurrence over discrete token representations.

A second approach is to use latent recurrent models where a shared Transformer component is repeatedly applied to the continuous internal representations. If the component is applied  $T$  times, its unrolled sequential depth grows with  $T$  while its total parameters remain fixed. Furthermore, when  $T$  can vary between problem instances, the model can adapt its computation to the problem. The following section formalizes this latent form of iterative computation.

### 2.2.3 Latent Recurrent Transformers

In this section, we formulate a unified view of *latent recurrent* computations with Transformers. We distinguish models based on whether the *state size* (the number of stored latent vectors) increases or remains constant over recurrent steps. In a recurrent model, the number of recurrent steps determines how long the model computes, whereas the state update determines how much intermediate information it retains. We therefore distinguish the models that update a fixed number of latent vectors from models that append new state vectors.

**Latent recurrent computation.** Let  $\mathcal{V} \subseteq \mathbb{R}^{d_m}$  denote the representation space, where  $d_m$  is the model dimension. We index the recurrent computation with  $\tau = 0, 1, 2, \dots$ . At step  $\tau$ , the model maintains a *latent state*  $S_\tau \in \mathcal{V}^{n_\tau}$  (equivalently,  $S_\tau \in \mathbb{R}^{n_\tau \times d_m}$ ), where  $n_\tau$  denotes the number of state vectors (latent tokens). Given an initial state  $S_0 = X$  (the input embedding), a

single recurrent step consists of:

$$Z_\tau = F_\phi(S_\tau), \quad (2.2)$$

$$S_{\tau+1} = u(S_\tau, Z_\tau), \quad (2.3)$$

where  $F_\phi : \mathcal{V}^{n_\tau} \rightarrow \mathcal{V}^{n_\tau}$  is a Transformer (a stack of Transformer blocks) and  $u$  is a state update operator. We refer to this computation as latent recurrence since each recurrence step receives the continuous internal representation from the previous step as input and processes it to generate the output in the internal representation itself without decoding it to any discrete tokens or symbolic states.

**Fixed-state latent recurrence.** A model is *fixed-state* if the state size is constant:

$$n_\tau \equiv N \quad \forall \tau$$

Equivalently,  $S_\tau \in \mathbb{R}^{N \times d_m}$  for all steps. A standard choice for  $u(\cdot)$  is an *overwrite* update, e.g.,

$$u(S_\tau, Z_\tau) = Z_\tau$$

which keeps the state shape fixed while allowing iterative refinement of the representation. Universal Transformers are one example of fixed-state latent recurrence [7].

**Expanding-state latent recurrence.** A model is *expanding-state* if the state size grows over the recurrent steps:  $n_{\tau+1} > n_\tau$ . A convenient choice for  $u$  is an *append* update:

$$u(S_\tau, Z_\tau) = [S_\tau; \rho(Z_\tau)],$$

where  $[\cdot; \cdot]$  denotes concatenation along the sequence length, and  $\rho(\cdot)$  is a function that extracts  $k$  new vectors from the current model activation  $Z_\tau$ . This preserves the past state content while adding new latent information.

**Example: Storing Intermediate Results.** Consider computing  $2 + 3 + 5 + 7$  by first computing

$$r_1 = 2 + 3 = 5, \quad r_2 = 5 + 7 = 12, \quad r_3 = r_1 + r_2 = 17.$$

A fixed-state recurrent model maintains a fixed-size state  $S_\tau \in \mathcal{V}^N$  at every recurrent step. It must therefore encode  $r_1$  and  $r_2$  within the same fixed number of state vectors before combining them to obtain  $r_3$ .

An expanding-state model can instead append a new latent vector for each intermediate result:

$$S_1 = [S_0; h_1], \quad S_2 = [S_1; h_2], \quad S_3 = [S_2; h_3],$$

where  $h_i$  represents the intermediate result  $r_i$ . Thus, the representations of  $r_1$  and  $r_2$  remain separately available when the model computes the final result. Although both approaches may solve this simple problem, they differ in how they store the intermediate computation: the fixed-state model must retain the relevant results within a fixed number of vectors, whereas the expanding-state model can preserve each result in a separate vector. The final computation might be simpler with an expanding state model since it retains the necessary intermediate results, but it also increases the memory and attention computation required at later steps.

The ST developed in this thesis is an expanding-state latent recurrent model. At each recurrent step, it performs a search operation to generate a new latent search node and appends that node to the existing state.

### 2.3 Multiple Instance Learning (MIL)

ST generates multiple search node representations, while it receives only a single binary label for the complete problem. We therefore formulate its readout using the MIL framework. In multiple-instance learning, a training example consists of a finite, unordered collection of instances, called a *bag*, and one bag-level label. Only a single label for the entire bag is observed, and the labels for individual instances are not available.

Let  $\mathcal{X}$  be the instance space and let  $\mathcal{O}$  denote the space of bags over  $\mathcal{X}$ . The training set is then given by

$$\mathcal{D} = \{(O_i, y_i)\}_{i=1}^N, \quad O_i = \{x_{i,1}, \dots, x_{i,m_i}\} \in \mathcal{O}, \quad y_i \in \mathcal{Y}.$$

The task is to learn a bag-level classifier  $F : \mathcal{O} \rightarrow \mathcal{Y}$ . Here, we focus on binary classification, where  $\mathcal{Y} = \{0, 1\}$ . The MIL setting alone does not specify how the instance-level information determines the bag label. A common assumption, which motivates our main readout, is the existential assumption. Under this assumption, each instance has an unobserved label  $z_{i,j} \in \{0, 1\}$ , and the bag label is

$$y_i = \max_{1 \leq j \leq m_i} z_{i,j}.$$

Thus, a bag is labeled positive if at least one instance has a positive label, and it is negative if all instances have negative labels. Other MIL settings allow the bag label to depend on multiple instances jointly. In the binary classification setting below,  $\ell(O) \in \mathbb{R}$  denotes a prediction logit for the bag from which the prediction  $F(O)$  is obtained by applying a sigmoid and a decision threshold.

Amores [1] presents three multiple-instance classification paradigms. Here, we focus on the Instance Space and Embedded Space paradigms because they correspond to the readout functions evaluated in this thesis. The two paradigms model discriminative information at different levels. The Instance Space paradigm assumes that the discriminative information is available locally within individual instances. The instances are therefore scored independently, then their scores are aggregated to obtain the bag-level prediction. In the Embedded Space paradigm, the individual instance representations are first aggregated into a single bag representation, and then the prediction is made using this representation. This allows the classifier to use bag-level information formed jointly from multiple instances.

### 2.3.1 Instance Space (IS) Paradigm

Given a bag of instances  $O = \{x_1, \dots, x_m\}$ , the goal of the Instance Space (IS) paradigm is to learn an instance-level scoring function  $s : \mathcal{X} \rightarrow \mathbb{R}$  and obtain the bag-level logit by pooling the instance scores:

$$\ell_{\text{IS}}(O) = \text{Pool}(s(x_1), \dots, s(x_m)).$$

Here, Pool is a permutation-invariant function, such as max pooling or LogMeanExp pooling.

**Pooling Instance Scores.** Let  $\ell_i = s(x_i)$  be the score predicted for instance  $x_i$  and let  $\ell = [\ell_1, \dots, \ell_m]$  be the scores for the bag.

*Max pooling.* The bag-level logit is the largest instance score:

$$\ell_{\text{max}}(O) = \max_{1 \leq i \leq m} \ell_i.$$

This directly mirrors the existential assumption: a single high instance score can make the bag-level logit high.

*LogMeanExp pooling.* It is a smooth alternative to the max pooling. It can be easier to optimize because it distributes gradient across multiple instances, whereas max pooling may propagate gradient only through the instance attaining highest score<sup>1</sup>.

$$\ell_{\text{LME}}(O) = \tau_{\text{pool}} \log \left( \frac{1}{m} \sum_{i=1}^m \exp \left( \frac{\ell_i}{\tau_{\text{pool}}} \right) \right), \quad \tau_{\text{pool}} > 0.$$

As  $\tau_{\text{pool}} \rightarrow 0^+$ , this operation converges to max pooling. Whereas, larger temperatures distribute the contribution across more instances.

<sup>1</sup>In case of tied score, it can propagate gradient through the tied instances depending on the implementation.

### 2.3.2 Embedded Space (ES) Paradigm

Let  $O = \{x_1, \dots, x_m\}$  be a bag of instance embeddings with  $x_i \in \mathbb{R}^d$ . We first aggregate the instance embeddings to obtain a single vectorial representation, which is then used for classification:

$$\Phi : \mathcal{O} \rightarrow \mathbb{R}^d, \quad g : \mathbb{R}^d \rightarrow \mathbb{R}, \quad \ell_{\text{ES}}(O) = g(\Phi(O)).$$

$\Phi$  is a permutation-invariant pooling operation, and  $g$  predicts a bag-level logit from the aggregated representation. Below, we present two pooling operations for instance embeddings.

We consider *mean* and *max* pooling operations:

$$\Phi_{\text{mean}}(O) = \frac{1}{m} \sum_{i=1}^m x_i, \quad [\Phi_{\text{max}}(O)]_k = \max_{1 \leq i \leq m} [x_i]_k, \quad k = 1, \dots, d.$$

Mean pooling averages the information across all instances, whereas max pooling performs the elementwise maximum operation and retains the largest value in each feature dimension.

The main readout of ST follows the IS paradigm: each search node is scored against the query before the resulting scores are pooled. The Embedded Space (ES) alternative first pools the search-node representations and then performs query matching using the aggregated representation. We evaluate this alternative in an ablation study.

## 2.4 Adaptive Computation

Recurrent models such as the one described in Section 2.2.3 perform a fixed amount of computation for a fixed input size and recurrent horizon (number of recurrent steps). Thus, two problems of similar size receive a similar amount of computation irrespective of their complexity. Although computation can scale with input size, input size does not necessarily correspond to the inherent problem complexity.

Consider two logical-deduction problems with the same fact  $A$  and the same rules  $A \rightarrow B$ ,  $B \rightarrow C$ , and  $C \rightarrow D$ , but with different queries  $B$  and  $D$ . Shortest derivation of  $B$  requires one rule application, whereas shortest derivation of  $D$  requires three. Thus, inputs of the same size can have different derivation depths. A model with a fixed recurrent horizon  $T$  nevertheless applies the same  $T$  recurrent updates to both inputs. This motivates a mechanism that allows the recurrent horizon to vary with the problem instance.

In the following section, we describe one approach that enables the model to perform variable amount of computation depending on the input. So, the model is not only trained to solve the main task, but also to determine when to stop the recurrent computation.

### 2.4.1 PonderNet

Here, we discuss PonderNet [3], a general framework for adaptive computation.

**Recurrent Computation.** We use the latent recurrent framework from Section 2.2.3. Let  $S_t$  be the latent state or representation after recurrence step  $t$ .

At each step  $t$ , PonderNet requires the recurrent model to produce a task prediction and a conditional halting probability from its current state:

$$\hat{y}_t = o_\omega(S_t), \quad \lambda_t = h_\psi(S_t) \in (0, 1).$$

$o_\omega$  is the readout function with learnable parameter  $\omega$  and  $h_\psi$  is the halting head with learnable parameters  $\psi$ .

**Halting Distribution.** Let  $H \in \mathbb{N}_{>0}$  be the random recurrent step at which the computation halts. Then, we define  $\lambda_t$  as the halting probability conditioned on the computation reaching step  $t$ :

$$\lambda_t = \Pr(H = t \mid H \geq t).$$

Now, the unconditional probability of halting at step  $t$  can be obtained as:

$$p_t = \Pr(H = t) = \lambda_t \prod_{i=1}^{t-1} (1 - \lambda_i).$$

This represents the computation continuing through the first  $t - 1$  steps and halting at step  $t$ . This induces a generalized geometric halting-time distribution  $p = (p_1, p_2, \dots)$ .

**Finite Computation Horizon.** When the halting distribution  $p$  is computed for a finite horizon  $T$ , there can still be probability mass remaining for later steps, resulting in  $\sum_{i=1}^T p_i < 1$ . Here, we present two approaches from PonderNet for obtaining a valid halting distribution over the finite horizon.

*Remaining Mass.* This method assigns the remaining probability mass to the last computation step:

$$\tilde{p}_t^{\text{rem}} = \begin{cases} p_t, & 1 \leq t < T, \\ 1 - \sum_{i=1}^{T-1} p_i, & t = T. \end{cases}$$

By doing so, this method forces the computation to halt at the last step if it has not halted before.

*Prefix Normalization.* The second method normalizes the probabilities over the finite horizon as:

$$\tilde{p}_t^{\text{norm}} = \frac{p_t}{\sum_{i=1}^T p_i}.$$

This normalization corresponds to conditioning the halting distribution on the event that the model halts by step  $T$ . We use  $\tilde{p}$  to denote the finite-horizon distribution obtained after performing the correction using one of the methods. The specific method used in ST is described in Chapter 4.

**Training Objective.** Using  $o_\omega(\cdot)$ , we produce the task prediction at each step. Using the finite-horizon halting distribution  $\tilde{p}$  from above, we compute the expected loss over the halting distribution. Let

$$p_G(t; \lambda_p) = \lambda_p (1 - \lambda_p)^{t-1}, \quad \lambda_p \in (0, 1),$$

be a geometric prior over the halting step, and let  $\tilde{p}_G(\lambda_p)$  denote the prior after applying the same finite-horizon correction. The PonderNet objective is

$$\mathcal{L}_{\text{PN}}(x, y) = \underbrace{\sum_{t=1}^T \tilde{p}_t \mathcal{L}(y, \hat{y}^{(t)})}_{\mathcal{L}_{\text{rec}}} + \underbrace{\beta \text{KL}(\tilde{p} \parallel \tilde{p}_G(\lambda_p))}_{\mathcal{L}_{\text{reg}}}, \quad \beta \geq 0.$$

The first term  $\mathcal{L}_{\text{rec}}$  (reconstruction loss) represents the expected prediction loss with respect to the finite-horizon halting distribution  $\tilde{p}$ . The second term  $\mathcal{L}_{\text{reg}}$  is a regularizer that biases the predicted halting distribution toward a geometric prior. Using only the reconstruction loss may cause the halting distribution to concentrate prematurely on a narrow range of steps, leaving other computation depths with little probability mass and hence little training signal. The regularizer above discourages this collapse and promotes exploration of different halting steps.

We refer to the ST variant equipped with this adaptive halting mechanism as ST-Ponder. Section 4.4 describes its model-specific configuration and training objective.

## 2.5 Memory Pointers

In standard classification problem, a model predicts over a vocabulary of fixed size. Ptr-Net [32] however predicts an index to the input sequence, so the number of possible outputs can vary with the input size.

Let  $\mathbf{E} = [e_1, \dots, e_K]^\top$  with  $e_i \in \mathbb{R}^{d_m}$  be an indexed sequence of memory elements. At each decoder step, the Ptr-Net scores each memory element using its current hidden state represen-

tation  $h$  and normalizes these scores:

$$\ell_i = s(h, e_i), \quad p_i = \frac{\exp(\ell_i)}{\sum_{j=1}^K \exp(\ell_j)}.$$

Here,  $s(\cdot, \cdot)$  is a scoring function and  $p_i$  is the probability of pointing to position  $i$ .

A hard pointer selects or samples an index, whereas a soft pointer retains the full distribution. Then, a soft/differentiable memory read is performed as:

$$a = \sum_{i=1}^K p_i e_i.$$

## 2.6 Curriculum Learning

Curriculum learning changes the training distribution over time by presenting examples in a specific order to facilitate optimization [4]. Defining a curriculum requires two choices: a difficulty measure for ordering examples and a pacing rule that determines when harder examples enter training.

**Difficulty Measure.** For each training example, we define a function  $\rho : \mathcal{X} \rightarrow \{1, \dots, R\}$  that assigns the example to one of  $R$  ordered curriculum rungs. These rungs can be constructed from one or more task-specific difficulty measures. At epoch  $k$ , let  $r_k$  denote the highest active curriculum rung. Then, the corresponding training set for epoch  $k$  contains all examples assigned to a rung that does not exceed  $r_k$ :

$$\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N, \quad \mathcal{D}_k = \{(x, y) \in \mathcal{D} : \rho(x) \leq r_k\}.$$

Here  $\mathcal{D}$  is the set of all available training examples and  $\mathcal{D}_k$  is the subset of available training examples at epoch  $k$ . Since the curriculum is cumulative, the examples from earlier rungs remain available when  $r_k$  increases.

**Performance-Based Curriculum Pacing.** Curriculum pacing determines how the active rung changes during training. It can follow a fixed schedule or depend on the observed model performance. We use a *performance-based* pacing rule that increases the active rung when an epoch-level performance measure, such as training or validation accuracy, reaches a predefined threshold. Let  $m_k$  be the performance score at epoch  $k$  and let  $\delta_c$  be the advancement threshold. Then, the active rung for the next epoch is updated as

$$r_{k+1} = \begin{cases} \min(r_k + 1, R), & m_k \geq \delta_c, \\ r_k, & \text{otherwise.} \end{cases}$$

The curriculum rungs used in this work are constructed using the task specific complexity parameters. Section 2.7 defines these parameters, while Section 4.6.1 describes how they are ordered into curriculum rungs during training.

## 2.7 Reasoning Tasks

In this section, we will look into two decision problems that are used to evaluate the performance of ST. Both tasks ask whether a sequence of valid operations can reach a specific goal. They differ in the information they must maintain during the search: logical-deduction problem requires maintaining only the reached nodes, whereas, the path exact-cover requires maintaining both the node and the set of items collected so far. Both datasets are generated programmatically using “depth” as an input parameter and are solver verified. In addition, the model receives a set of vector encoded objects as input rather than a natural-language description.

### 2.7.1 Logical Deduction

Let  $At$  be the finite set of atoms,  $F \subseteq At$  be the set of known facts,  $\mathcal{R}$  be the set of deduction rules, and  $q \in At$  be the query atom. Each rule has the form  $B \rightarrow h$  where  $B \subseteq At$  contains one or two atoms whose conjunction forms the rule body and  $h \in At$  is the rule head. Then the task is to decide whether the query  $q$  can be derived by repeated application of available rules.

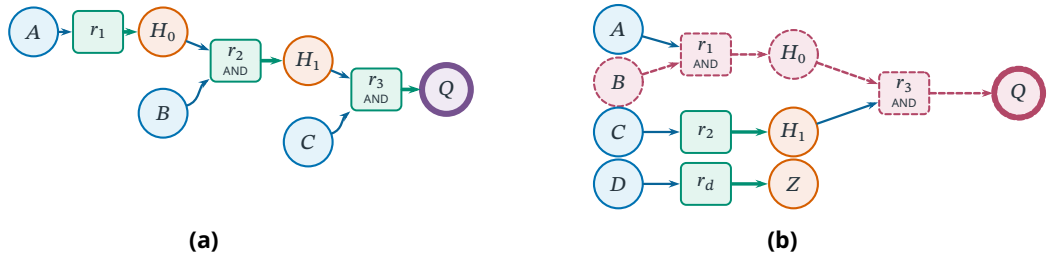


Figure 2.2: Forward-chaining logical deduction with query  $Q$ . **(a)** A positive instance with facts  $\{A, B, C\}$ , where applying  $r_1$ ,  $r_2$ , and  $r_3$  derives  $H_0$ ,  $H_1$ , and  $Q$ . **(b)** A negative instance with facts  $\{A, C, D\}$ . The missing fact  $B$  blocks  $r_1$ , so  $H_0$  and  $Q$  cannot be derived, while  $r_2$  and the distractor rule  $r_d$  remain applicable. Dashed elements show the blocked part of the proof.

**Problem Depth.** In logical-deduction, the problem *depth* refers to the shortest number of steps required to derive the query in a positive instance. A negative example is obtained by first generating a positive example of a specific “depth” and removing one of the required facts, making query derivation impossible. The negative example therefore keeps the depth of the original positive example and is used for the curriculum tier; so, this depth does not represent the number of steps required to prove non-derivability.

The positive instance in Figure 2.2 (a) has depth 3 since the shortest proof chain is of length 3. The negative instance in Figure 2.2 (b) was generated from a positive instance of depth 2 by removing the fact  $B$  required for deriving  $Q$ . It therefore retains the depth 2.

### 2.7.2 Edge-Labeled Path Exact Cover

Given a directed graph  $G = (V, E_G)$ , item set  $\mathcal{C}$ , an edge labeling  $l : E_G \rightarrow \mathcal{C}$ , and a target item set  $C \subseteq \mathcal{C}$ , the task is to decide whether there exists a directed edge sequence  $(e_1, \dots, e_m)$  from source node  $s$  to a target node  $t$  with multiset of its labels equal to the target set  $C$ , i.e.  $\{\{l(e_1), \dots, l(e_m)\}\} = \{\{c \mid c \in C\}\}$ .

Every item in  $C$  must occur once on the path, and no item outside  $C$  may occur. For simplicity, we refer to this problem by “Path Exact-Cover” or “Path-EC” throughout the document.



Figure 2.3: A representative example from the Disjoint Path-EC profile with target set  $C = \{a, b, c\}$ . In the positive instance (left), the highlighted path from  $s$  to  $t$  collects  $a$ ,  $b$ , and  $c$  exactly once. The paired counterfactual negative (right) is obtained by exchanging two edge labels. This preserves the graph topology and global item counts but removes every valid exact-cover path.

**Problem Depth.** We define *depth* as the size of item set:  $\text{depth} = |C|$ . We use this as the difficulty measure since a valid path contains exactly  $|C|$  edge traversals, with one item collected at each edge. In Figure 2.3, both examples have a target set of size 3, hence their depth is 3. The highlighted path in the left figure has length 3 and collects the required items  $a$ ,  $b$ , and  $c$ . In the right figure, none of the paths collects the required items exactly once, hence it has a negative label.

Both tasks can be formulated as a graph reachability in a task-specific state space. In logical-deduction, a state is a set of atoms derived so far and the transition applies an applicable rule. In *path exact-cover*, the state is the graph node and the collected items and transition traverses a valid edge. More details on data generation for logical deduction is available on Appendix A.1.1 and for path exact-cover on Appendix A.1.2.

## 3 Related Work

In this chapter, we review previous work most closely related to ST. We first discuss recurrent Transformer models that perform multiple computation steps in latent representations using shared components and enable adaptive computation. We then review learning-to-plan methods that use an internal transition model to generate relevant information for downstream tasks, providing architectural intuition for the successor-generation mechanism of ST. Finally, we relate ST to token-based and continuous reasoning methods based on LLMs.

### 3.1 Recurrent Computation with Transformers

Recurrent Transformer architectures reuse shared Transformer components to process the input over multiple computation steps. Depth-recurrent models are one variant in which recurrence is applied over computation depth by repeatedly refining a collection of latent representations. This is different from standard Recurrent Neural Networks (RNNs), where the recurrence is typically applied over the sequence dimension. At each step, the shared component receives latent representations from the previous step and produces updated representations for the next step. Applying the Transformer component for  $T$  steps therefore gives an unrolled recurrent depth of  $T$  without introducing separate parameters for each step. This decoupling of parameter count from computation depth allows these models to use additional recurrent steps at inference time and also allows adaptive computation. The Universal Transformer [7] and Looped Transformer architectures [11] are closely related models of this form.

Dehghani et al. [7] introduced UT as a depth-recurrent Transformer that iteratively refines a sequence of representations whose length remains fixed across recurrent steps. At each recurrent step, it updates all positions in parallel using self-attention followed by a transition function, with the parameters shared across the recurrent steps. The authors evaluated UT on various algorithmic, language-modeling, and language-understanding tasks. On algorithmic tasks, when evaluated on sequences longer than those used during training, UT substantially outperformed the standard Transformer and Long Short-Term Memory (LSTM) baselines, although it remained below the curriculum-trained Neural GPU [17]. The model also outperformed Transformer baselines on several language-related tasks, showing that the recurrent architecture was applicable beyond synthetically generated algorithmic tasks. The authors also introduced an Adaptive Computation Time (ACT)-based variant in which each position could halt at a different recurrent depth; we discuss its results in the following section.

Looped Transformer architectures place a Transformer (with one or more layers) in a loop that allows the parameter-shared Transformer component to update the latent state (sequence of representations) iteratively [11]. Each iteration receives the state produced by the previous iteration, with some variants additionally injecting the original input or iteration information. Yang et al. [34] showed that this structure can learn iterative algorithms for in-context data-fitting problems. On the tasks they considered, the looped models achieved performance comparable to standard Transformer baselines while using less than 10% as many parameters. This provides evidence that weight sharing can give a parameter-efficient inductive bias for learning iterative computation.

Beyond parameter efficiency, looped Transformer architectures have also been studied for length generalization and reasoning. Fan et al. [8] considered algorithmic tasks that can be expressed as repeated applications of an operation in RASP-L, a restricted programming language for describing computations realizable by a fixed-depth Transformer. On copying, parity, and addition, models whose number of loops increased with input length generalized substantially better to longer instances than fixed-depth Transformers. However, their method assumes that the required number of steps is available during training, and the step count is also used during inference. On selected synthetic reasoning tasks, including addition and multi-hop induction, Saunshi et al. [28] found that a shallow Transformer applied repeatedly could nearly match a non-looped Transformer with the same effective depth while using fewer parameters. At language-model scale, Geiping et al. [10] showed that a depth-recurrent model could improve on several reasoning benchmarks, particularly in mathematics and coding, when given more recurrent steps at inference. Together, these results show that recurrent depth can support reusable computation in particular settings.

Núñez-Molina et al. [22] proposed Search Transformer as an expanding-state latent recurrent Transformer. The core hypothesis was that reasoning processes can be formulated as search and that organizing the test-time computation around differentiable node selection and successor generation should provide a better inductive bias than less structured recurrent computation. At each step, the model predicts which latent node to expand and which action to apply. It then generates candidate successors and appends one successor to the existing nodes. The Output Transformer takes the generated nodes as additional latent context for predicting the output. The work contains the high-level proposal for the architecture without a concrete implementation or an empirical evaluation.

Our work is an extension of the proposal by Núñez-Molina et al. [22] where we keep the overall concept of node expansion and successor generation the same while making changes to various other architectural components described in Chapter 4 so that it learns to solve the problems.

### 3.2 Adaptive Computation and Learned Halting

Parameter sharing allows depth-recurrent models to use different numbers of updates with a fixed set of model parameters. However, the number of updates still needs to be specified in advance. A fixed number may be insufficient for a complex problem, while using the same budget for a simpler problem can waste computation. Adaptive computation methods instead learn when to halt for each problem instance. This makes the learning problem more challenging because the model must learn both to solve the task and to allocate its computation.

Graves [12] introduced Adaptive Computation Time for RNNs, allowing a model to perform a variable number of internal updates between receiving an input and producing an output. At each pondering step, the model predicts a halting probability. These probabilities are used to combine the intermediate states and outputs, while a ponder cost penalizes excessive computation. On four synthetic tasks, ACT improved performance and learned to allocate more computation according to task requirements. The performance gains were small on the character-level language-modeling task, but the model consistently used more pondering steps at harder-to-predict transitions, such as spaces between words and ends of sentences. This suggests that the learned pondering reflected the structure and demands of the data.

Dehghani et al. [7] adapted ACT to UT so that each sequence position can halt at a different recurrent step. Positions that halt early keep their current representation while the remaining positions continue to be updated. Dynamic halting improved performance on several structured tasks but slightly degraded performance on machine translation. On the bAbI tasks, the average ponder time increased with the number of supporting facts required to answer a question, suggesting that the model allocated more computation when more supporting facts had to be considered.

Banino et al. [3] introduced PonderNet as an alternative to ACT. At each recurrent step, the model predicts an output and the probability of halting conditioned on not having halted before. These probabilities define a distribution over the step at which the model stops. Training minimizes the expected task loss under this distribution and regularizes the halting distribution toward a geometric prior. Unlike the ponder cost used by ACT, this regularizer encourages the model to explore different computation depths while biasing it toward the prior’s expected depth.

On the parity task, PonderNet achieved higher accuracy than ACT while using fewer computation steps. When trained on examples containing 1–48 nonzero elements and evaluated on examples containing 49–96, PonderNet increased its average computation from around three to five steps and maintained high accuracy, while ACT remained at chance level. This provides evidence that PonderNet can increase its computation during extrapolation. On the bAbI tasks, a Transformer using PonderNet solved all 20 tasks with lower average error and substan-

tially fewer reported pondering steps than the UT baseline using ACT. These results support PonderNet as a viable alternative to ACT for adaptive computation in recurrent architectures.

ST generates one successor node at each search step. If the new node matches the query, the search can stop, and the model can predict a positive label. We equip ST with a PonderNet-style halting mechanism and refer to this model as ST-Ponder. We study ST-Ponder with a focus on whether its learned halting behavior varies with problem depth and whether adaptive halting improves performance over fixed-step ST.

### 3.3 Learning to Plan

We compare ST with learning-to-plan methods [23] that apply an internal model repeatedly before producing an action or a value estimate. These methods were developed for sequential decision problems, whereas this thesis considers static structured problems that require a binary decision. They also differ from ST in their training objectives and in how they use their internal models. The comparison is therefore architectural: these methods demonstrate how repeated application of internal transitions can generate useful information for a downstream task and motivate different components of ST.

Imagination-Augmented Agents (I2A) [27] and the Imagination-Based Planner (IBP) [26] use model rollouts to provide additional context to the policy network. In the evaluated I2A architecture, an environment model generates a set of fixed-depth rollouts. A rollout encoder then converts each trajectory into a summary of imagined outcomes, and the resulting summaries provide additional context to the policy. The IBP learns whether to act in the real world or perform an imagined transition using a learned model, which action to use for that transition, and which previous imagined state to continue from. It accumulates the imagined actions and outcomes into a plan context that influences subsequent imagined and real actions. The reported experiments show improvements over model-free or fixed-imagination baselines on several evaluated tasks, providing evidence that summaries of model-generated outcomes can be useful for a downstream decision. ST follows a related high-level idea: its search-structured internal computation repeatedly generates latent nodes and makes the accumulated information available to the final readout function. Since the earlier nodes remain available for selection, the architecture permits a parent to be revisited and a different successor to be generated rather than restricting the computation to a single rollout.

Both I2A and the continuous-control IBP use learned dynamics models to predict the next observation or state and, for I2A, the associated reward. I2A pretrains its model separately, whereas the continuous-control IBP learns its model online from the agent’s experience using supervised next-state targets; the maze study instead uses perfect dynamics. Results from other planning methods suggest that predicting the complete next observation is not always necessary. Value Prediction Networks (VPNs) [25] combine an option-conditioned abstract

transition with outcome and value modules to predict rewards, discounts, and values rather than future observations. TreeQN [9] recursively applies learned action-conditioned transitions to construct a complete fixed-depth tree and then uses a predefined soft value backup to obtain the root-action values. MuZero [29] also learns planning-relevant latent dynamics and uses them for planning with Monte-Carlo Tree Search (MCTS). Although these approaches still use environment-defined actions within their internal rollouts, their transitions operate entirely in a learned latent space and are trained to produce information useful for the planner rather than to reconstruct the observation. This is closer to ST, whose transition function takes a latent node and generates a latent successor node conditioned on an action representation retrieved from problem memory. In the reported experiments, the search nodes have no direct semantic targets and are learned only through their effect on the final classification loss.

The value equivalence principle [13] provides a theoretical perspective on why task-directed abstract models can be useful for value-based planning. Two models are value equivalent with respect to a set of policies and value functions when they produce the same Bellman updates. Consequently, a planner may not require a model that reproduces the complete environment dynamics if a simpler model preserves the quantities required for those updates. Grimm et al. [13] argue that this perspective underlies methods including VPNs, TreeQN, and MuZero. The principle does not directly apply to ST, since the training setting considered here has neither Bellman updates nor explicit value semantics. We use it only as an intuition for the transition design: a latent successor node may be sufficient without having to reconstruct a known problem state if it preserves the information needed for the final prediction. Thus, the success of learned abstract models in planning provides motivation for this design choice, even though these planning methods operate in a different setting.

### 3.4 Reasoning with LLMs

Generating intermediate reasoning tokens can improve the performance of sufficiently capable models on some reasoning problems, a process commonly known as CoT reasoning [33]. Because ST and LLMs differ significantly in their training objectives, input representations, and scale, we keep the comparison mainly architectural. In both cases, intermediate representations are appended to a growing scratchpad and remain available when producing the final output. We distinguish between two LLM-based reasoning paradigms: discrete CoT, which generates intermediate tokens in language space, and continuous CoT, which feeds back real-valued hidden representations [14].

LLMs can be used to generate explicit CoT tokens for reasoning through either prompting [18] or training [6, 24]. In addition to providing more computation, generating CoT allows a model to solve part of the problem or gather relevant information before outputting the solution. ST and these methods share this notion of deriving useful information recurrently to assist the

downstream task, although their mechanisms differ. Standard autoregressive CoT produces one linear sequence within each decoding trajectory and does not by itself maintain earlier states to which the process can backtrack or from which it can try alternative reasoning paths. Although ST is not guaranteed to perform search and exploration in latent space, its architecture permits revisiting earlier nodes and generating alternative successors.

Tree of Thoughts (ToT) [35] builds on a pretrained LLM by organizing CoT generation as a search process. It is an inference-time method in which an off-the-shelf LLM is used to construct and evaluate states and perform search over them, allowing the procedure to evaluate multiple candidates and backtrack when necessary. This is similar to how ST can use its selection process to return to previous nodes and explore an alternative continuation. The main difference is that the selection and transition functions in the select–expand–append process of ST are learned from the final prediction loss, whereas ToT uses an externally specified search procedure over textual thoughts.

The search nodes generated by ST are latent vectors, so continuous CoT methods such as COCONUT [14] are closer to ST in the sense that they produce continuous vectors and feed them back into the model without decoding them into tokens. Although continuous thoughts are real-valued vectors, they remain part of the representation space of a pretrained LLM. Discrete CoT can be verbose or redundant [5, 30]. On the evaluated tasks, the authors report that COCONUT uses fewer reasoning tokens than discrete CoT [14]. Zhu et al. [37] present theoretical and empirical results suggesting that COCONUT-style models can generate continuous thoughts that represent a superposition of multiple possibilities. Intuitively, they argue that when a model has two promising paths, discrete CoT must choose one within a single trajectory, whereas continuous CoT can encode both possibilities in the next continuous state.

In terms of scratchpad representation, continuous CoT provides the closest comparison to ST, since both append continuous states without decoding them. In terms of search-like behavior, ToT provides a closer parallel. Besides ToT’s use of a pretrained LLM, the main difference is that ST learns the selection and expansion components, whereas ToT uses a predefined search algorithm over textual thoughts. Thus, ST combines the concept of a continuous scratchpad with an internal search-oriented controller and trains its learnable components using the final decision loss. In addition, ST as implemented in this thesis requires its input to be encoded as a set of vectors and does not operate directly on natural-language inputs.

### 3.5 Summary

Recurrent Transformer models provide one way to perform intermediate computation in latent representations rather than generating discrete CoT tokens. Learning-to-plan methods offer a complementary viewpoint, suggesting that the repeated application of a learned abstract transition model can generate useful “imagined” outcomes for downstream tasks and that

the transition can operate entirely in an abstract representation space without needing to reconstruct the next observation. In particular, IBP learns to select a previous imagined state to continue from, allowing its internal computation to revisit earlier states and create branches. While these methods differ from ST in problem formulation and scope, they still offer intuition for several of ST’s architectural ideas. This thesis therefore asks whether structuring recurrent computation around learned node selection and an abstract transition function offers a beneficial inductive bias for reasoning problems in contrast to the joint refinement of a fixed collection of representations in UT.

## 4 Methods

This chapter presents the ST architecture used in this work. The overall search formulation is adopted from [22]. This thesis extends the original architecture by using memory pointers for actions, a pooling operation with goal matching for output prediction, and a PonderNet-based halting mechanism. Furthermore, we assume the reasoning tasks we solve in this work are decision problems with a structured input so that the model receives an encoded set of vectors, rather than having to learn the parsing itself.

### 4.1 Problem Formulation

The problems we solve in this work need some form of a non-trivial “reasoning” process. Here, we define “reasoning” as a process of manipulating existing knowledge available in some form of representation to reach a conclusion. Furthermore, the “reasoning” performed by ST follows prescribed steps analogous to those used in search. Hence, the “reasoning” problems we solve are those that can be represented as a search procedure where we start from an initial state and, after some computation, need to decide whether the goal is reachable within the available computation budget.

Although learning the representations from raw sensory data has been an integral part of Deep Learning (DL) systems, our focus here is mainly on the ability of a recurrent DL-based system to manipulate a given set of representation elements to make a decision. So, ST needs to discover the implicit relationships in a given problem description and utilize recursive computation to derive a solution. For the problems we solve, a problem context and a query are given to the model. Then, the model needs to decide whether the query is provable or reachable using the information from the problem context. So, we model the task as a binary classification problem where class labels 1 and 0 represent “yes” and “no” decisions, respectively.

#### 4.1.1 Problem Representation

**Example: logical deduction**

Facts:  $A, C$

Rules:  $A \rightarrow B, B \wedge C \rightarrow D$

Query:  $D?$

Label: 1

Let  $\mathcal{E} \subseteq \mathbb{R}^d$  be the set of all possible elements of dimension  $d$ . Here, an “element” represents a specific object, relationship, concept, or a combination of them. Using the elements, we define a problem as a tuple  $(E, q)$ , where  $E \subseteq \mathcal{E}$  is the problem context given by the set of elements and  $q \in \mathcal{E}$  is the query given by a single element.

The above example of logical deduction provides a textual description of the problem. To solve this problem with ST, we need to translate the problem from textual form to a tuple of elements  $(E, q)$ . In this work, we assume that there exists an encoding function (or an encoder) that can be utilized to convert a problem in this form into a tuple of elements. For the deduction problem, the encoding function can be of the form  $e : \text{At} \cup \mathcal{R} \rightarrow \mathcal{E}$ , which takes an item from the set of atoms  $\text{At}$  used to define facts and queries, or an item from the set of rules  $\mathcal{R}$ , and returns an element vector. The encoder is a domain-dependent function, and we refer to Appendix A.1.1 and A.1.2 for more detailed descriptions of the encoders used in this project.

We make a distinction between problem context  $E$  and query  $q$  because a problem can change significantly when a different query is asked in the same context. The problem context in the above example is the set of known facts and rules. When it is combined with a specific query such as “Is the atom D provable?”, the tuple of the two represents a problem instance that we want the model to solve. If a different query such as “Is the atom B provable?” is asked in the same problem context, the solution path changes drastically.

We represent a training example by a tuple  $(x, y)$  with  $x = (E, q)$ , where  $E$  is a set of elements representing the problem context,  $q$  is an element representing the query, and  $y \in \{0, 1\}$  is the decision label. For the above example, let  $E_0 = \{e_A, e_C, e_{A \rightarrow B}, e_{B \wedge C \rightarrow D}\}$  be the set of encoded elements for the given facts and rules,  $q_0 = e_D$  be the encoded query, and  $y_0 = 1$  be the decision label. The model then receives  $x_0 = (E_0, q_0)$  as input and  $y_0$  as the target label.

Although we use a predefined, problem-dependent function as an encoder, learning the encoder in an end-to-end manner is possible, and we consider this future work.

**Problem Solving as Search.** Here, we model the problem-solving process as a search procedure that can be used by ST to solve decision problems. At any time step, we assume that the model has access to a set of frontier nodes. Then, the model needs to select a node from that frontier and an action/operator to execute at the selected node. Using the selected node–action pair, the model performs an expansion operation to obtain a new node. We add this successor node to the frontier, and the search process repeats.

The search process starts with a single “start” node in the frontier, where the start node is a problem-specific element and its nature is explained later. In addition, the search is a goal-directed process, where the goal is defined as an element representing the query  $q$ . That is, we perform search in order to reach a node that is sufficiently close to the goal. In the example above, the goal is a vector representing the query atom “D”. During search, the generated

search nodes can contain information about atoms derived so far or newly derived atoms. To test whether a search node proves the goal, we assume there exists a goal-matching function  $m : \mathcal{N} \times \mathcal{E} \rightarrow \mathbb{R}$  that takes a search node and the query element as input and returns a real number indicating the degree to which the search node contains the query element. The search process terminates after reaching a search node that is sufficiently close to the goal or running out of search budget.

If the search process finds a node that matches the goal with sufficient confidence, the model outputs a “yes” decision; otherwise, it outputs a “no” decision. This aligns with the IS paradigm we describe in Section 2.3.1. So, the question the model answers is: “Can we reach the goal in this problem instance?” In logical deduction, the question becomes: “Can we prove the query atom given the facts and rules?” In Path-EC, it becomes: “Can we reach the target node from the start node while collecting the exact set of target objects?”

Below, we describe a more precise architecture to achieve this using neural network architectures that can be trained in an end-to-end manner.

## 4.2 Overview of Notations Used

As defined above,  $\mathcal{E} \subseteq \mathbb{R}^d$  represents the set of problem elements and  $\mathcal{N} \subseteq \mathbb{R}^d$  represents the set of search nodes. We use  $[K]$  to represent the integers from 1 to  $K$ , i.e.,  $[K] = \{1, \dots, K\}$ . We use  $t$  to represent the time step of the search process and  $T$  to represent the total number of search steps, so  $t \in [T]$ . We represent search nodes by  $n_i$  and the list of search nodes available at time step  $t$  with  $N_t = n_{0:t-1} = [n_0, \dots, n_{t-1}] \in \mathcal{N}^t$ . Problem-context elements are represented by  $e_i$ , and the problem context is a set of  $K$  elements,  $E = \{e_1, \dots, e_K\} \subseteq \mathcal{E}$ , with  $|E| = K$ . The query is a single element represented by  $q \in \mathcal{E}$ . The symbol  $d$  represents the dimension of the elements, the model dimension of the Transformer components, and the dimension of the search nodes; i.e.,  $e_i, n_t, q \in \mathbb{R}^d$ . We use  $\mathbf{0}_d \in \mathbb{R}^d$  and  $\mathbf{1}_d \in \mathbb{R}^d$  to denote the all-zero and all-ones vectors, respectively, and omit the dimension when it is clear from context.

## 4.3 Search Transformer Architecture

In this section, we describe the main components and ideas surrounding the Search Transformer architecture. The goal of this section is to provide a mathematical model of ST that is implementation-agnostic and can be implemented with different methods.

### 4.3.1 Memory Layout

We assume ST has access to a memory tape. A memory tape is a unidirectional sequence of memory slots, where each slot can hold an object, which in our case is a  $d$ -dimensional vector

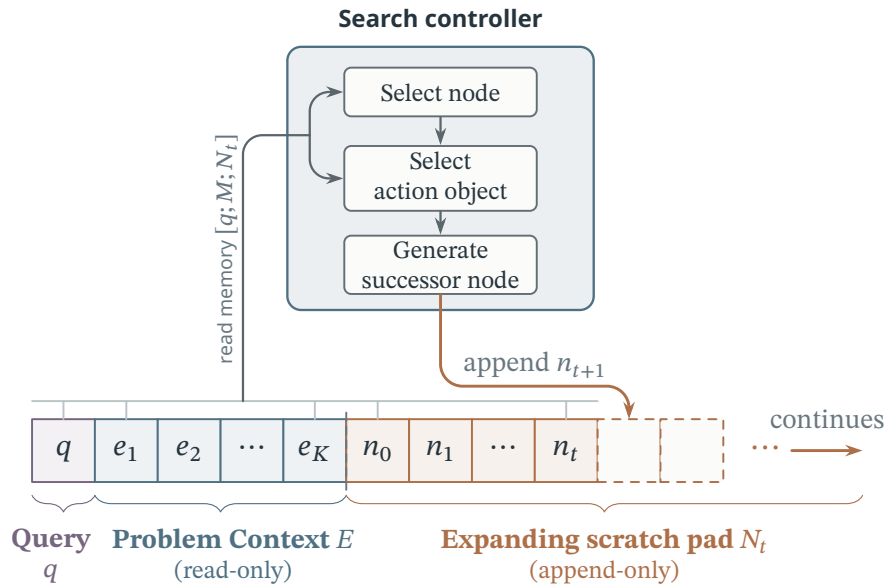


Figure 4.1: One search step works by reading information available on the tape, performing some computation, and writing a new item to the first available space on the right. The query  $q$ , problem context  $E$ , and the starting node  $n_0$  are already populated on the tape. A search step  $t$  appends a node  $n_t$ . The search process is iterative and is carried out until (1) the search budget runs out, (2) a solution (either positive or negative) is found, or (3) a halting condition is met, where each of these conditions depends on the implementation.

of fixed-precision floating-point numbers. In addition, we assume the memory tape is infinite in one direction, which we arbitrarily choose to be the right side of the tape.

**Fixed Input Memory.** The prefix of the memory tape is already populated with problem elements comprising the problem context and the query, as shown in Figure 4.1. Without loss of generality, we assume that the query is stored in the first memory slot and subsequent memory slots store the problem context. The elements of the problem context have no inherent ordering, so they can be stored in an arbitrary but fixed order in memory. We call this section of memory the memory prefix, or problem description, which ST uses to guide the search process.

**Expanding Scratchpad.** ST uses a “read” operation to read memory elements from different locations on the tape and uses an “append” operation to add new information to the first blank space on the tape (when scanning from left to right). Specifically, ST uses these blank memory slots to store the generated search nodes without having to modify existing memory slots. In addition, we assume that ST has access to an infinite amount of memory on the tape to use as a scratchpad. This assumption is generally not an issue when solving simpler problems, as we do in this work, but might be problematic when solving problems that require a search over a large state space.

### 4.3.2 Search Loop

At a high level, a single search step performs the following operations:

1. Predict selection weights over the existing search nodes.
2. For each node, apply the action-usage mask to its base memory pointer and retrieve an action from the problem-context encodings (memory).
3. Generate candidate successors for all node–action pairs in parallel.
4. Aggregate the candidate successors using the selection weights and append the resulting node.
5. Update the action-usage values based on the contribution of each node–action pair.

The search process begins with a single “start node”, which is the only node available for expansion in the first step. Each search step appends one new node while keeping all previous nodes available for later expansion. Since both node and action selection are soft, multiple node–action pairs can contribute to the new node. Thus, ST performs a differentiable relaxation of search rather than a discrete search process. The following subsections describe each component in detail.

### 4.3.3 Start Node Initialization

To start the search, we need a start node. For both logical deduction and Path-EC, we initialize the start node as a zero vector. Below, we describe two possible approaches one can use to initialize the start node.

**Problem-Dependent Initialization.** In problems such as planning or pathfinding, it is generally natural to initialize the start node as the agent’s start state. This can provide a useful inductive bias through which the model can learn more interpretable node representations. We present this as one possibility, though we do not use this approach here.

**Problem-Independent Initialization.** In some situations, we can initialize the start node with a zero vector, a fixed vector, or a learnable embedding. We use zero initialization in this project. Zero initialization indicates that the start node contains no information, and the transition function can utilize this to derive nodes containing useful information, such as newly proven facts. In addition, this allows the selection module to revisit this blank node and construct useful representations by trying different actions, allowing multiple problem-solving trajectories.

### 4.3.4 Node Selection

At search step  $t$ , the model has access to the input elements  $E = \{e_1, \dots, e_K\}$ , the query  $q$ , and the frontier nodes  $n_{0:t-1} = [n_0, \dots, n_{t-1}]$ . The selection function  $f_{\text{sel}}$  estimates a categorical

distribution over the frontier to determine which search node to expand next. Since the node-selection module has access to the query, it can learn a goal-directed expansion.

We implement  $f_{\text{sel}}$  using a Transformer block containing self-attention and cross-attention, as shown in Figure 4.2.

At time step  $t$ , the selection logits are computed as follows:

$$\ell_t^{\text{sel}} = f_{\text{sel}}(n_{0:t-1}, E, q) \in \mathbb{R}^t.$$

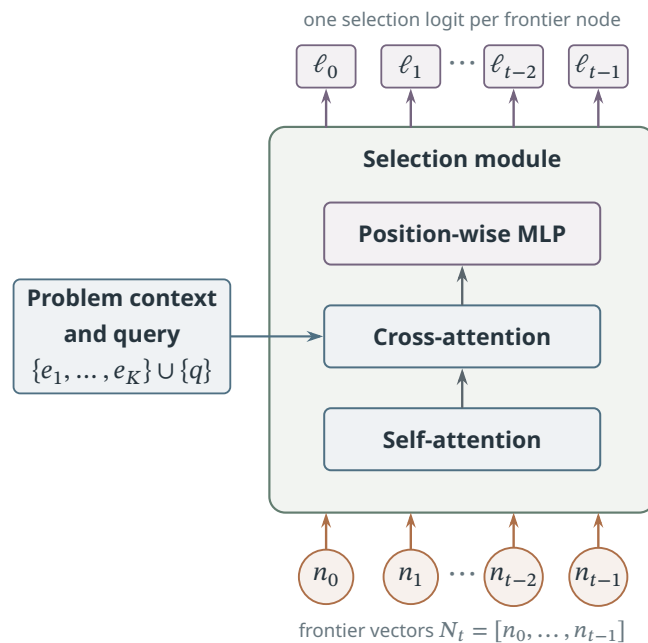


Figure 4.2: Selection over the frontier nodes. Self-attention contextualizes the frontier nodes, cross-attention reads the problem memory, and a position-wise MLP produces one selection logit for each node.

#### 4.3.5 Action Selection as Memory Pointer

A search node can represent a compressed set of relevant information, and an action represents an operation that can be performed at that node. In standard search, the actions available at a particular state are generally known. This has a natural meaning in planning, where the available actions define the operations that an agent can perform, such as moving to the left or right.

In this work, we represent the action as a memory pointer. The problem context  $E$  contains elements that can serve as possible operations. For example, an action in a deduction task can refer to a fact to retrieve or a rule to apply, while in the Path-EC problem it can refer to a

specific edge to traverse. Hence, elements of the problem context can function as actions in both tasks considered here.

For each node, the pointer defines a distribution over the possible actions (or memory elements). When revisiting a node, we want it to use a different action so that it can expand another branch. In a discrete setting, this could be implemented by sampling without replacement from that node’s action distribution. Since we use soft action selection, we approximate this behavior using the action-usage mask described in Section 4.3.7.

Following the pointer mechanism described in Section 2.5, we predict a distribution over the indexed memory elements. Inspired by Transformer attention, we compute its logits using multiple scaled dot-product pointer-scoring heads. The head logits are averaged before applying Softmax, so the heads jointly produce a single pointer distribution.

**Pointer Distribution.** As defined earlier,  $E$  represents a problem context containing  $K$  elements and does not contain the query element. Since  $E$  is a set and has no inherent ordering, we assign its elements a fixed order so that they can be addressed by a pointer. We use  $\mathbf{E} = [e_1, \dots, e_K]^\top \in \mathbb{R}^{K \times d}$  (bold  $E$ ) to represent the resulting list of elements.

For a node  $n_i$ , we concatenate the node with the problem query  $q$  to obtain the pointer-query representation  $z_i = [n_i; q] \in \mathbb{R}^{2d}$ , where  $[\cdot]$  denotes vector concatenation. The node represents the current search state, while the problem query conditions which memory element is useful for reaching the goal.

Following Transformer terminology, we project  $z_i$  into the query space and  $\mathbf{E}$  into the key space. We then use scaled dot-product scoring to compute one pointer logit for each memory element.

The action module uses  $H$  pointer-scoring heads. For each head  $h \in \{1, \dots, H\}$ , it computes:

$$r_i^{(h)} = W_Q^{(h)} z_i, \quad \text{project } z_i \text{ into the head's query space,} \quad (4.1)$$

$$K_E^{(h)} = W_K^{(h)} \mathbf{E}^\top, \quad \text{project the memory elements into the head's key space,} \quad (4.2)$$

$$\ell_i^{(h)} = \frac{\left(K_E^{(h)}\right)^\top r_i^{(h)}}{\sqrt{d_k}}, \quad \text{compute one pointer score per memory element.} \quad (4.3)$$

Here,  $W_Q^{(h)} \in \mathbb{R}^{d_k \times 2d}$ ,  $W_K^{(h)} \in \mathbb{R}^{d_k \times d}$ ,  $r_i^{(h)} \in \mathbb{R}^{d_k}$ ,  $K_E^{(h)} \in \mathbb{R}^{d_k \times K}$ , and  $\ell_i^{(h)} \in \mathbb{R}^K$ .

The pointer logits from the heads are averaged to obtain the base action logits:

$$\ell_i^{\text{base}} = \frac{1}{H} \sum_{h=1}^H \ell_i^{(h)}. \quad (4.4)$$

Using the base action logits, we define the unmasked soft action-pointer distribution for node  $n_i$  using Softmax:

$$\pi_{i,j}^{\text{base}} = \frac{\exp(\ell_{i,j}^{\text{base}}/\tau_{\text{act}})}{\sum_{k=1}^K \exp(\ell_{i,k}^{\text{base}}/\tau_{\text{act}})}. \quad (4.5)$$

Here,  $\tau_{\text{act}} > 0$  controls the sharpness of the pointer distribution. During training, we anneal it from its initial value of 1.0 to a lower value in the range 0.3–0.4, making the distribution sharper. The resulting  $\pi_i^{\text{base}}$  defines the base action distribution over the memory elements. We compute the base action logits once when the node is added and cache them for the remainder of the search. Then, we use a manual node–action usage masking approach (Section 4.3.7) to generate diverse actions when expanding the node again.

**Memory Read.** Let  $\pi_i^{(t)}$  be the pointer distribution after applying the action-usage mask (Section 4.3.7) for node  $n_i$  at search step  $t$ . We obtain the corresponding action representation using a weighted memory read:

$$a_i^{(t)} = \sum_{k=1}^K \pi_{i,k}^{(t)} e_k \in \mathbb{R}^d. \quad (4.6)$$

Thus,  $\pi_i^{(t)}$  is the soft pointer over memory indices, while  $a_i^{(t)}$  is a soft memory read that serves as an action representation.

#### 4.3.6 Candidate Expansion and Aggregation

A transition function  $f_{\text{tran}} : \mathcal{N} \times \mathbb{R}^d \rightarrow \mathcal{N}$  takes a node–action pair  $(n, a)$  as input and returns a successor node. The transition function is deterministic, so its output is fixed for a fixed node–action pair. We represent  $f_{\text{tran}}$  with a neural network, specifically an MLP with a gated update, as shown in Figure 4.3a.

Given a search node  $n_i$  and an action representation  $a_i^{(t)}$  retrieved using Equation 4.6, we compute the delta update as follows:

$$\delta_i^{(t)} = \text{MLP}_{\delta}([n_i; a_i^{(t)}]).$$

Then, we add the delta update, controlled elementwise by a sigmoid gate, to the node to obtain the successor:

$$\hat{n}_i^{(t)} = n_i + \sigma(\text{MLP}_g([n_i; a_i^{(t)}; \delta_i^{(t)}])) \odot \delta_i^{(t)}.$$

Here,  $[;]$  denotes vector concatenation,  $\odot$  denotes elementwise multiplication, and  $\sigma$  is the logistic sigmoid function.

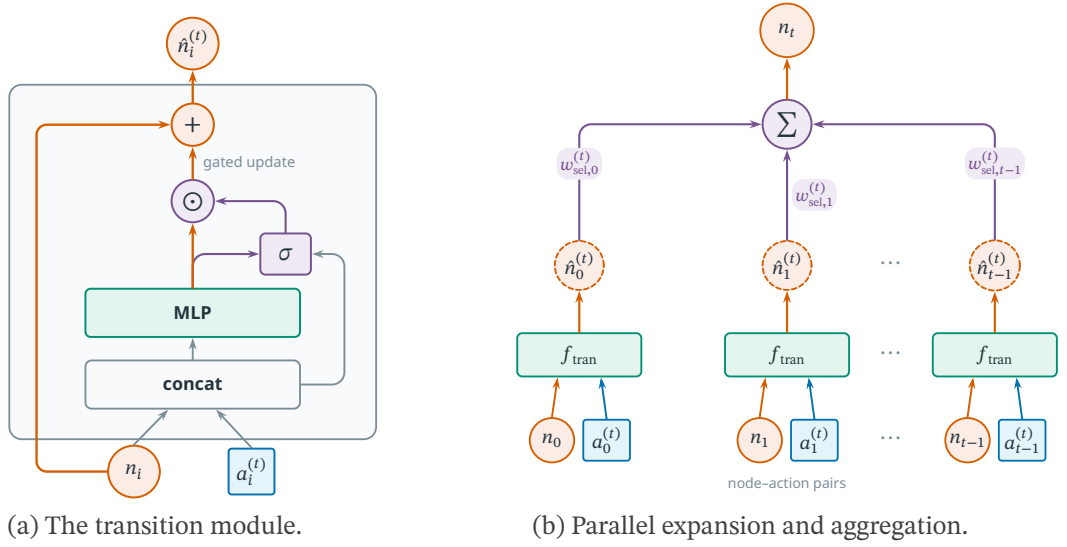


Figure 4.3: Candidate expansion and aggregation at search step  $t$ . Left: the transition module maps one node-action pair  $(n_i, a_i^{(t)})$  to a candidate successor  $\hat{n}_i^{(t)}$ . Right: the shared transition module shown on the left, abbreviated as  $f_{\text{tran}}$ , expands all frontier nodes in parallel. The selection weights  $w_{\text{sel},i}^{(t)}$  aggregate the candidate successors into the single successor node  $n_t$ , which is appended to the frontier.

**Parallel Expansion.** Let  $N_t = [n_0, \dots, n_{t-1}]$  be the list of nodes in the frontier. We perform the node expansion for all frontier nodes in parallel. Let  $[a_0^{(t)}, \dots, a_{t-1}^{(t)}]$  be the action representations obtained using memory pointers for the respective frontier nodes, and let  $[\hat{n}_0^{(t)}, \dots, \hat{n}_{t-1}^{(t)}]$  be the expanded successors obtained with  $\hat{n}_i^{(t)} = f_{\text{tran}}(n_i, a_i^{(t)})$ . Then, we aggregate all the candidate successor nodes to obtain a single successor using Gumbel-Softmax [16, 20] as a differentiable sampling method.

Let  $\ell_i^{(t)} := (\ell_t^{\text{sel}})_i$  be the selection logit for node  $i$  predicted at step  $t$  (obtained with the selection module in Section 4.3.4). Then, we estimate the weight for each node in the frontier as follows:

$$w_{\text{sel},i}^{(t)} = \frac{\exp((\ell_i^{(t)} + g_i)/\tau)}{\sum_{j=0}^{t-1} \exp((\ell_j^{(t)} + g_j)/\tau)}, \quad i \in \{0, \dots, t-1\}. \quad (4.7)$$

$\tau$  is the temperature parameter that controls the sharpness of the distribution that we keep constant at 1.0, and  $g_i$  is a sample from the standard Gumbel distribution. Lowering  $\tau$  makes the continuous relaxation closer to a discrete sample, while the Gumbel variables provide the sampling noise.

During training, we add the Gumbel noise  $g_i$  before applying Softmax, introducing stochasticity while remaining differentiable. During inference, we remove the Gumbel noise (setting  $g_i = 0$ ) and compute the selection weights directly using Softmax to avoid stochasticity. We keep the selection soft at inference to avoid the mismatch between training and inference. Alternatively,

training ST with temperature annealing or a straight-through estimator to enable discrete node selection remains an interesting direction for future work.

**Aggregation.** Using the selection weights, we obtain a successor node by aggregating the candidate successors:

$$n_t = \sum_{i=0}^{t-1} w_{\text{sel},i}^{(t)} \hat{n}_i^{(t)}.$$

Finally, we append the aggregated successor node  $n_t$  to the list of frontier nodes (or, equivalently, to the first empty memory slot) and iterate the same search loop until a stopping condition is reached.

#### 4.3.7 Update Action Usage

For each node, we compute the base memory-pointer logits when the node is added to the frontier and cache them for the remainder of the search. If we reused the resulting distribution without changing it, the same node would retrieve the same action representation and generate the same successor. Since the node representation is fixed, we need to change the action representation to generate different successors from the same node. We do this by applying a soft mask to memory elements that have already contributed, which shifts the action distribution when the model revisits a node.

If the node and action selection were discrete, then the selected node–action pair could simply be marked as unavailable. However, under the soft selection of nodes and actions, several nodes and actions contribute simultaneously to generate a successor node. We therefore maintain a soft availability value for every node–action pair. We then use that availability value to update the action selection logits at each time step, implementing a soft mask as follows:

$$\tilde{\ell}_{i,k}^{(t)} = \ell_{i,k}^{\text{base}} + \tau_{\text{act}} \alpha \log(r_{i,k}^{(t-1)}),$$

where  $r_{i,k}^{(t-1)}$  is the remaining availability. When a node  $n_i$  is added to the frontier, its availability values are initialized as  $r_{i,k} = 1$  for all memory elements  $k$ .  $\ell_{i,k}^{\text{base}}$  is the base action pointer logit generated as described in Equation 4.4. The parameter  $\alpha$  controls the strength of the availability penalty in the probability space.

We use  $\tilde{\ell}_{i,k}^{(t)}$  to compute the action-pointer distribution using Softmax as follows:

$$\begin{aligned} \pi_{i,k}^{(t)} &= \text{softmax}_k \left( \frac{\tilde{\ell}_{i,k}^{(t)}}{\tau_{\text{act}}} \right) \\ &= \frac{\pi_{i,k}^{\text{base}} (r_{i,k}^{(t-1)})^\alpha}{\sum_j \pi_{i,j}^{\text{base}} (r_{i,j}^{(t-1)})^\alpha}. \end{aligned}$$

The availability value  $r_{i,k}^{(t)} \in [0, 1]$  decreases as node  $i$  and memory element  $k$  are used. So, we scale the base selection probability  $\pi_{i,k}^{\text{base}}$  by its remaining availability  $(r_{i,k}^{(t-1)})^\alpha$ , and then renormalize over the memory elements. An unused action with  $r_{i,k}^{(t-1)} = 1$  retains its base weight, while an action that has contributed before is progressively downweighted. The parameter  $\alpha$  controls the strength of this penalty, and the renormalization redistributes the removed probability mass among other actions.

**Updating the Remaining Availability.** The remaining availability value is used to apply a soft mask that penalizes memory elements/actions that have been used before. Essentially,  $r_{i,k}^{(t)} \in [0, 1]$  quantifies the remaining availability of the node–action pair at time step  $t$  based on the pair’s contribution to generating successors before this time step.

Let  $w_{\text{sel},i}^{(t)}$  be the selection weight assigned to node  $n_i$  as given in Equation 4.7 and let

$$\pi_{i,k}^{(t)} = P(A_t = k \mid I_t = i)$$

be the conditional probability of selecting memory item  $e_k$  as the action from that node. The soft contribution attributed to the node–action pair is then

$$c_{i,k}^{(t)} = w_{\text{sel},i}^{(t)} \pi_{i,k}^{(t)} = P(I_t = i, A_t = k).$$

Before any availability update is applied to node  $n_i$ , its action distribution is the base distribution  $\pi_{i,k}^{\text{base}}$  defined in Equation 4.5.

We then update the remaining availability as follows:

$$r_{i,k}^{(t)} = r_{i,k}^{(t-1)} \cdot (1 - c_{i,k}^{(t)}).$$

Here, the term  $(1 - c_{i,k}^{(t)})$  denotes the fraction of previous availability that is retained after the current contribution at time step  $t$ . A node–action pair with zero new contribution remains unchanged, whereas a contribution approaching one removes most of its remaining availability. For example:

$$\begin{aligned} r_{i,k}^{(t-1)} = 0.5, \quad c_{i,k}^{(t)} = 0.1 &\implies r_{i,k}^{(t)} = 0.5 \cdot (1 - 0.1) = 0.45, \\ r_{i,k}^{(t-1)} = 0.5, \quad c_{i,k}^{(t)} = 0.9 &\implies r_{i,k}^{(t)} = 0.5 \cdot (1 - 0.9) = 0.05. \end{aligned}$$

Thus, an action is not removed after it has made a partial contribution. Instead, repeated contributions progressively reduce its probability when the same node is expanded again, enabling the model to utilize different actions.

### 4.3.8 Readout and Pooling Operation

After completing the search process, we obtain a set of search nodes. Although we treat them as an unordered bag of objects, there exists an implicit hierarchical relationship between the search nodes, in which later nodes are obtained by aggregating information from earlier nodes. We treat each search node as a representation that can be used to answer the decision question by comparing it with the goal. In other words, each search node can be viewed as a compressed and refined representation of a state, generated in a goal-directed manner and useful for answering whether the goal state is reachable. With this view, we treat the decision problem over search nodes as a classification problem under the Multiple Instance Learning paradigm described in Section 2.3.

More specifically, we use the instance-space paradigm defined in Section 2.3.1, where each search node is scored separately for classification. Instead of using the node features to directly classify whether the query has been reached, we use a matching function between each search node and the goal vector. This allows a search node to carry a compressed representation of the visited state without requiring it to contain explicit goal-reachability information, which is instead computed at the readout stage.

**Goal Matching.** We define a parameterized goal-matching function  $m : \mathcal{N} \times \mathcal{E} \rightarrow \mathbb{R}$  that takes a search node  $n_i \in \mathcal{N}$  and a query element  $q \in \mathcal{E}$  and returns a score that indicates whether the search node satisfies the query. We use a standard MLP with one hidden layer that takes a search node, the goal vector, and a manually defined feature interaction  $|q - n_i|$ , and outputs a logit. Although we believe that this manual feature interaction could be avoided by using a larger hidden dimension or more hidden layers, we retain it because the difference between the goal and node features provides useful information for the task we are solving. In addition, we exclude the start node from the goal-matching and pooling operations since it is not informative for answering the decision problem.

In this work, we use *LogMeanExp pooling* as the main pooling operation, as described in Section 2.3.1, whereas other operations such as *max pooling* and *attention-weighted pooling* serve as alternative methods that can be used in future work. Figure 4.4 shows the goal-matching and pooling operations used to obtain the final classification logit.

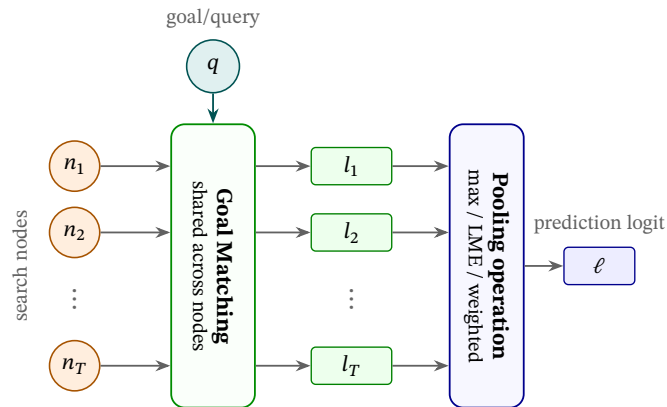


Figure 4.4: Query-conditioned readout. Each generated search node is scored against the query, and the resulting node logits are pooled into a single prediction logit.

Finally, we would like to highlight that the embedded-space paradigm (Section 2.3.2) is an interesting alternative in which we aggregate the information available in all search nodes and perform classification based on the aggregated representation. We use this in an ablation study of the pooling operation.

We present the computation performed by fixed-step ST in Algorithm 1. To keep the algorithm compact, we use named procedures for operations that are described in detail in the preceding sections.

## 4.4 Adaptive Computation for ST

We can divide the operations performed by ST into two phases: running the search for a fixed number of steps and performing the pooling operation using the goal-matching scores of the generated search nodes. The two phases are performed one after the other. A natural question that arises is: “Can we perform the goal-matching operation during the search instead of doing it at the end?” The dynamic halting approach for ST is motivated by this question.

Performing goal matching during search has several advantages. If we find a node that is sufficiently similar to the goal, then we can halt the process early, saving time and computation. In addition, the same mechanism can be used in the other direction: more complex problems might require more search steps, allowing us to scale the inference-time computation. It also aligns with many classical search algorithms where, upon visiting a node, it is tested against the goal to determine whether the goal has been reached. Then, if the node satisfies the goal conditions, the search process is terminated. At a high level, we adopt these ideas in the framework of ST to allow goal matching inside the search loop and to halt the search process when a node that is sufficiently similar to the goal is reached.

**Algorithm 1 Fixed-Step Search Transformer**


---

**Input:** problem context encodings  $\mathbf{E} = [e_1, \dots, e_K]^\top$ , query encoding  $q$ , search horizon  $T$   
**Output:** binary prediction  $\hat{y} \in \{0, 1\}$

```

1   $n_0 \leftarrow \mathbf{0}$ 
2   $\text{frontier} \leftarrow [n_0]$ 
3   $\text{base\_pointer\_logits} \leftarrow [\text{COMPUTEPOINTERLOGITS}(n_0, \mathbf{E}, q)]$ 
4   $\text{action\_availability} \leftarrow [\mathbf{1}_K]$ 
5  for  $t = 1, \dots, T$  do
6       $\text{sel\_weights} \leftarrow \text{COMPUSELECTIONWEIGHTS}(\text{frontier}, \mathbf{E}, q)$ 
7       $\text{action\_pointers} \leftarrow \text{APPLYACTIONMASK}(\text{base\_pointer\_logits}, \text{action\_availability})$ 
8       $\text{actions} \leftarrow \text{READMEMORY}(\text{action\_pointers}, \mathbf{E})$ 
9       $\text{successors} \leftarrow \text{PARALLELEXPAND}(\text{frontier}, \text{actions})$ 
10      $n_t \leftarrow \text{AGGREGATE}(\text{successors}, \text{sel\_weights})$ 
11      $\text{APPEND}(\text{frontier}, n_t)$ 
12      $\text{UPDATEAVAILABILITY}(\text{action\_availability}, \text{sel\_weights}, \text{action\_pointers})$ 
13     if  $t < T$  then
14          $\text{APPEND}(\text{base\_pointer\_logits}, \text{COMPUTEPOINTERLOGITS}(n_t, \mathbf{E}, q))$ 
15          $\text{APPEND}(\text{action\_availability}, \mathbf{1}_K)$ 

16   $s \leftarrow \text{QUERYMATCH}([n_1, \dots, n_T], q)$ 
17   $\ell_{\text{out}} \leftarrow \text{LOGMEANEXPPool}(s)$ 
18   $\hat{y} \leftarrow \mathbb{1}[\sigma(\ell_{\text{out}}) \geq 0.5]$ 
19  return  $\hat{y}$ 

```

---

We implement the dynamic halting mechanism using PonderNet [3] (see Section 2.4.1 for details), and we call this model ST-Ponder. Below, we describe the approach we use to equip ST-Ponder with the ability to predict the conditional halting probability and the decision output at each search step.

#### 4.4.1 Halting Head

We augment ST with a halting head that predicts the conditional halting probability at each step. We implement the halting head with a component that is very similar to the goal-matching MLP with a single hidden layer. At step  $t$ , the halting head takes the node  $n_t$  along with the goal/query  $q$  and returns the conditional halting probability:

$$\lambda_t = \sigma(\text{MLP}_{\text{halt}}([n_t; q])).$$

With  $\lambda_t$  as the conditional halting probability, we predict the unconditional halting distribution  $p$  using the approach described in Section 2.4.1 and use the *prefix normalization* approach to correct the truncated distribution.

#### 4.4.2 Kullback–Leibler (KL) Regularization

As in the PonderNet paper, we utilize KL regularization to bias the halting distribution toward a geometric prior distribution. We utilize this regularization especially at the start of training, when it is often advantageous to make sure most of the generated search nodes are used for output prediction. Without the regularization, the halting distribution can assign negligible probability mass to later nodes, leading to fewer effective search steps and reduced contribution from later nodes. We choose the parameter of the geometric prior

$$\lambda_p \approx \frac{1}{0.8T}$$

so that the mean of the untruncated geometric prior is approximately 80% of the total budget  $T$ . We gradually decrease the regularization weight  $\beta \rightarrow 0$  using cosine annealing with a predefined schedule, allowing the network to learn the halting distribution freely.

#### 4.4.3 Prediction Using Search Prefixes

At each search step, the model with dynamic halting needs to predict an output logit in addition to the halting probability. We obtain the output at step  $t$  by applying the same pooling operation used by fixed-step ST to all search nodes generated up to that step. Thus, the output at step  $t$  is the prediction that a fixed-step ST would make if its search were stopped after  $t$  steps.

Let  $n_1, \dots, n_T$  be the nodes generated after  $T$  search steps, excluding the start node  $n_0$ , and let

$$s_i = m(n_i, q)$$

be the goal-matching logit of node  $n_i$ . We define the output logit at step  $t$  as

$$\ell_t = \text{Pool}(s_1, \dots, s_t), \quad t \in [T],$$

where Pool is the LogMeanExp pooling operation described earlier. The resulting logits  $(\ell_1, \dots, \ell_T)$  are then used in the PonderNet objective.

An alternative approach is to use only  $s_t$ , the goal-matching logit of the node generated at search step  $t$ . However, this does not align with our view of the output at step  $t$  as the result of running the search with a budget of  $t$  steps, since a solution found earlier should still contribute to the prediction. In addition, prefix-based prediction converged more reliably in our preliminary experiments than this alternative approach, so we use it for ST-Ponder.

#### 4.4.4 Halting During Inference

At inference time, we need to use the halting probabilities predicted by the halting head  $\text{MLP}_{\text{halt}}$  to make the stopping decision. We adopt a threshold-based approach in which inference stops

once the cumulative halting probability reaches a predefined threshold. Let  $c_t$  be the cumulative halting probability:

$$c_t = \sum_{i=1}^t p_i = 1 - \prod_{i=1}^t (1 - \lambda_i).$$

Here,  $p_t$  denotes the probability of halting at step  $t$ , and  $c_t$  denotes the probability of halting by step  $t$ . Thus, it is intuitive to halt once the cumulative probability reaches a sufficient level. We compute the cumulative probability from the unconditional and pre-correction probabilities  $p_i$  since they are available sequentially during inference. The normalized finite-horizon distribution  $\tilde{p}$  requires predictions from all  $T$  steps and is therefore used only during training.

Using  $c_t$ , we define the halting step as

$$t_{\text{halt}} := \min(\{t \in \{1, \dots, T_{\text{max}}\} : c_t \geq \delta_{\text{halt}}\} \cup \{T_{\text{max}}\}).$$

$T_{\text{max}}$  is the maximum search budget allocated for inference, which can differ from the budget  $T$  used during training.  $\delta_{\text{halt}} \in (0, 1)$  is a hyperparameter for which higher values generally result in later halting and therefore more computation. If the cumulative halting probability does not reach the threshold within  $T_{\text{max}}$  steps, the halting step is  $t_{\text{halt}} = T_{\text{max}}$ .

At the halting step, ST-Ponder returns the prefix logit  $\ell_{t_{\text{halt}}}$ . Its positive-class probability is  $\sigma(\ell_{t_{\text{halt}}})$ . Using a probability threshold of 0.5, the binary prediction is

$$\hat{y} = \mathbb{1}[\sigma(\ell_{t_{\text{halt}}}) \geq 0.5] = \mathbb{1}[\ell_{t_{\text{halt}}} \geq 0].$$

## 4.5 Learning Objectives

We use only the final decision label as supervision. For fixed-step ST, we apply the Binary Cross-Entropy loss to the final output. ST-Ponder instead predicts an output at each search step and uses the PonderNet objective, which computes the expected task loss with respect to the predicted halting distribution, together with a KL-regularization term.

### 4.5.1 Task-Level Supervision

We define the problem as a binary classification or decision problem, where  $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N_{\mathcal{D}}}$  contains  $N_{\mathcal{D}}$  training examples, where  $x_i = (E_i, q_i)$  is the input instance, and  $y_i \in \{0, 1\}$  is the target or decision label; 1 and 0 indicate “yes” and “no” decisions, respectively. Let  $\ell_i$  be the (pooled) logit output for an input instance  $x_i$ . We then use Binary Cross-Entropy (BCE) to define the task loss as follows:

$$\mathcal{L}_{\text{task}}(x_i, y_i, \ell_i) = -[y_i \cdot \log \sigma(\ell_i) + (1 - y_i) \cdot \log(1 - \sigma(\ell_i))]. \quad (4.8)$$

Here,  $\sigma$  is the logistic sigmoid function.

#### 4.5.2 Dynamic Halting Objective

We adopt the training objective of PonderNet [3] to train ST with dynamic halting. Let  $\tilde{p} = (\tilde{p}_1, \dots, \tilde{p}_T)$  be the normalized halting distribution and  $\ell = (\ell_1, \dots, \ell_T)$  be the output logits for every time step  $t \in [T]$ . We define the objective function as the expected loss with respect to the halting distribution:

$$\mathcal{L}_{\text{ac}}(x_i, y_i, \tilde{p}, \ell) = \sum_{t=1}^T \tilde{p}_t \mathcal{L}_{\text{task}}(x_i, y_i, \ell_t) + \beta \text{KL}(\tilde{p} \parallel \tilde{p}_G(\lambda_p)). \quad (4.9)$$

Here,  $\tilde{p}_G(\lambda_p)$  is the finite-horizon geometric prior defined in Section 2.4.1.

## 4.6 Training

We present three methods used during training to improve convergence and stability. Curriculum learning is used for all models, while fixed-step ST uses search-horizon scaling and all ST variants use entropy regularization of node selection.

### 4.6.1 Curriculum Learning

For all training runs, we employ curriculum learning. For most configurations, the rungs are ordered by problem depth. For the Disjoint Path-EC profile, we order them instead by the pair  $(B, D)$ , where  $B$  is the deceptive branch count and  $D$  is the problem depth. Within each fixed value of  $B$ , the curriculum advances through all depths before increasing  $B$ . We make all valid deceptive-prefix lengths  $P$  available within each  $(B, D)$  rung, so  $P$  is not a curriculum dimension. Table 4.1 summarizes the resulting rung order.

Table 4.1: Ordered curriculum rungs used for each training configuration. In the Disjoint profile,  $B$  is the deceptive branch count and  $D$  is the problem depth.

| Configuration             | Ordered curriculum rungs                                                                                      |
|---------------------------|---------------------------------------------------------------------------------------------------------------|
| LD Easy, Medium, and Hard | $D = 2 \rightarrow 3 \rightarrow 4$                                                                           |
| Path-EC Disjoint          | $(B, D)$ : for $B = 0, 1, 2$ in this order, $D = 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$ within each $B$ |
| Path-EC Convergent        | $D = 4 \rightarrow 5 \rightarrow 6$                                                                           |
| Path-EC State-Alias       | $D = 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$                                                             |

As described in Section 2.6, we use a performance-based curriculum scheduler. When the training accuracy on the cumulative active training set reaches a predefined threshold of 0.9 over an epoch, the curriculum rung increases by one. Upon reaching a higher rung, we retain all examples from earlier rungs. Hence, the amount of available data keeps increasing until the model reaches the highest curriculum rung.

### 4.6.2 Scaling Search Horizon

ST allows scaling the search during both training and inference. Increasing it during inference can be useful for solving more complex problems. Increasing it during training, especially at the early phase, gives the network more opportunities to explore different selection and transition dynamics, which might lead to faster convergence. For fixed-step ST, we multiply the configured search horizon  $T$  by a factor and anneal this factor to 1 over the first 12,000 training steps using a cosine schedule. The search horizon scaling showed better convergence in our preliminary experiments, therefore we employ it for the fixed-step ST runs.

### 4.6.3 Entropy Regularization for Node Selection

During the early phase of training, the node selection policy may prematurely focus on a single search node before it has learned a useful node selection. To avoid this, we encourage early exploration by maximizing the normalized entropy of the selection distribution. Let  $p_{b,t}$  be the predicted node-selection distribution for minibatch example  $b$  at search step  $t \in \{2, \dots, T\}$ , over the  $|N_t| = t$  search nodes available at that step. Let  $\mathcal{L}_{\text{base}}$  denote  $\mathcal{L}_{\text{task}}$  for fixed-step ST and  $\mathcal{L}_{\text{ac}}$  for ST-Ponder. We define the regularized training objective as:

$$\mathcal{L} = \mathcal{L}_{\text{base}} - \lambda_{\text{ent}}(s) \frac{1}{T-1} \sum_{t=2}^T \mathbb{E}_b \left[ \frac{H(p_{b,t})}{\log |N_t|} \right], \quad H(p) = - \sum_i p_i \log p_i.$$

Here,  $s$  is the training step and  $\lambda_{\text{ent}}(s) \geq 0$  is the entropy-regularization weight. Since the objective is minimized, subtracting the entropy term encourages a higher selection entropy. For all training runs, we decrease  $\lambda_{\text{ent}}(s)$  from 0.01 to zero over the first 12,000 training steps using cosine annealing. Thus, the selection distribution starts diffuse and is later allowed to become sharp depending on the task requirements. We obtained this hyperparameter through a few experiments on selection collapse and did not tune it further.

## 5 Results

This chapter presents the experimental setup, datasets, training configurations, results, and analysis of various experiments. We position ST as an alternative architecture to UT for performing reasoning in a latent space with Transformer-based components. So, we compare their performance in two task domains to understand their advantages and disadvantages. A high-level question we want answered here is whether ST can match or exceed the performance of UT on in-distribution and out-of-distribution tasks. We also implement ST-Ponder and perform studies to understand its halting behavior. More specifically, we want to understand whether ST-Ponder can utilize more compute and improve the performance of ST on the OOD test sets.

We evaluate both models on three logical-deduction datasets with increasing difficulty and three Path-EC dataset profiles. The Path-EC profiles are not intended to form a difficulty hierarchy; rather, they present different challenges in graph search. All main experiments are performed with five random seeds and we present the mean and standard deviation of the performance scores, along with seed-wise results when relevant. All runs had a maximum budget of 1 million training steps. With a batch size of 128, each model is exposed to up to 128 million examples. Though our training dataset sizes are on the order of tens of thousands, we use a data augmentation method with random relabeling, which exposes the model to unique examples that are helpful for learning important invariances. Although all training runs have the same maximum budget, we stop a run to save computation when its ID and OOD validation-set accuracies do not improve over the last 5,000 training steps or start to degrade.

We first provide the configuration and setup we use for running an experiment and evaluating its performance. Then, we provide the main results with a detailed analysis of each dataset class. Next, we study the effect of inference-time computation on ST using different fixed search budgets and adaptive computation. Finally, we present the ablation studies to understand the importance of different components of ST.

### 5.1 Experimental Setup

This section describes how ST and UT were trained and evaluated along with their model configurations, benchmark settings, training procedure, and evaluation metrics. The information provided in this section, combined with additional information in the appendix, allows readers to reproduce our results.

### 5.1.1 Benchmark Configuration

We use two synthetically generated dataset classes to evaluate the performance of ST and UT. Descriptions of the two datasets are provided in Section 2.7, and additional information is available in Appendix A.1. The two datasets evaluate different capabilities. Logical deduction evaluates the models’ ability to perform multi-step rule composition to decide whether the query is provable. Path exact cover evaluates the models’ ability to perform graph search, with each dataset profile evaluating different structural requirements, including deceptive paths, convergent paths, and path-dependent state aliases.

Table 5.1: Dataset configurations used for training and evaluation. The in-distribution (ID) depths are the problem depths used for training, whereas the out-of-distribution (OOD) depths correspond to problems requiring more steps to solve than the ID problems.

| Dataset             | ID depths | OOD depths |
|---------------------|-----------|------------|
| LD Easy             | 2–4       | 5          |
| LD Medium           | 2–4       | 5–8        |
| LD Hard             | 2–4       | 5–12       |
| Path-EC Disjoint    | 3–6       | 7–8        |
| Path-EC Convergent  | 4–6       | 7–8        |
| Path-EC State-Alias | 3–6       | 7–8        |

We do not use problems of depth less than 2 for logical deduction as they do not require composing more than one rule, so no internal representation is needed. Similarly, the depth in Path-EC problems controls the size of the graph, so we choose the minimum depth that allows us to encode the profile-specific property in the graph.

**Depth and generalization.** We provide a detailed description of *depth* for each problem in Section 2.7. In simple terms, *depth* in logical deduction determines the proof length for positive examples, whereas for negative examples, depth is a remaining artifact of their generation. Depth for Path-EC is the size of the target cover set or the path length from the source to the target node. We verify solvability of all positive and negative examples using a symbolic solver. We refer to the set of problems with depths not seen during training as OOD depths, or simply OOD. We also use the term extrapolation to refer to the model’s ability to solve OOD problems. For positive logical-deduction examples, OOD problems require more steps to derive the query, and in Path-EC, they require information propagation over a longer path between source and target. Here, we give a symbolic interpretation in which each step applies a correct rule or traverses the correct path; the actual behavior that the model learns can differ, and we analyze it below. We emphasize the model performance on OOD test sets since one of our hypotheses is that using a search inductive bias can enable learning useful dynamics and components with extrapolation capability.

### 5.1.2 Compared Models

We compare the performance of *ST* and *UT* with comparable parameter counts, where both are given a fixed budget of 16 iterations. So, *ST* generates 16 successor nodes that are used to produce its output, and *UT* updates its input embeddings 16 times using the attention mechanism and an MLP. We would like to highlight that even though the two models use the same iteration count, the computation performed by them differs. So, our comparison mainly allows a comparable number of sequential computations for the two architectures. We also compare the performance of *ST-Ponder* with fixed-step *ST* to evaluate whether adaptive computation is useful or not.

The fixed-step *ST* and *UT* variants have approximately 1.9–2.0M and 1.8–1.9M parameters, respectively, where the variability comes from the feature projection that depends on the specific dataset. *ST-Ponder* adds a halting head of approximately 390K parameters. Each dataset type has a different feature dimension, so the projection layer depends on the dataset and model dimension.

We train all models with a fixed 16-step loop. This is 4× the maximum training depth in logical deduction and approximately 2.7× the maximum training depth in Path-EC. So, it should provide the models with enough recurrence to solve ID tasks. *ST-Ponder* is allowed a maximum of 32 steps during inference, and its actual number of iterations depends on the halting prediction. All models receive the same input obtained by encoding the problems. Since the feature ordering does not matter, we use no positional embeddings in any of the models.

### 5.1.3 Evaluation Metrics

This section describes the metrics used for model evaluation for both logical deduction and Path-EC. In all cases, we use the threshold value of 0.5, so a prediction is labeled 1 when it exceeds or matches the threshold; otherwise, it is labeled 0.

**Pair Accuracy (PA).** This metric applies to Path-EC, where we generate tasks as pairs of positive and negative examples. Each pair is always assigned to the same dataset split. For a dataset of pairs  $\mathcal{P}$ , we compute the pair accuracy as follows:

$$\text{PA}(\mathcal{P}) = \frac{1}{|\mathcal{P}|} \sum_{(x^+, x^-) \in \mathcal{P}} \mathbb{I}[f_\theta(x^+) = 1] \mathbb{I}[f_\theta(x^-) = 0].$$

Here,  $x^+$  and  $x^-$  denote the positive and negative examples in a pair, respectively, and  $f_\theta(x) \in \{0, 1\}$  denotes the predicted class. Thus, the PA of a dataset is the fraction of pairs for which both the positive and negative examples are classified correctly.

**Accuracy (Acc.).** Accuracy measures the fraction of predictions that are correct. All of our main datasets are balanced, so it is also the balanced accuracy.

**True Positive Rate (TPR).** Also called positive recall, it measures the fraction of positive examples that are classified correctly.

**True Negative Rate (TNR).** Also called specificity, it measures the fraction of negative examples that are classified correctly.

Let TP, TN, FP, and FN denote true positive, true negative, false positive, and false negative predictions, respectively. Then,

$$\text{Acc.} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}, \quad \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}}.$$

**Area under the ROC curve (AUROC).** AUROC measures the likelihood of assigning a higher score to a randomly chosen positive example than to a randomly chosen negative example. AUROC tells us about the separation of the two classes, where a higher value means better separation. So, if a model performs poorly in terms of accuracy but has high AUROC, its ranking remains strong despite the chosen classification threshold. This pattern can indicate threshold miscalibration, which in some cases might be mitigated by calibrating the threshold using the validation set.

#### 5.1.4 Training and Optimization

We present the base configuration used for all experiments in Table 5.2. Unless otherwise stated, these values are used throughout all experiments.

Table 5.2: Base training and optimization settings, listed in approximate order of importance for interpreting and reproducing the training procedure.

| Configuration             | Value                                                                                                          |
|---------------------------|----------------------------------------------------------------------------------------------------------------|
| Optimizer                 | AdamW [19]                                                                                                     |
| Base learning rate        | $3 \times 10^{-5}$                                                                                             |
| Learning-rate schedule    | Linearly increased from $10^{-8}$ to $3 \times 10^{-5}$ over 1,500 steps.                                      |
| Primary objective         | BCE                                                                                                            |
| Regularization            | Selection entropy regularization for ST: cosine annealing from 0.01 to 0.0 in the first 12,000 training steps  |
| Batch size                | 128                                                                                                            |
| Training budget           | Until convergence, with a maximum of 1 million training steps                                                  |
| Weight decay              | 0.1                                                                                                            |
| Curriculum learning       | Enabled and configured per dataset                                                                             |
| Gradient clipping         | Enabled with maximum norm 1.0                                                                                  |
| Search-horizon curriculum | Enabled for fixed-step ST; $2\times$ the number of steps at the start, annealed to $1\times$ over 12,000 steps |
| Numerical precision       | TF32 enabled                                                                                                   |

### 5.1.5 Hyperparameters

We performed a basic hyperparameter search for both models separately using a single base dataset from each dataset class: LD Medium for deduction and Disjoint for Path-EC. We performed the hyperparameter search for learning rate, weight decay, and model dimensions. For each parameter, three runs with different seeds were launched, and we selected the parameter by manually analyzing the training and validation performance curves. For the learning rate, we tested values  $10^{-3}$ ,  $3 \times 10^{-4}$ ,  $10^{-4}$ ,  $3 \times 10^{-5}$ ; for weight decay, we tested  $10^{-3}$ ,  $5 \times 10^{-2}$ , 0.1. For model parameters, we varied the model dimension and the hidden layer dimension of the Transformer MLP for ST and UT differently, and we chose configurations for the two models that had good performance while having comparable parameter counts. For UT, we performed the search over [(192, 512), (256, 1024), (384, 1536)], and for ST, we searched over [(192, 512), (256, 1024)], where the first item in the tuple represents the model dimension and the second item represents the MLP’s hidden-layer dimension.

For ST, we performed many iterations of architectural search over different components, gating operations, goal-matching functions, pooling operations, and action representations during the development phase. We do not include these experiments here, as they led to the final concrete ST we present in this thesis.

**Implementation.** We implemented all models in PyTorch and ran each experiment on a single GPU. Runs used either an NVIDIA A10 24 GB or an NVIDIA L40S 48 GB, assigned based on their availability in the cluster, while the numerical precision, batch size, and optimization settings were kept unchanged. We used Hydra for experiment configuration and uv for dependency management. Training metrics and checkpoints were monitored with Weights & Biases. Detailed information on the implementation environment and softwares is provided in Appendix A.2.2.

## 5.2 Performance Comparison

This section provides a comprehensive performance comparison between ST and UT on all datasets. In some analyses, we choose a specific dataset that is appropriate for our study or useful to test our hypothesis.

### 5.2.1 Overall ID and OOD Performance

Table 5.3 shows the overall performance of ST and UT for all the datasets. Each experiment was repeated with five seeds, and we present the average accuracy and standard deviation over completed runs. We mark a run as complete if it reaches the final curriculum rung. A run remains incomplete if it fails to reach the training accuracy threshold of 0.9 on a curriculum rung and therefore does not advance to the final rung within the training budget. We report

Table 5.3: Overall accuracy comparison of ST and UT on different dataset profiles. *Acc.* is the model accuracy and *Comp.* is the number of completed runs out of five seeds. Accuracy entries report the mean and standard deviation across completed runs; when only one run completed, only its accuracy is shown.

| Dataset                          | UT                |       | ST                |       |
|----------------------------------|-------------------|-------|-------------------|-------|
|                                  | Acc.              | Comp. | Acc.              | Comp. |
| <b>In-distribution (ID)</b>      |                   |       |                   |       |
| LD Easy                          | $0.970 \pm 0.001$ | 5/5   | $0.998 \pm 0.001$ | 5/5   |
| LD Medium                        | $0.994 \pm 0.007$ | 5/5   | $0.999 \pm 0.000$ | 5/5   |
| LD Hard                          | $0.992 \pm 0.005$ | 5/5   | $0.999 \pm 0.000$ | 5/5   |
| Path-EC Disjoint                 | <b>0.992</b>      | 1/5   | $0.893 \pm 0.008$ | 5/5   |
| Path-EC Convergent               | $0.998 \pm 0.001$ | 5/5   | $0.998 \pm 0.002$ | 5/5   |
| Path-EC State-Alias              | $0.996 \pm 0.003$ | 5/5   | $0.998 \pm 0.002$ | 5/5   |
| <b>Out-of-distribution (OOD)</b> |                   |       |                   |       |
| LD Easy                          | $0.928 \pm 0.002$ | 5/5   | $0.985 \pm 0.006$ | 5/5   |
| LD Medium                        | $0.784 \pm 0.056$ | 5/5   | $0.900 \pm 0.008$ | 5/5   |
| LD Hard                          | $0.642 \pm 0.069$ | 5/5   | $0.752 \pm 0.011$ | 5/5   |
| Path-EC Disjoint                 | <b>0.950</b>      | 1/5   | $0.897 \pm 0.003$ | 5/5   |
| Path-EC Convergent               | $0.711 \pm 0.048$ | 5/5   | $0.629 \pm 0.098$ | 5/5   |
| Path-EC State-Alias              | $0.986 \pm 0.014$ | 5/5   | $0.967 \pm 0.039$ | 5/5   |

performance only over completed runs because incomplete runs have not been exposed to all training examples. We report the number of completed runs separately to indicate how reliably each model completes training.

The in-distribution performance of ST and UT is comparable for the logical-deduction tasks, whereas ST consistently outperforms UT on the OOD test sets of all three datasets. The ID performance on the Path-EC Convergent and State-Alias profiles is comparable for the two models. For the Disjoint profile, UT performs better than ST on both the ID and OOD test sets; however, only a single UT run was complete, whereas all five ST runs were complete. The OOD performance of ST and UT on the Path-EC datasets tells a different story, where UT seems to perform well on all of them. To understand the discrepancy between the two, we perform dataset-specific analyses below.

### 5.2.2 Logical Deduction: LD Hard

The logical-deduction dataset measures the model’s ability to compose proofs using the available facts and rules. We present the models’ performance on the LD Hard dataset since it provides the widest OOD range, from 5 to 12, which can reveal where the models’ performance begins to struggle with extrapolation. The results on LD Easy and LD Medium are available in Appendix A.3.1.

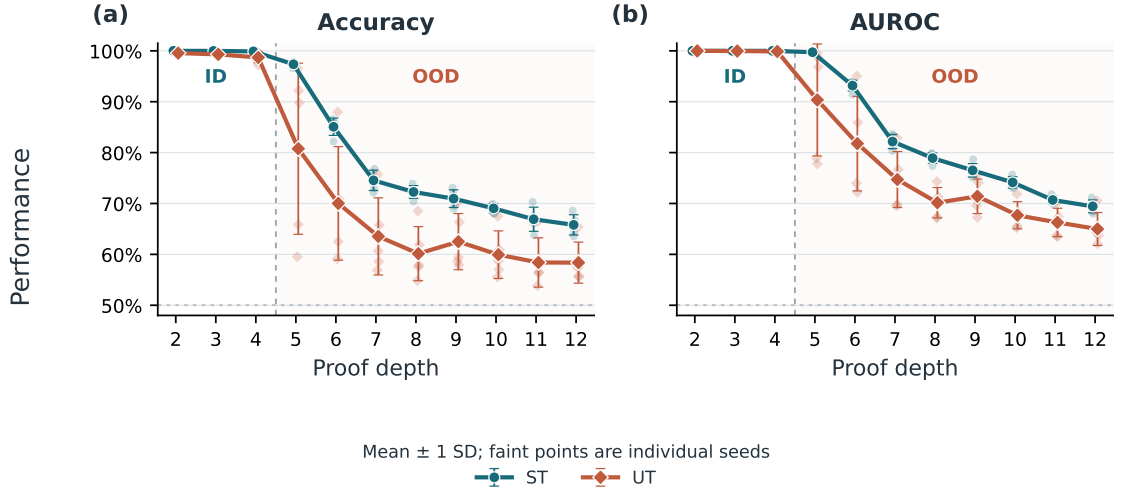


Figure 5.1: Depth-wise performance comparison on LD Hard. Plots report the mean and one standard deviation across five seeds. (a) reports accuracy, and (b) reports the area under the ROC curve.

Both accuracy and AUROC, averaged over seeds, are higher for ST at all depths. UT shows higher variance across seeds. Importantly, the higher AUROC for ST shows that ST, on average, has learned better class separation, especially for OOD examples, while the ID performance is almost the same for both models.

**Post-hoc analysis of best runs.** Since UT shows high variance in OOD performance across seeds, we perform a post-hoc analysis to see whether UT can achieve similar performance to ST on the OOD test set.

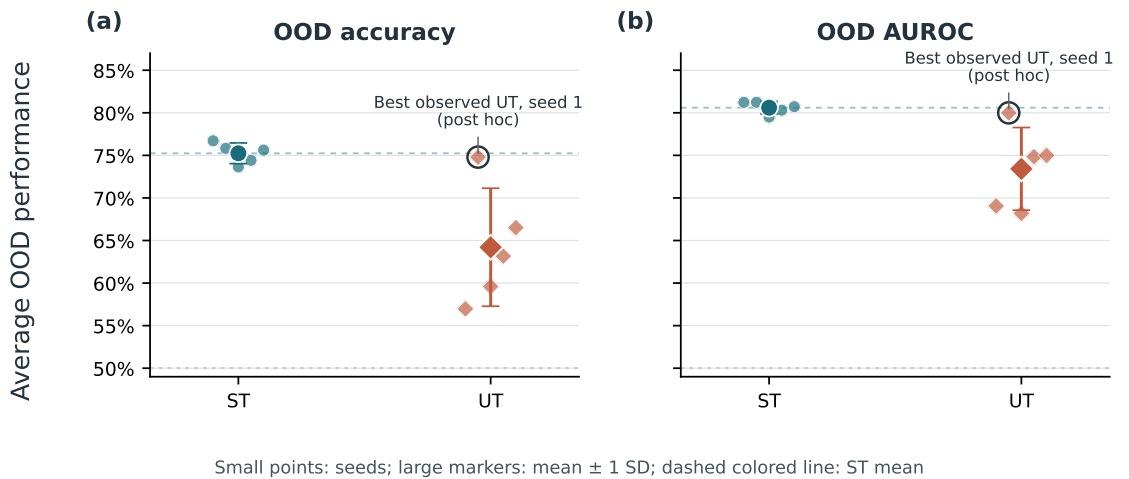


Figure 5.2: Average performance on the OOD test set over different seeds. Small points represent individual seeds, while large markers and whiskers show the mean and one standard deviation across seeds.

Figure 5.2 shows that one run of UT with seed 1 performs similarly to ST on the OOD test set, whereas almost all runs of ST perform at around the same level. It highlights that UT is a capable architecture for solving the logical deduction task, but it is more uncertain whether it will find a generalizable solution or not.

**Performance under query substitution.** To recap, the decision problems we tackle in this work are of the form  $(c_n, q_n)$ , where  $c_n$  is the problem context and  $q_n$  is the query. For logical deduction, the context consists of facts and rules, and the query is a single atom. Using an existing test example, we can generate new problems by only changing the query.

Let  $A_n = \{a_1, \dots, a_K\}$  be the set of unique atoms present in an example  $e_n = (c_n, q_n)$  where  $c_n = (F_n, R_n)$  is the problem context with facts  $F_n$  and rules  $R_n$  and  $q_n$  is its query atom. Then, let

$$B_n = A_n \setminus F_n = \{b_1, \dots, b_M\}.$$

We then obtain  $M$  problems by substituting each candidate atom  $b_i$  for the query:

$$e_n^{(i)} = (c_n, b_i), \quad i \in \{1, \dots, M\}.$$

Here, the context  $c_n$  remains the same while we only change the query to one of the candidate atoms. We exclude facts from the query since positive training examples all use derived atoms as their queries, and including facts can obscure performance on actual deduction.

For the problem  $e_n$ , we then run a forward-chaining algorithm to obtain all derivable atoms. Let  $D_n = \{r_1, \dots, r_L\}$  be the set of derivable atoms with  $d(r_i)$  representing the number of steps required to derive them. Then, we assign each  $e_n^{(i)}$  the label 1 with a depth value of  $d(b_i)$  if  $b_i \in D_n$ ; otherwise, we assign the label 0. We refer to the examples with label 1 as derivable and those with label 0 as non-derivable.

Figure 5.3 (a) shows that ST performs consistently better than UT at identifying both derivable and non-derivable queries. ST obtains a balanced accuracy of  $96.6 \pm 0.8\%$ , in contrast to  $77.6 \pm 8.1\%$  for UT. Also, UT shows higher performance variability across seeds.

Figure 5.3 (b) shows that for test problems with derivable queries, ST consistently outperforms UT with lower seed sensitivity. Depths 2–4 are ID depths used for training. Even at these depths, UT shows lower performance than ST. This means that most UT runs that previously achieved perfect ID performance have learned less robust computations that do not transfer well, whereas almost all ST runs transfer better to problems generated in a different way. After depth 5, UT’s performance falls below the random accuracy, meaning that the model is biased toward negative predictions.

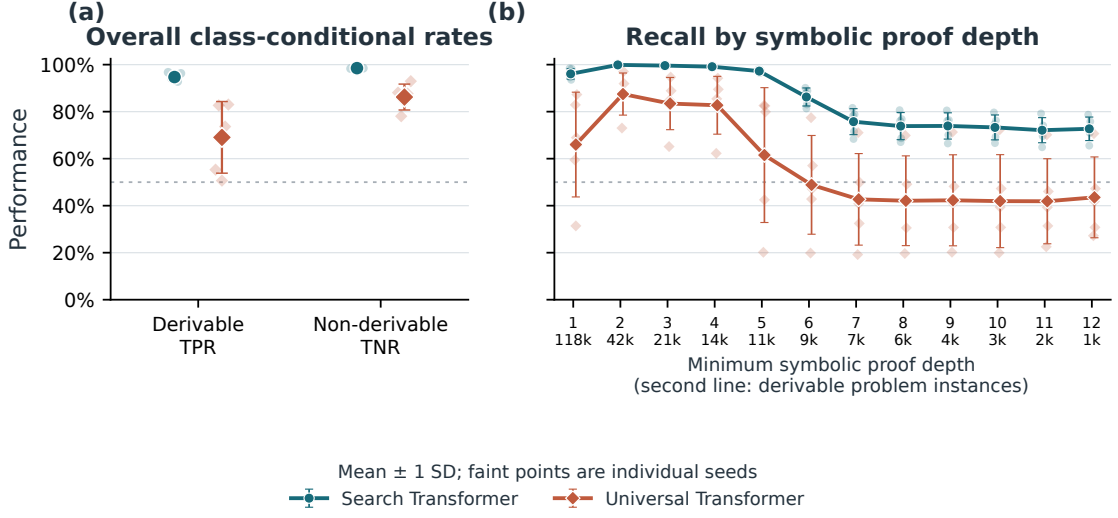


Figure 5.3: Performance comparison on test problems obtained with query substitution. (a) Overall recall for derivable queries and specificity for non-derivable queries. (b) Derivable query recall for each proof depth. Small points denote five independently trained seeds; large markers or lines show the mean, and whiskers indicate one standard deviation. ST maintains higher recall with substantially lower seed sensitivity, whereas UT becomes less reliable on deeper proofs.

**Convergence speed and sample efficiency.** In addition to this robustness, ST and UT also differ in how fast they converge.

Figure 5.4 shows that both architectures eventually reach a sustained balanced accuracy of at least 90% on the LD Hard dataset. However, across five seeds, ST requires a median of 3.84 million training examples to reach this criterion, compared with 53.76 million for UT. This corresponds to a 14-fold difference between the medians. UT also shows higher seed-to-seed variance in performance during training. Furthermore, ST starts improving from the random level very early on. In contrast, the performance of UT remains at a random level even after seeing more than 2 million examples. Hence, under the current training setup, ST is more sample-efficient than UT.

**Evidence Position.** When performing a  $T$ -step search with ST, we can treat the output at any step  $t \leq T$  as final and perform the classification. Using the predicted logit at each step, we define a step as a stable-positive evidence position for an example if its prediction logit crosses the zero threshold at this time step and remains positive for the remainder of the search. This means that the prediction at each later step also remains positive. For this analysis, we double the number of search steps from the standard 16 steps to 32.

Figure 5.5 shows the fraction of examples with stable-positive evidence for each step. Up to depth 7, we see a clear correlation between the problem depth and evidence position. For

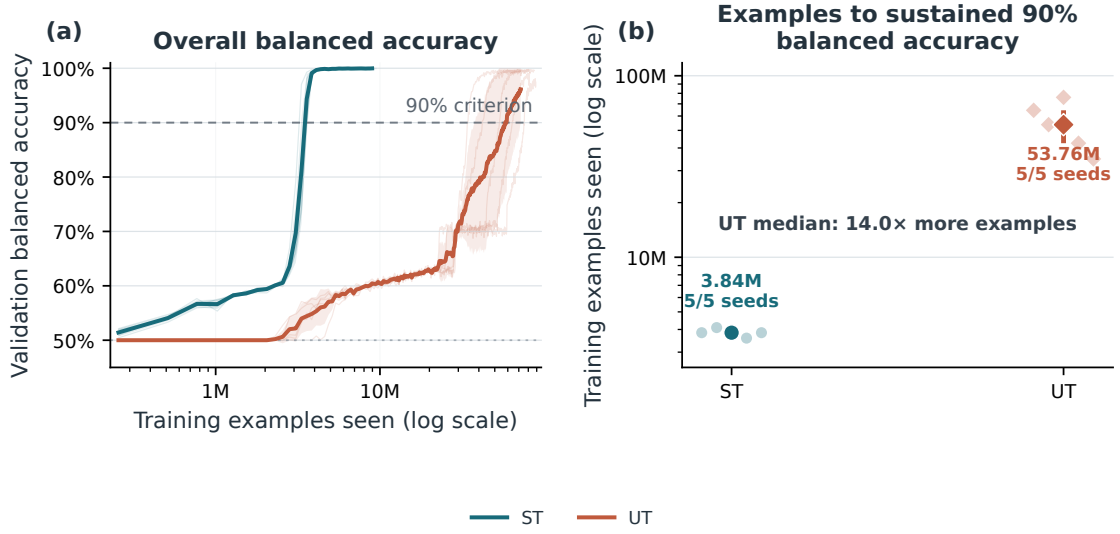


Figure 5.4: Performance on the validation set at different training steps. (a) Balanced accuracy on the validation set over the number of training examples seen. The faint lines represent individual-seed performance. The bold curves and shaded areas show the mean and one standard deviation across seeds. (b) The number of examples required to exceed 90% validation accuracy for each seed. We report the median across the five seeds.

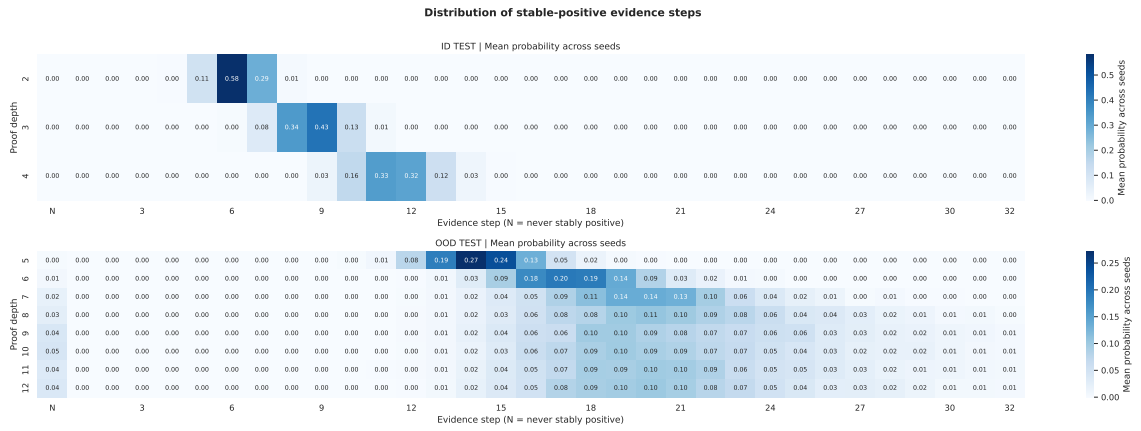


Figure 5.5: Distribution of the search step at which the prediction first becomes stably positive for LD Hard problems, grouped by proof depth. Each cell shows the mean probability across seeds.  $N$  denotes problems for which the prediction never becomes stably positive. The upper and lower panels show the ID and OOD test sets, respectively.

problems of depth  $\geq 8$ , the evidence position is more diffuse and unclear. Since we see that at lower depths the model finds evidence earlier, it could have stopped the search process earlier than waiting for all 32 steps, motivating the adaptive computation approach for ST.

### 5.2.3 Path Exact-Cover

Here, we compare the performance of ST and UT on the Path-EC datasets. We are more interested in their performance on the *Disjoint* profile, which contains multiple deceptive branches that can have different lengths.

#### Disjoint Profile

The Disjoint Path-EC dataset contains multiple disjoint linear paths from the source node to the target node. We fix the number of paths to 4. In addition, we make a few of those paths “deceptive,” where a deceptive path contains consecutive items that belong to the target cover. So, looking from the start node, a deceptive path looks similar to a true path. For each example, we fix the length of the deceptive path that looks valid. So, a deceptive prefix of size 1 contains the first item on the path that belongs to the cover, and no subsequent item does. The solution can be obtained by starting from the start node or from the target node, so to make the deceptive paths available for both approaches, we make them symmetric. Hence, the dataset complexity for the Disjoint profile varies along three axes: problem depth  $d$ , deceptive branch count  $B$ , and deceptive prefix length  $P$ . This factorization allows us to understand where ST is able to work and where it fails, and may reveal interesting things. But, it also adds complexity to the analysis.

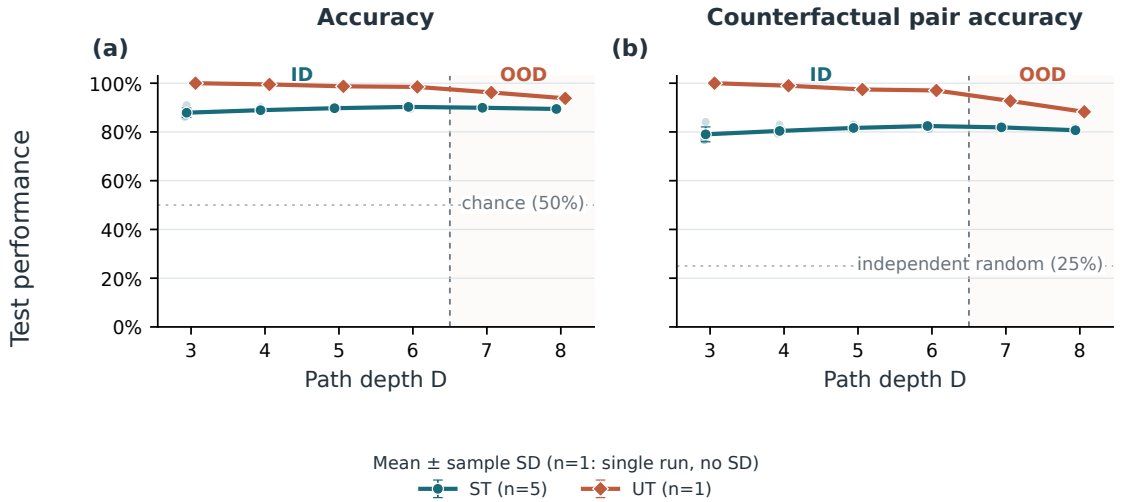


Figure 5.6: Performance comparison on the Disjoint dataset. All five ST runs and only a single UT run completed the curriculum. We include results only from completed runs here. (a) shows accuracy, and (b) shows pair accuracy.

Figure 5.6 shows that UT outperforms ST consistently across all depths. As the depth increases, we observe that the performance of UT decreases, with a larger decline on OOD examples. In contrast, ST shows a slight improvement in accuracy as depth increases. A similar trend is visible in the pair accuracy plot as well. This suggests that ST has learned a reproducible but lower-performing solution, whereas the evaluated UT run learns a substantially stronger solution that degrades more under depth extrapolation.

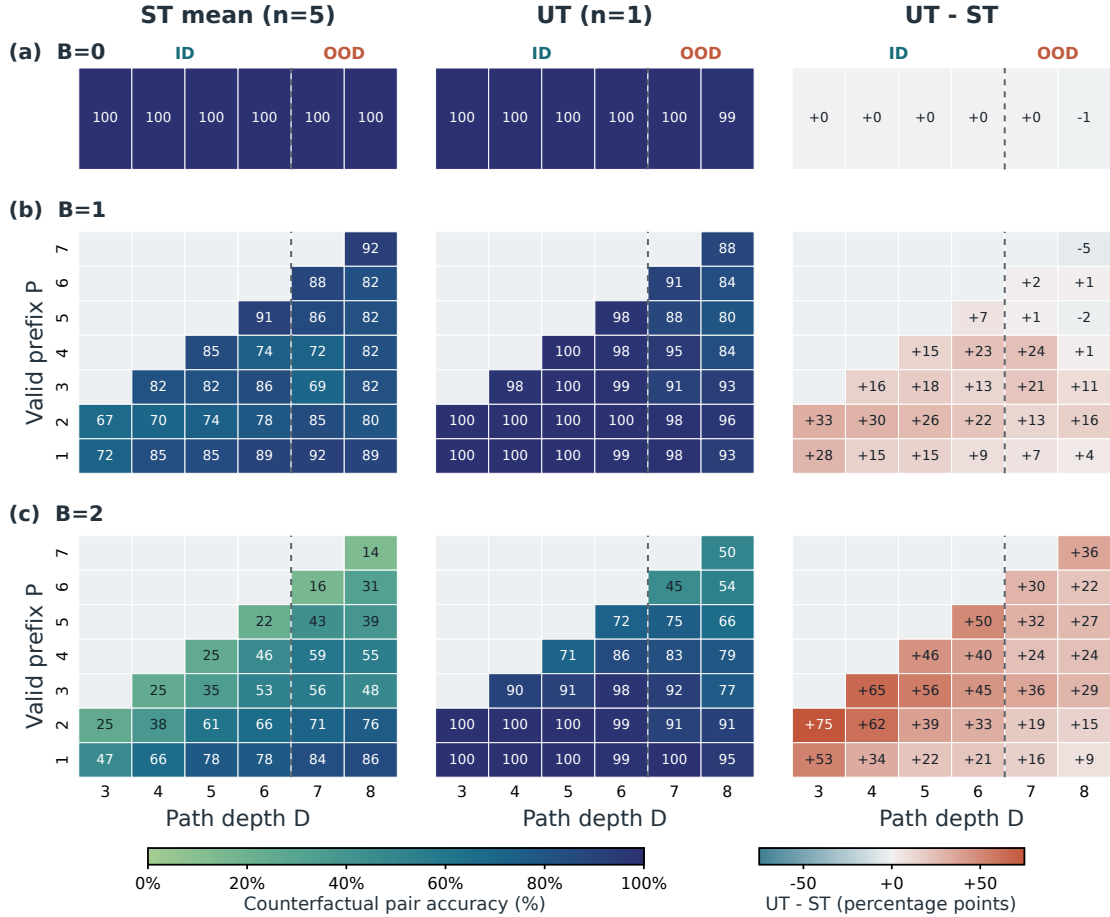


Figure 5.7: Pair accuracy on the Path-EC Disjoint profile by depth, branch count, and prefix length.

Figure 5.7 presents the breakdown of the pair accuracy. The performance of both models on  $B = 0$  is comparable, meaning both models are able to identify the existence of a path containing the set of items when there are no deceptive paths. For  $B = 1$ , the ID performance of UT is very strong, with slight degradation on OOD examples as the deceptive-prefix length increases. ST, however, shows a modest and non-intuitive pattern in which, for a fixed  $P$ , the performance increases as we increase the depth, and for a fixed depth, the performance first degrades and then increases as  $P$  increases. It implies that ST may be performing global information aggregation to solve the problem, rather than a search process in which a longer

deceptive prefix and greater depth require longer searches that generally result in degraded representations and lower performance. For  $B = 2$ , the ID performance of UT is still strong with degradation as the deceptive prefix length increases. ST also shows the same trend, but with much lower overall performance.

**Counterfactual path-splice negatives.** For the Disjoint profile, we first generate a positive example. Then, we obtain its paired negative example by swapping one item from the solution path with one item from a randomly chosen edge that is not part of the solution path. To evaluate whether ST has learned to capture statistics specific to the way we generate the negative example, we change our approach to negative-example generation. We take the positive example and cut the solution path into two segments at a random point. Then, we randomly choose another path from the source to the target and cut it at the same point. This is possible since all paths are of the same length. Then, we make crisscross connections between the two paths. Now, the positive example and its paired negative example, which we call a “path-splice” pair, contain the same statistics about the edges, nodes, and edge items, but only the label changes. We also validate the label using a symbolic solver.

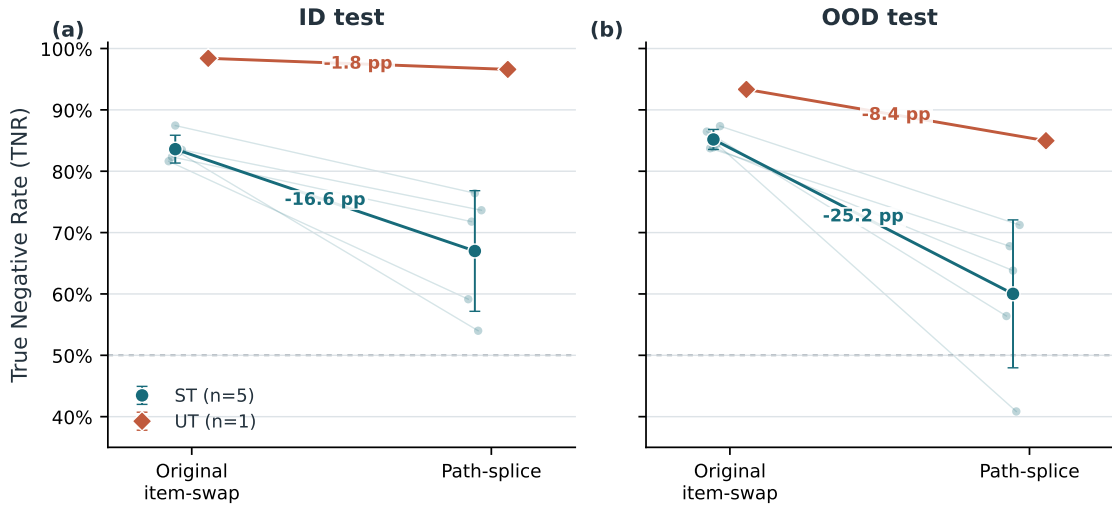


Figure 5.8: True negative rate on original and path-splice negative test examples. Five completed runs were used for ST and a single run for UT.

Figure 5.8 shows that UT retains the ability to detect differently generated negative examples better than ST. ST shows high variance in the change in performance, with some runs falling close to random guessing. The drop increases for both models on OOD examples, but the drop is larger for ST. This means that, although ST shows better convergence on the problem, UT’s converged run shows a better ability to distinguish the negative examples. We hypothesize that this might be due to the soft selection of nodes and actions, where aggregating the successors of multiple nodes creates a state that might make it difficult to distinguish whether the node was generated by actually accumulating the objects along the path or only through node

aggregation. Enabling ST to filter the relevant information obtained from the action pointer and perform a more discrete selection by annealing the selection temperature or using straight-through Gumbel-Softmax might be appropriate for the Path-EC problem. We leave these problem-dependent modifications for future work.

**Evidence Position for Disjoint  $B = 0$ .** Similar to the logical-deduction analysis, we also evaluate whether there exists a correlation between the number of search steps and the problem depth. We only consider the dataset with no deceptive branches, where ST performs quite well.

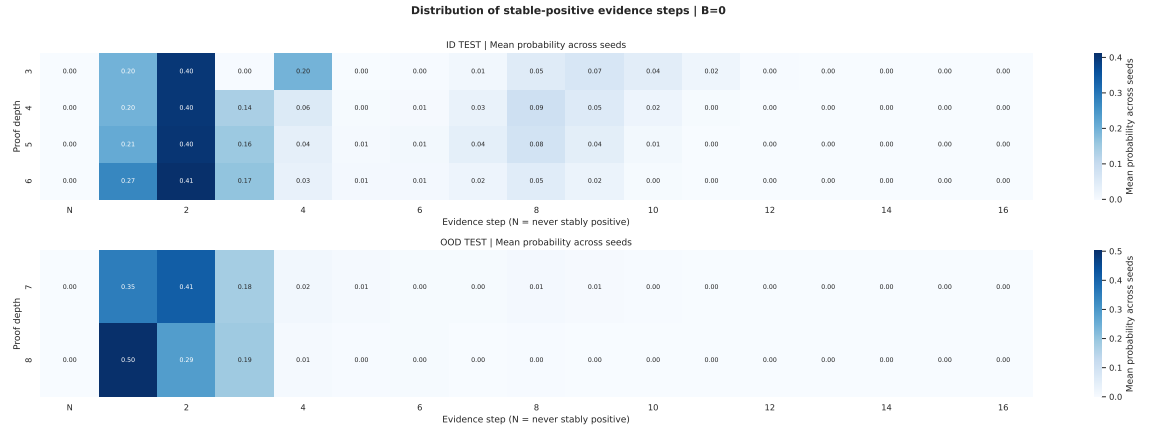


Figure 5.9: The distribution of the positions at which the prediction logits for positive examples exceed 0 and remain above 0.

Figure 5.9 shows no apparent correlation between the step at which ST finds evidence for the positive label and the problem depth. This provides important information about how ST might be solving the problem as this problem can be solved with a simple greedy search in which one can just select the edge containing an item that belongs to the cover. At the end, if the collected items match the required items, the label is 1; otherwise, it is 0. Also, the number of search steps required here will be equal to the problem depth, and hence the solution position will correlate with the search steps. Since we see no evidence of this, we can infer that ST is using a different method to identify the existence of an exact cover.

### Convergent Profile

The overall performance of ST is quite poor on the Convergent profile. Here, we look in detail at why this is the case. We would like to identify whether it is a capability problem or something else.

Figure 5.10 (a) shows the per-depth OOD performance for each seed. In Table 5.3, the average accuracy difference between ST and UT on the OOD test set is around 10 percentage points, with ST at the lower end. Figure 5.10 shows the seed variability in the accuracy of ST, especially

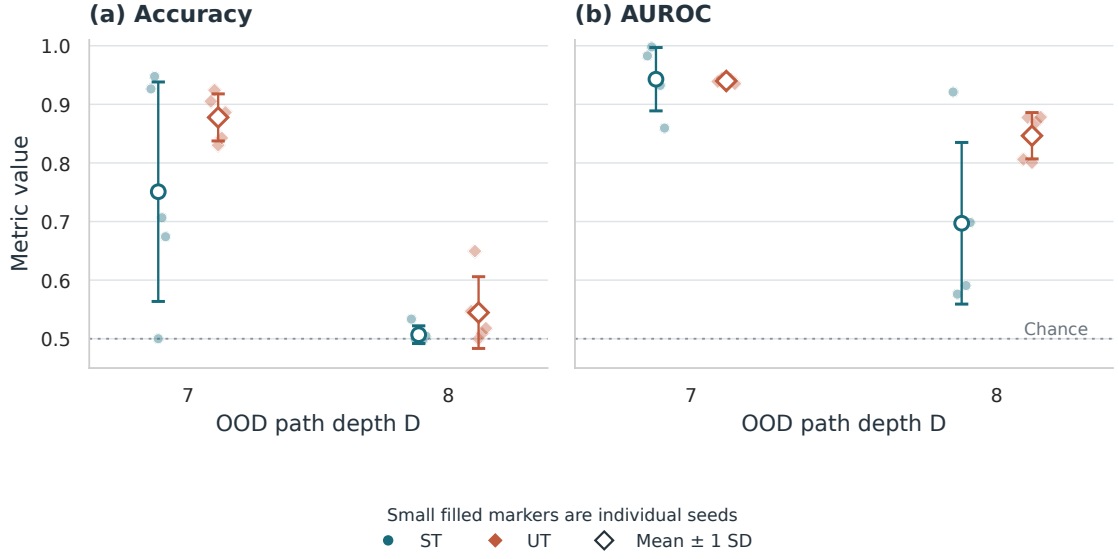


Figure 5.10: Per-seed accuracy and AUROC performance of ST and UT on the Path-EC Convergent profile.

for depth-7 test examples. UT’s performance is more stable across multiple seeds, and hence its average accuracy is higher.

Figure 5.10 (b) shows the depth-wise AUROC comparison. Although ST still shows high AUROC variability at both depths, it also shows higher values. So, ST runs have greater variation while having the capability to learn better class separation than UT in some cases.

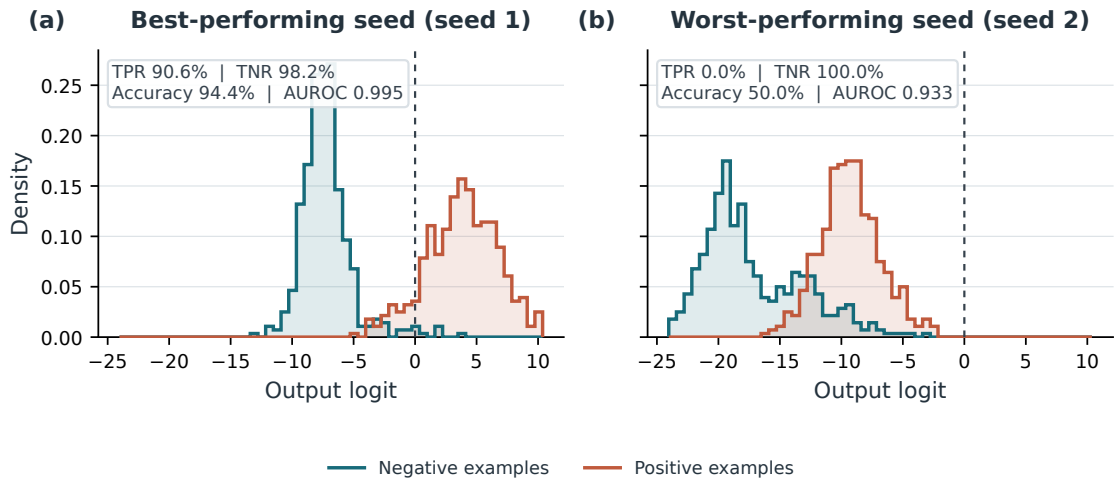


Figure 5.11: Class-conditional logit distributions for the best- and worst-performing ST checkpoints on depth-7 Path-EC Convergent examples.

We take the ST run with 0.5 accuracy in Figure 5.10 (a) and plot its logit distribution for depth 7 in Figure 5.11. The figure shows that the worst-performing ST still retains the ability to separate

the positive and negative examples, though the logit distribution has shifted toward negative values, resulting in low accuracy. It also has an AUROC of 0.933, indicating that the issue is threshold calibration rather than a lack of discrimination ability. We perform a sweep over

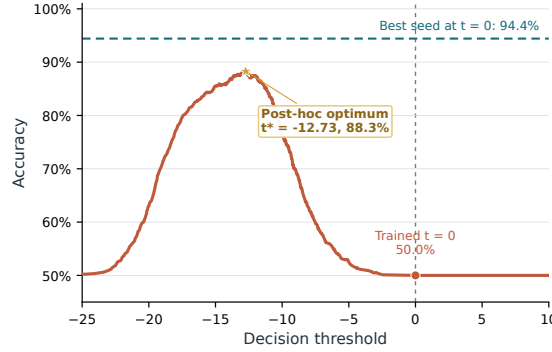


Figure 5.12: Performance after threshold calibration for the weakest ST seed at depth 7. At the trained threshold  $t = 0$ , accuracy is 50.0%. Selecting a threshold  $t^* = -12.73$  on the OOD validation set increases accuracy to 88.3%, with a TPR of 92.9% and a TNR of 83.7%. The best seed achieves 94.4% accuracy at  $t = 0$ .

the threshold values using the OOD validation set and choose the threshold with the highest validation accuracy. Figure 5.12 shows the test accuracy obtained by changing the threshold and indicates that we can recover accuracy to a level close to the best-performing seed after threshold calibration.

We provide further analysis of the OOD performance of ST and UT and a detailed comparison in Appendix A.3.4.

### State-Alias Profile

The performance of ST is better than UT on the ID test set but worse on the OOD test set of the State-Alias profile. Here, we analyze why.

Figure 5.13 shows that both ST and UT achieve top accuracy and AUROC for depth 7 problems, while ST shows higher variability across seeds. This accuracy variability is greater for depth 8 problems, where even with a high AUROC, the accuracy is lower for some seeds. This suggests that both ST and UT have similar discrimination ability, while ST has worse threshold transfer to higher depth problems, resulting in lower accuracy for some seeds.

We provide a more detailed analysis of the logit distribution for each seed and the thresholding issue in Appendix A.3.5.

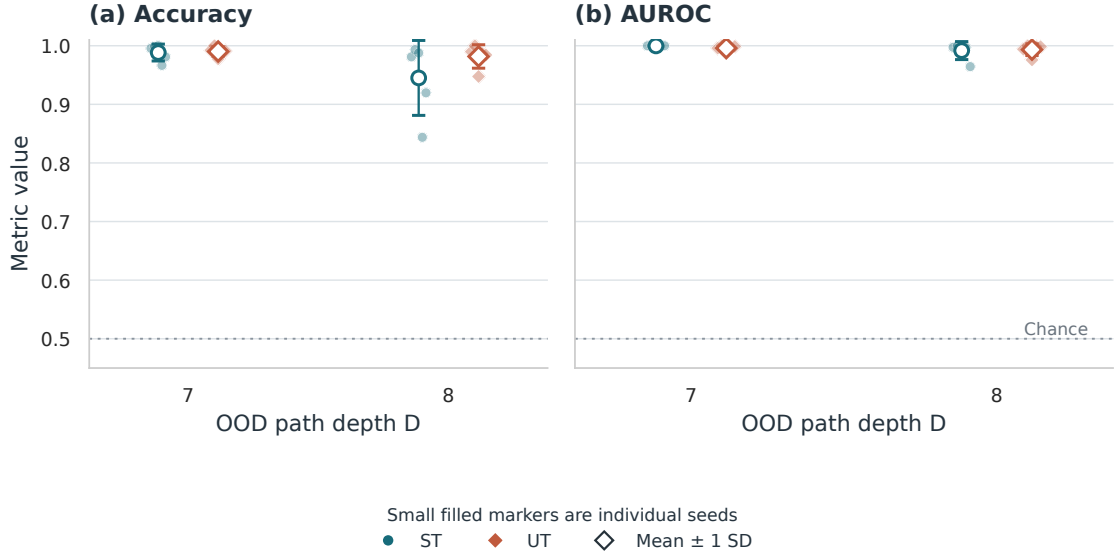


Figure 5.13: Comparison of accuracy and AUROC for ST and UT on the OOD State-Alias dataset.

### 5.3 Effect of Inference-Time Computation

Search Transformer, with its iterative computation, provides an opportunity for scaling computation at inference time. We study inference-time scalability under two different frameworks: performance under a fixed computation budget, where all examples receive a fixed search budget, and under adaptive computation, where the model learns to allocate a search budget to each problem instance.

#### 5.3.1 Fixed-Step Search Budget

We take the standard ST trained for a fixed budget of 16 steps on LD Hard and the Disjoint profile and evaluate it with search budgets ranging from 1 to 64 steps to study the effect on model performance.

**Result on LD Hard dataset.** Figure 5.14 (a) shows that increasing the inference-time search budget from 16 steps to 24 steps improves OOD accuracy consistently across all five seeds by  $5.0 \pm 1.36$  percentage points (mean  $\pm$  standard deviation across seeds). Increasing the search budget beyond 24 steps shows no noticeable improvement. From 16 to 24 steps, TPR rises by 16.0 pp, while TNR falls by 6.0 pp. Thus, the additional search shifts the prediction logit toward the positive class. However, AUROC also improves by 3.79 pp, showing that the gain is not only due to a threshold shift but also to improved class separation.

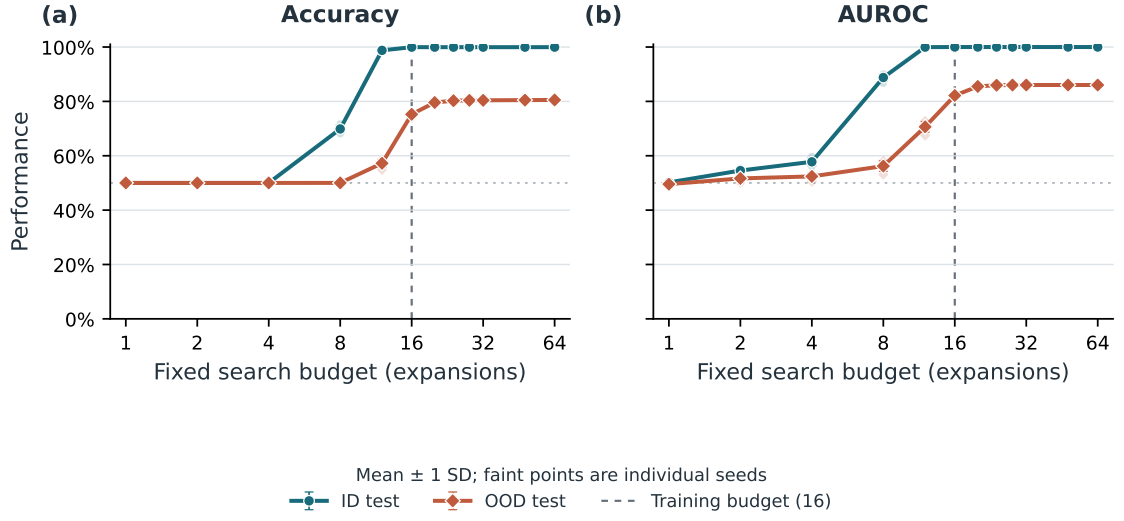


Figure 5.14: ST performance for different search steps: (a) compares accuracy for ID and OOD, and (b) shows the AUROC result.

**Result on Disjoint profile.** The performance on both the ID and OOD sets of the Disjoint profile is around 0.9. Here, we study the effect of scaling the inference-time computation on this dataset.

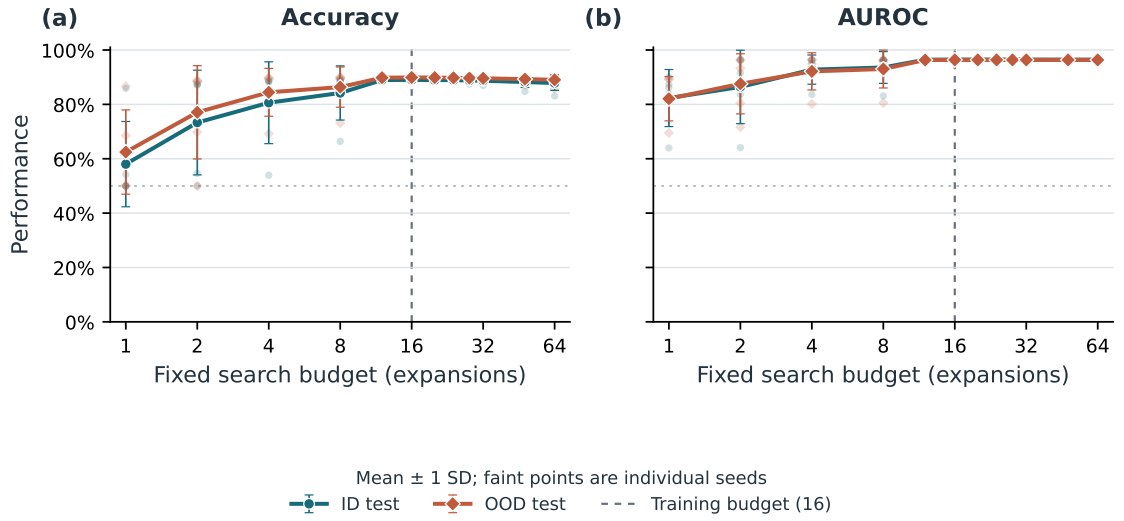


Figure 5.15: Performance on the Disjoint profile across different inference-time computation budgets.

Figure 5.15 (a) shows that the ID and OOD accuracies reach a maximum of 0.89 with a 16-step search. We then see a slight decrease in ID performance, which reaches 0.87 at 64 steps, while the OOD performance is more stable at larger search budgets. Unlike the LD Hard result, the scaling shows no sign of improvement with search budgets longer than the training budget. The Disjoint profile also did not show a correlation between evidence position and problem

depth. We hypothesize that since ST is not using iterative computation corresponding to the size of the graph, longer search does not provide any useful advantage.

Additional results for search step scaling are provided in Appendix A.3.2.

### 5.3.2 Adaptive Computation

Adaptive-computation methods generally show better extrapolation ability in the literature [3, 7]. We compare ST-Ponder with fixed-step ST on the LD Hard dataset to study the effect of adaptive computation. We focus our evaluation on the LD Hard dataset since fixed-step ST demonstrated a clear correlation between proof depth and evidence position; in contrast, the Disjoint profile exhibited diffused evidence positions, making the halting less applicable. We train ST-Ponder with a fixed budget of 16 iterations. During inference, however, we allow at most 32 iterations where adaptive computation can result in earlier halting. ST-Ponder halts its search process when its cumulative halting probability exceeds 0.99.

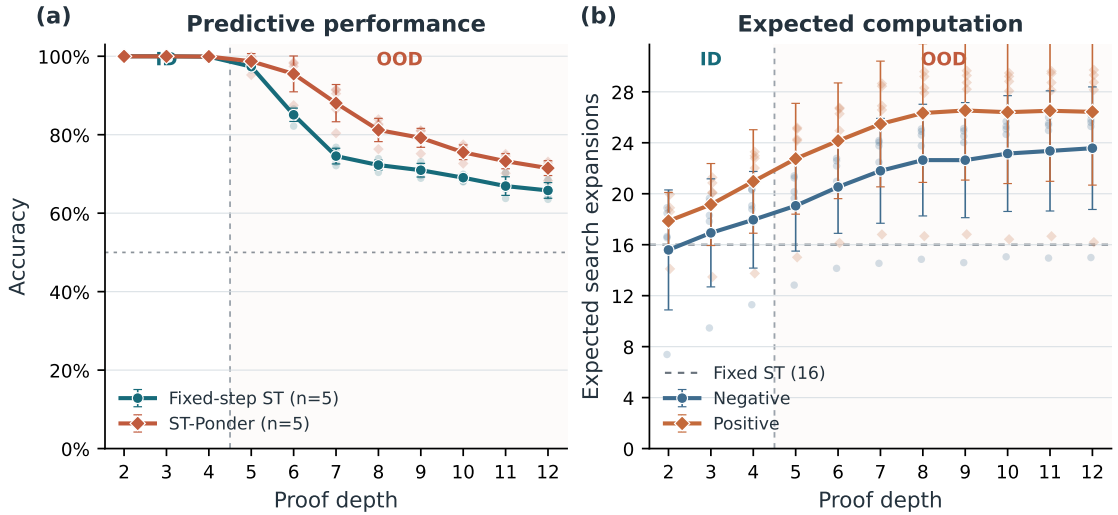


Figure 5.16: Comparison of fixed-step ST and ST-Ponder on LD Hard. Curves show the mean and one standard deviation across five seeds, with faint markers denoting individual seeds. (a) compares the accuracy of ST and ST-Ponder, and (b) shows the expected number of steps taken for each depth.

Figure 5.16 (a) shows comparable ID performance, while ST-Ponder clearly performs better on OOD examples, especially up to depth 7. Figure 5.16 (b) shows a clear upward trend as we increase the problem depth, suggesting that dynamic halting has learned to allocate more compute to problems requiring multiple steps. Although the variance among seeds is high, the average trend across them is upward. Beyond depth 8, we see that the model does not allocate more compute, and hence the performance also drops for larger depths. The performance difference for depth beyond 8 remains, but it mainly comes from the additional computation beyond the 16 steps used by ST-Ponder.

## 5.4 Ablation Studies

This section presents ablation studies conducted to assess the importance of various ST components. We use LD Hard as the primary testbed. For some components, we additionally use other datasets when the LD Hard results do not provide sufficient insights into their role. We explain these choices in the corresponding experiments.

### 5.4.1 Node Selection and Action Representation

We ablate two design choices of ST: the node-selection policy and the action representation. For node selection, we compare the default “Learned” policy against a deterministic “Newest node” policy that always selects the most recently generated node. For action representation, the standard ST uses a pointer into problem memory. We compare this with two alternatives: “Latent action” and “Node only”. The latent-action variant is based on the unconstrained action representation proposed by Núñez-Molina et al. [22]. In our implementation, an action-head MLP predicts this vector from the contextualized node representation produced by the selection module. The node-only variant removes the separate action prediction and instead uses the node representation as the input to the transition module, which predicts a successor using cross-attention to problem encodings. Together, the six configurations test whether learned node selection is useful, whether a separate action representation is needed, and whether grounding that action in problem memory provides a further benefit. We train all six configurations on the LD Hard dataset.

Table 5.4: Results of the combined node-selection and action/transition-design ablation. Performance entries report the mean and standard deviation across five seeds. For runs that did not reach the final curriculum rung, performance is taken from the final available checkpoint. *Comp.* gives the number of runs that completed the curriculum out of five.

| Node policy | Action variant | In-distribution   |                   |                   | Out-of-distribution |                   |                   | Comp. |
|-------------|----------------|-------------------|-------------------|-------------------|---------------------|-------------------|-------------------|-------|
|             |                | Acc.              | TPR               | TNR               | Acc.                | TPR               | TNR               |       |
| Learned     | Memory pointer | 1.000 $\pm$ 0.000 | 1.000 $\pm$ 0.000 | 0.999 $\pm$ 0.000 | 0.752 $\pm$ 0.012   | 0.799 $\pm$ 0.049 | 0.706 $\pm$ 0.025 | 5/5   |
| Learned     | Latent action  | 0.503 $\pm$ 0.004 | 0.558 $\pm$ 0.075 | 0.448 $\pm$ 0.071 | 0.502 $\pm$ 0.003   | 0.567 $\pm$ 0.078 | 0.437 $\pm$ 0.078 | 0/5   |
| Learned     | Node only      | 0.508 $\pm$ 0.007 | 0.434 $\pm$ 0.149 | 0.582 $\pm$ 0.147 | 0.501 $\pm$ 0.07    | 0.434 $\pm$ 0.161 | 0.567 $\pm$ 0.153 | 0/5   |
| Newest node | Memory pointer | 0.999 $\pm$ 0.000 | 1.000 $\pm$ 0.000 | 0.999 $\pm$ 0.00  | 0.737 $\pm$ 0.017   | 0.767 $\pm$ 0.066 | 0.707 $\pm$ 0.033 | 5/5   |
| Newest node | Latent action  | 0.505 $\pm$ 0.004 | 0.638 $\pm$ 0.120 | 0.372 $\pm$ 0.119 | 0.501 $\pm$ 0.004   | 0.643 $\pm$ 0.136 | 0.360 $\pm$ 0.137 | 0/5   |
| Newest node | Node only      | 0.543 $\pm$ 0.006 | 0.528 $\pm$ 0.052 | 0.558 $\pm$ 0.053 | 0.532 $\pm$ 0.003   | 0.677 $\pm$ 0.052 | 0.387 $\pm$ 0.055 | 0/5   |

Only the two configurations using the memory pointer complete the LD Hard curriculum, with all five runs completing under both selection policies. Neither the latent-action nor the node-only configurations complete it, and their accuracies remain close to random chance. This suggests that grounding the action in problem memory is crucial for learning LD Hard under our training setup. Among the memory-pointer configurations, the two selection policies achieve nearly identical ID performance. Learned selection improves average OOD accuracy from 73.7% to 75.2%, a difference of 1.5 percentage points. Thus, LD Hard provides substantial evidence for the memory pointer, but only limited evidence for learned node selection.

**Ablating the selection policy on the State-Alias profile.** The preceding LD Hard result shows that learned selection provides only a small improvement on the OOD test set. Logical deduction can be solved through forward chaining; therefore, it may be sufficient to expand only the newest node. So, the State-Alias profile offers a stronger test because reaching the same graph node via different paths may represent different states, which can make selecting the earlier nodes potentially useful.

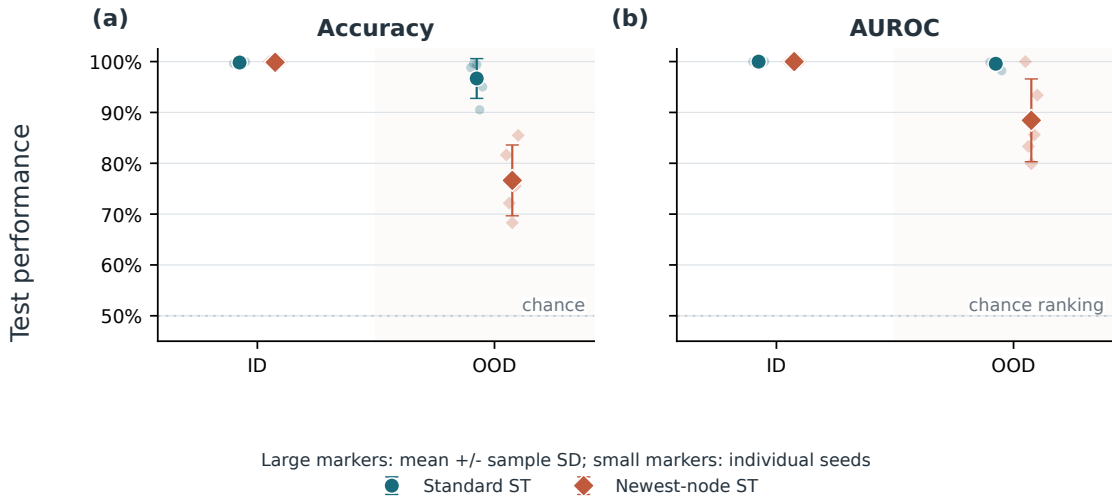


Figure 5.17: Accuracy and AUROC comparison of the standard learned node-selection policy and the newest-node policy on the Path-EC State-Alias profile.

Figure 5.17 shows similar ID performance, but the newest-node ST exhibits lower average OOD accuracy and AUROC. Together with the LD Hard result, this indicates that the benefit of learned selection is task-dependent. It offers only a minor improvement when a newest-node chain is sufficient, but a larger benefit on State-Alias, where access to earlier latent nodes for selection may be useful.

A more detailed depth-wise comparison and per-seed results are provided in Appendix A.3.6.

**Learned action vector vs no separate action.** Because both the latent-action and the node-only variants failed to complete training on LD Hard, those results do not provide a meaningful

comparison for the two action representations. We therefore repeat the comparison on LD Medium, where both variants complete training, to determine whether a learned latent action offers any benefit after successful learning.

Table 5.5: Ablation results comparing the latent-action and no-action variants on LD Medium. The reference row reports the standard ST configuration over five seeds, while the ablation rows report the mean and standard deviation over three seeds. *Comp.* gives the number of completed runs.

| Node policy                           | Action variant         | In-distribution |               |               | Out-of-distribution |               |               | Comp. |
|---------------------------------------|------------------------|-----------------|---------------|---------------|---------------------|---------------|---------------|-------|
|                                       |                        | Acc.            | TPR           | TNR           | Acc.                | TPR           | TNR           |       |
| Reference configuration (standard ST) |                        |                 |               |               |                     |               |               |       |
| Learned                               | Memory pointer         | 0.999 ± 0.000   | 1.000 ± 0.000 | 0.999 ± 0.000 | 0.900 ± 0.008       | 0.864 ± 0.043 | 0.937 ± 0.030 | 5/5   |
| Ablation variants                     |                        |                 |               |               |                     |               |               |       |
| Learned                               | Latent action          | 0.985 ± 0.002   | 0.993 ± 0.001 | 0.976 ± 0.005 | 0.845 ± 0.002       | 0.807 ± 0.014 | 0.884 ± 0.012 | 3/3   |
| Learned                               | Node query (no action) | 0.997 ± 0.001   | 0.997 ± 0.001 | 0.996 ± 0.002 | 0.840 ± 0.008       | 0.773 ± 0.034 | 0.907 ± 0.019 | 3/3   |
| Newest node                           | Latent action          | 0.993 ± 0.001   | 0.996 ± 0.001 | 0.989 ± 0.002 | 0.846 ± 0.005       | 0.790 ± 0.004 | 0.903 ± 0.006 | 3/3   |
| Newest node                           | Node query (no action) | 0.996 ± 0.001   | 0.997 ± 0.002 | 0.994 ± 0.001 | 0.837 ± 0.016       | 0.791 ± 0.056 | 0.883 ± 0.024 | 3/3   |

All latent-action and node-only variants complete training on LD Medium. However, their OOD performance remains similar, showing no clear advantage from learning a separate latent-action vector. Both variants exhibit lower performance than the memory-pointer reference, suggesting that grounding the action in problem elements is more advantageous than adding an unconstrained action representation under our training settings.

**Change in Model Parameters.** The variant with *Learned* selection and *Latent* action needs the selection module to predict the selection logit and the action vector. The selection module without the final prediction heads has approximately 395K total parameters. The selection head has around 100K and the action head has around 115K learnable parameters. The node-only variant passes the search node directly to the transition module, removing the entire selection module under the newest-node policy but only the action head under learned selection. The learnable parameters of all ST variants trained in Table 5.4 can be computed by adding or removing the respective components.

Together, these ablations demonstrate that the memory pointer is the most reliably beneficial action representation considered here. Learned selection offers a marginal improvement on LD Hard but demonstrates a clear advantage on State-Alias, suggesting that its usefulness depends on the task being solved.

### 5.4.2 Readout Paradigm

We use the IS paradigm (Section 2.3.1) for the main experiments. This choice is motivated by the intuition that each node can individually represent the set of proved atoms and hence should have sufficient information for query matching. Furthermore, because we use the *LogMeanExp* pooling operation, the information regarding a proved atom can exist in any search node. Here, we study the effect that the choice of readout paradigm has on our final results. We use the LD Hard dataset for this study. We repeat each experiment with three seeds, and report their mean and standard deviation. We only vary the readout function while keeping everything else fixed for all experiments.

Table 5.6: Readout paradigm ablation on LD Hard. Each entry reports the mean and standard deviation over three training seeds.

| Readout        | In-distribution  |                 |                  | Out-of-distribution |                   |                   |
|----------------|------------------|-----------------|------------------|---------------------|-------------------|-------------------|
|                | Acc.             | TPR             | TNR              | Acc.                | TPR               | TNR               |
| IS: LogMeanExp | 1.00 $\pm$ 0.00  | 1.00 $\pm$ 0.00 | 0.999 $\pm$ 0.00 | 0.752 $\pm$ 0.012   | 0.799 $\pm$ 0.049 | 0.706 $\pm$ 0.025 |
| ES: Mean       | 1.00 $\pm$ 0.00  | 1.00 $\pm$ 0.00 | 0.999 $\pm$ 0.00 | 0.773 $\pm$ 0.000   | 0.968 $\pm$ 0.019 | 0.578 $\pm$ 0.019 |
| ES: Max        | 1.000 $\pm$ 0.00 | 1.00 $\pm$ 0.00 | 1.00 $\pm$ 0.00  | 0.776 $\pm$ 0.004   | 0.997 $\pm$ 0.002 | 0.556 $\pm$ 0.006 |

*IS: LogMeanExp* is an instance-space paradigm in which *LogMeanExp* is used for pooling the prediction logits of individual nodes. *ES: Mean* and *ES: Max* are the mean and max pooling operations performed using the node embeddings, which are then used with the matching function to produce the prediction logit.

Table 5.6 shows similar in-distribution performance across all three readout variants. The *Mean* and *Max* variants have better out-of-distribution accuracy. However, both of these variants appear to be biased toward positive predictions, with high TPR and low TNR. Our standard approach shows more balanced performance across the positive and negative labels. Hence, for logical deduction, the use of different MIL learning paradigms does not seem to have an effect, while we believe the instance-space paradigm provides a more interpretable and intuitive readout for ST.

## 6 Conclusion

In this thesis, we investigated whether structuring the latent recurrent computation in a search-like manner offers a useful inductive bias for learning and extrapolation on structured decision problems. Building on the conceptual ST proposal of Núñez-Molina et al. [22], we developed a concrete, trainable variant and compared it against a similarly sized UT on logical deduction and Path-EC decision problems.

### 6.1 Summary

Across the Easy, Medium, and Hard logical deduction datasets, ST achieves average out-of-distribution accuracy 5.7, 11.6, and 11.0 percentage points higher than UT, respectively. On the LD Hard, ST reaches a sustained 90% validation accuracy after seeing a median of 3.84 million training examples, compared with 53.76 million for UT, meaning that UT requires approximately 14 times as many examples under this criterion. Under query substitution, ST maintains a balanced accuracy of  $96.6 \pm 0.8\%$ , in contrast to  $77.6 \pm 8.1\%$  for UT, which also varies more across seeds. However, this advantage does not extend to the Path-EC problems. Both models achieve comparable ID accuracy on the Convergent and State-Alias profiles, but the average OOD accuracy of ST is lower by 8.2 and 1.9 percentage points, respectively. Several ST runs nevertheless retain a high AUROC, suggesting that part of these performance gaps can be explained by poor threshold transfer. The Disjoint profile presents a different trade-off: all five ST runs reach the final curriculum rung and plateau at approximately 90% accuracy, whereas only one of five UT runs reaches the final rung. That run nevertheless exceeds the ID and OOD accuracy of ST by 9.9 and 5.3 percentage points, respectively.

The recurrent nature of ST allows us to vary its computation budget during inference. We study this by varying the number of search steps for a fixed-step ST trained with 16 steps and by using ST-Ponder with learned dynamic halting. On the LD Hard, increasing the fixed budget from 16 to 24 steps improves OOD accuracy by  $5.0 \pm 1.36$  percentage points, although increasing it further does not show noticeable improvement. In contrast, increasing the budget beyond 16 steps does not improve performance on the Disjoint profile. ST-Ponder also achieves higher OOD accuracy than the fixed 16-step model on LD Hard and allocates progressively more steps as problem depth increases until approximately depth 7, after which the allocation saturates. Together, these results show that additional search steps are beneficial only when the learned computation generates useful nodes when running the search for longer.

We ablate the node-selection policy, action/transition configuration, and readout mechanism to identify which components contributed to the performance of ST. Compared with the newest-node policy, learned selection provides a marginal improvement in OOD accuracy on LD Hard but yields substantially larger gains in accuracy and AUROC on the State-Alias profile. This suggests that its benefit depends on the structure of the task, particularly on whether access to prior latent nodes is useful for the task. Among the action/transition configurations tested on LD Hard, only the memory pointer variants complete the curriculum, indicating that the standard memory-pointer configuration is important for reliable learning in our training setup. On LD Medium, the latent-action and node-only variants perform similarly, providing no evidence that a separate unconstrained action vector improves performance in this setting. Finally, all readout variants achieve comparable ID performance, but the mean and max embedded-space readouts are strongly biased toward positive OOD predictions. Thus, neither of the two provides a uniformly better alternative to the standard instance-space readout.

Overall, the results show that ST is a viable latent recurrent architecture; however, its advantage over UT depends on the specific task. The advantages observed on the logical deduction tasks suggest that search-structured latent recurrence can facilitate learning and extrapolation, whereas the Path-EC results show that this same organization does not necessarily yield stronger solutions across various problems.

## 6.2 Limitations and Future Work

In this thesis, we represent a problem as a set of context vectors and a single query vector, and we restrict the output to a binary decision. This formulation allowed us to isolate the study of search-structured latent computation from input parsing and output generation, while limiting our conclusions to problems whose relevant elements are already explicitly encoded. As shown in Section A.4, an ST variant operating directly on token inputs is difficult to train using decision labels alone, although intermediate-state supervision improves its learnability. Future work could therefore combine ST with a learned input encoder and investigate whether token-based training can be made reliable.

While we define the computation performed by ST as search, the current implementation is more of a “soft” approximation of a discrete search process. At each step, all candidate parents are expanded, assigned soft selection weights, and aggregated to create the new node. The resulting node may therefore combine information from several candidate successors rather than being a single discrete transition. Future work could explore harder node selection through temperature annealing or a straight-through Gumbel–Softmax estimator. Currently, we employ a manual action masking rule that forces the model to use a different action when revisiting a node. Future work could instead condition the action selection on the actions previously used for that node, allowing this retry behavior to be learned.

# A Appendix

## A.1 Reasoning Problems

Here, we describe the benchmark datasets in detail, the configurations used to generate them, and the approach used for encoding the problems.

### A.1.1 Logical Deduction

Logical deduction consists of a set of facts, a set of rules, and a query. They are generated using three configurations that share the same generation procedure but differ in vocabulary size, number of facts and rules, and the range used for out-of-distribution evaluation. Table A.1 compares these configurations, and Table A.2 gives the corresponding split sizes.

Table A.1: Logical-deduction configurations used in the experiments.

| Parameter            | LD Easy | LD Medium | LD Hard |
|----------------------|---------|-----------|---------|
| Atom vocabulary size | 24      | 32        | 64      |
| Facts per example    | 6       | 10        | 18–24   |
| Rules per example    | 8       | 16        | 28–36   |
| ID depths            | 2–4     | 2–4       | 2–4     |
| OOD depths           | 5       | 5–8       | 5–12    |

All three configurations allow one or two atoms in a rule body. A proof rule is unary (containing a single atom in the body) with probability 0.20, whereas a distractor rule is unary with probability 0.25. For a binary distractor (a rule body with a conjunction of two atoms), the probability of constructing a hard-unreachable rule is 0.55. A hard-unreachable rule contains one provable body atom and one atom that is guaranteed to be unprovable, so that the rule is partially satisfied but cannot derive its head.

Table A.2: Logical-deduction split sizes. The datasets contain the same number of examples for each depth and label. The OOD totals differ because the three configurations cover different numbers of OOD depths.

| Split          | Per depth<br>and label | LD Easy | LD Medium | LD Hard |
|----------------|------------------------|---------|-----------|---------|
| Training       | 4096                   | 24 576  | 24 576    | 24 576  |
| ID validation  | 1280                   | 7680    | 7680      | 7680    |
| ID test        | 1280                   | 7680    | 7680      | 7680    |
| OOD validation | 640                    | 1280    | 5120      | 10 240  |
| OOD test       | 640                    | 1280    | 5120      | 10 240  |

## Problem Generation

The three configurations contain independently generated positive and negative examples; the examples are not grouped into positive–negative families. The negative-generation procedure is described below.

For generation, we fix an atom vocabulary, which is a list of atom strings. The atom strings do not contain any semantic information, so they could also be generated programmatically, for example, as  $a_1, \dots, a_v$ , where  $v$  is the vocabulary size, without affecting the model’s performance. An example consists of a set of facts and a set of rules. A fact is a randomly chosen atom from the vocabulary. A rule consists of the rule body and its head. We restrict the rule body to contain either one or two atoms, with the number chosen randomly during dataset generation. The rule head contains a single atom.

Data generation takes a “depth” parameter, which determines the length of the proof chain from one of the known facts to the query atom. Data generation proceeds by creating a sequential chain of atoms from one fact to the query, both of which are chosen randomly. From the chain, a rule is generated by using one atom from the chain and another atom randomly chosen from remaining atom set. Since second atom choice is randomly determined, some of the rules in the chain can have a single antecedent.

After generating the required starting fact and subsequent rules to prove the query, additional rules and facts are added as distractors to the example. This generates the positive example. We obtain a negative example by replacing one of the required facts with a randomly chosen one. We generate both positive and negative examples independently, so when the negative is obtained by replacing the fact, the original positive example does not get added to the example set. This is different from the approach we take for Path-EC, where a positive example and its counterfactual negative are generated together and added to the same dataset split.

## Input Encoding

The logical deduction problem consists of facts, rules, and a query. We encode each of them as a rule-like object. A fact has an empty body and stores its atom in the head field, whereas a query additionally activates a query flag.

Let  $A \rightarrow c$  be a rule where  $A \subseteq \mathcal{V}$  and  $1 \leq |A| \leq 2$ . All atoms come from a fixed-size ordered vocabulary  $\mathcal{V}$ . We encode each object using two vocabulary-sized vectors, one for the body and one for the head, followed by a binary query flag.

*Rule Body* is encoded as a multi-hot vector where the position  $i$  is set to 1 if the  $i^{\text{th}}$  atom is in the body:  $a_i \in A$ . *Rule Head* is encoded as a one-hot vector where the position of the head element is set to 1.

We concatenate the rule-body vector, rule-head vector, and query flag to obtain a feature vector of dimension  $2N_v + 1$ . For facts and queries, the rule body is a zero vector and the rule head contains the one-hot vector of the corresponding atom. The query flag is one only for the query object.

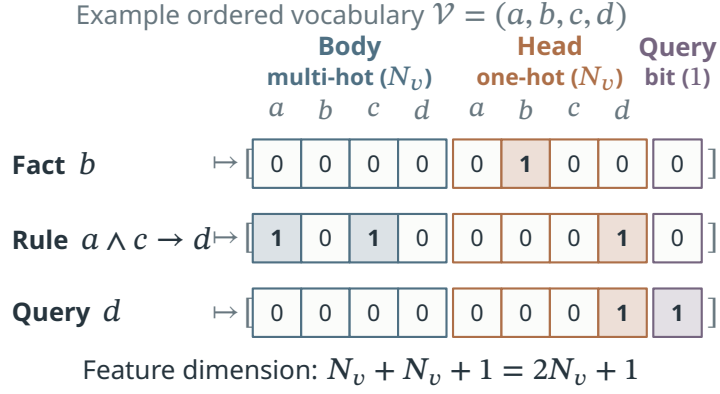


Figure A.1: Feature encoding of logical-deduction objects for the ordered example vocabulary  $\mathcal{V} = (a, b, c, d)$ . The body block is multi-hot, the head block is one-hot, and the final scalar marks the query object. Facts and queries have empty bodies. For a vocabulary of size  $N_v$ , each feature vector has dimension  $2N_v + 1$ .

### Example Problems

The following examples show one positive and one negative LD Hard problem at generation depth 2 in the human-readable representation produced by the generator. These are two independently generated examples rather than two members of the same family. The model receives the vector encoding described above. After each example, we show only the rules relevant for determining its label.

#### Positive example.

Facts: busy, tender, proud, blushing, witty, fearless, tense, innocent, crowded, foolish, worried, horrible, mysterious, amused, disobedient, bright, wrong, alert, helpful, sincere, frightened, mean, zealous

Rules:

```
condemned -> weary
friendly -> naughty
worried and disobedient -> strange
busy and naughty -> comfortable
victorious -> talented
good and rude -> glamorous
witty and foolish -> pessimistic
diplomatic and crowded -> troubled
worried -> frantic
versatile and frightened -> perfect
lucky and fearless -> curious
loving and disobedient -> diplomatic
perfect and rude -> fancy
```

```
attentive and frightened -> shy
curious and wrong -> elated
friendly and hypocritical -> muddy
mysterious and tender -> victorious
alert and hypocritical -> aggressive
strange and alert -> condemned
foolish and crowded -> lucky
bright and alert -> cruel
wrong -> lonely
alert -> joyous
bright -> bad-tempered
blushing and foolish -> friendly
stormy and rude -> modern
blushing and lucky -> easy
alert and bright -> hurt
sincere -> elegant
helpful and powerful -> fine
```

Query: easy Label: 1 Generation depth: 2

Here, foolish, crowded, and blushing are facts. The query is therefore derived using two rule applications:

$$\begin{aligned}\text{foolish} \wedge \text{crowded} &\longrightarrow \text{lucky}, \\ \text{blushing} \wedge \text{lucky} &\longrightarrow \text{easy}.\end{aligned}$$

**Negative example.**

Facts: disobedient, crowded, lonely, easy, tense, condemned, loving, frightened, victorious, naughty, mysterious, aggressive, curious, cruel, amused, muddy, shy, modern, busy, fancy, horrible, hilarious, bright

Rules:

glamorous and frantic -> weary  
innocent and hilarious -> elated  
strange -> zealous  
foolish and curious -> pessimistic  
fearless and frantic -> tender  
bright -> bad-tempered  
talented -> jittery  
friendly and easy -> elegant  
mysterious -> helpful  
bad-tempered and cruel -> attentive  
bright and friendly -> mean  
strange and busy -> foolish  
worried and powerful -> sincere  
aggressive and horrible -> proud  
cruel -> quaint  
modern -> glamorous  
rude and hilarious -> wrong  
shy and innocent -> fearless  
witty and hurt -> lucky  
modern -> good  
foolish and naughty -> comfortable  
comfortable and proud -> stormy  
strange and lonely -> troubled  
alert and attentive -> witty  
loving and powerful -> hypocritical  
diplomatic and mysterious -> hurt  
joyous and rude -> blushing  
lonely -> fine  
innocent -> talented  
condemned and versatile -> perfect

Query: pessimistic Label: 0 Generation depth: 2

The query rule requires both foolish and curious. Although curious and busy are facts, strange is neither a fact nor the head of another rule. The two relevant rule applications are therefore blocked in sequence:

strange unavailable  $\Rightarrow$  foolish not derived  $\Rightarrow$  pessimistic not derived.

**A.1.2 Path Exact-Cover Problem**

The goal of the Path-EC dataset is to test the graph-traversal capability of ST. The dataset follows a specific structure shown in Figure 2.3. The Path-EC benchmark contains three profiles that test different structural requirements. They are not intended to form an ordered difficulty hierarchy.

**Disjoint profile.** The Disjoint profile has three parameters that control the task structure. The fixed parameter  $K$  is the number of paths that lead from the source to the target vertex. The other two parameters,  $B$  (branching) and  $P$  (prefix length), are varied to obtain different conditions.

*Deceptive Branches ( $B$ )* denotes the number of deceptive branches that originate from both the source and target nodes and contain valid items in their paths up to the path length of  $P$ . This generation is symmetric, so if there are  $B$  deceptive paths, each with  $P$  valid prefix items, from source to target, the example has the same number of deceptive branches and the same prefix size from the target node. This prevents the model from using a shortcut by choosing whether to perform the graph search from the source or target depending on the location of the deceptive branch.

*Total Decoy Paths ( $K$ )* is fixed and denotes the number of decoy paths from the source to the target, in addition to the witness path. For a given value of  $B$ , there are  $B$  deceptive paths in each direction and therefore  $2B$  deceptive paths in total. The remaining  $K - 2B$  paths contain only items outside the target set and can be rejected immediately. In contrast, a deceptive path initially contains valid items, so the model may need to explore it before returning to an earlier search state.

**Convergent profile.** In the Disjoint profile, all  $K$  paths are disjoint, so there is no edge linking any two paths. The Convergent profile instead contains paths that can diverge and later converge within the graph. We keep the problem simple by setting  $P = 1$ , so that each internal deceptive branch has exactly one target-set item before it can return to the witness path.

**State-Alias profile.** This profile specifically tests “alias” states. An alias state is a state that can be reached from the source or target, where both paths contain valid items up to this point. After this, one of the paths is invalidated because it contains a repeated item between this node and the goal. The task therefore requires tracking not only the visited graph node but also the items covered so far.

### Problem Generation

The configurations used for the reported Path Exact-Cover datasets are shown in Table A.3. Table A.4 gives the corresponding split sizes.

The problem generation approach directly follows from the problem structure, where we randomly sample the edge labels (items) and vertex labels and construct a graph with specific properties, where the locations of deceptive paths, items, and alias paths are chosen randomly.

Families never cross dataset splits. The Disjoint profile uses master seed 2102, whereas the Convergent and State-Alias profiles use master seeds 2103 and 2104, respectively.

Table A.3: Configurations used to generate the Path Exact-Cover benchmark profiles. Here,  $D$  is the target-set size and the required path length,  $K$  is the number of decoy paths,  $B$  is the number of deceptive paths in each direction, and  $P$  is the length of a deceptive path’s valid segment.

| Profile           | Controlled structure                                                | ID depth | OOD depth | $K$ | $B$ | $P$               |
|-------------------|---------------------------------------------------------------------|----------|-----------|-----|-----|-------------------|
| Disjoint, $B = 0$ | Root-to-goal decoy paths without deceptive alternatives             | 3–6      | 7–8       | 4   | 0   | 0                 |
| Disjoint, $B = 1$ | Root-to-goal decoy paths with one deceptive path in each direction  | 3–6      | 7–8       | 4   | 1   | $0, \dots, D - 1$ |
| Disjoint, $B = 2$ | Root-to-goal decoy paths with two deceptive paths in each direction | 3–6      | 7–8       | 4   | 2   | $0, \dots, D - 1$ |
| Convergent        | Decoy paths that diverge and converge within the witness path       | 4–6      | 7–8       | 4   | 2   | 1                 |
| State-Alias       | Different collected-item sets reach the same graph node             | 3–6      | 7–8       | –   | –   | –                 |

Table A.4: Path-EC split sizes. Each family contains one positive example and its paired counterfactual negative, and therefore contributes two examples. The two per-depth columns apply to every profile; the profile columns report the total number of examples across the corresponding depth range. The three Disjoint conditions are combined when training the single Disjoint model.

| Split         | Families per depth | Examples per depth | Disjoint $B = 0$ | Disjoint $B = 1$ | Disjoint $B = 2$ | Convergent | State-Alias |
|---------------|--------------------|--------------------|------------------|------------------|------------------|------------|-------------|
| Training      | 600                | 1200               | 4800             | 4800             | 4800             | 3600       | 4800        |
| ID validation | 240                | 480                | 1920             | 1920             | 1920             | 1440       | 1920        |
| ID test       | 480                | 960                | 3840             | 3840             | 3840             | 2880       | 3840        |
| OOD test      | 448                | 896                | 1792             | 1792             | 1792             | 1792       | 1792        |

## Input Encoding

Here, we present the approach we took to translate the path exact-cover problem from the directed graph to a set of elements. We achieve this by encoding the graph as a set of edge objects followed by one query object. For a given maximum number of nodes  $N_n$  and maximum number of items  $N_i$ , we concatenate the current-or-goal, item, edge-source, edge-destination, and query-start vectors in this order. The four node vectors have size  $N_n$ , whereas the item vector has size  $N_i$ , giving a total feature dimension of  $4N_n + N_i$ .

For an edge  $u \xrightarrow{c} v$ , we activate item  $c$ , source node  $u$ , and destination node  $v$ ; the remaining blocks are zero. For a query with start node  $s$ , goal node  $t$ , and target item set  $C$ , we activate  $t$  in the current-or-goal block, every item in  $C$ , and  $s$  in the query-start block. The edge-source and edge-destination blocks are zero for the query. Thus, the item block represents the edge label for an edge object and the target item set for the query object. When representing a search state, the first two blocks instead store the current node and the items collected so far.

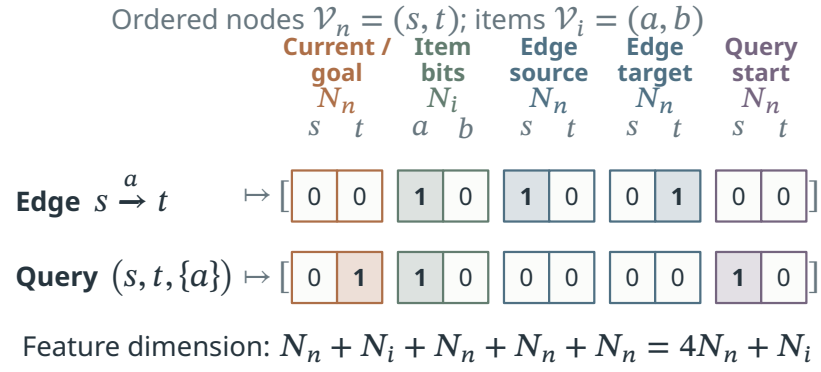


Figure A.2: Feature encoding of a Path-EC edge and query for the ordered node vocabulary  $\mathcal{V}_n = (s, t)$  and item vocabulary  $\mathcal{V}_i = (a, b)$ . The edge  $s \xrightarrow{a} t$  activates its item, source, and target positions. The query  $(s, t, \{a\})$  activates the goal, target-item, and query-start positions. Each feature vector has dimension  $4N_n + N_i$ .

### Example Problems

Figures A.3 and A.4 show one small generated family for each Path-EC profile. Each family contains a positive example and its solver-verified counterfactual negative. For readability, the generated item identifiers are renamed to letters; the graph structure and the counterfactual label exchange are kept unchanged.

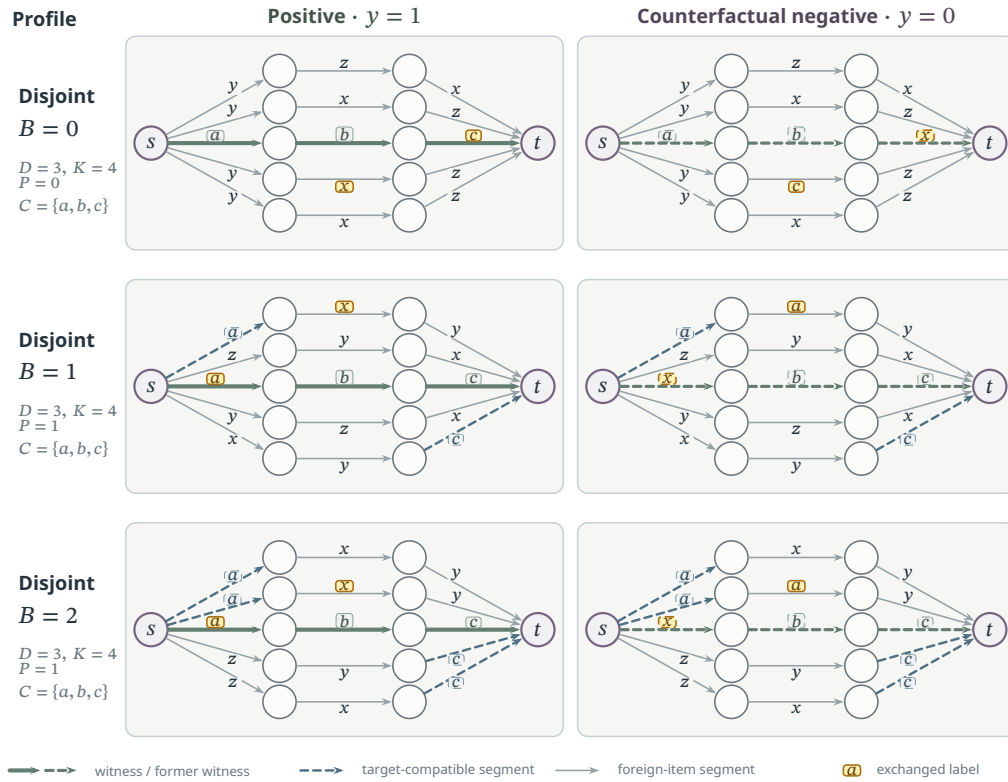


Figure A.3: Path-EC Disjoint example families at depth  $D = 3$ . In each row, the negative example is obtained by exchanging the two labels highlighted in yellow. This keeps the graph structure, query, and global item counts fixed but removes the exact-cover witness. Increasing  $B$  replaces immediately rejectable decoy paths with paths that begin or end with target items.

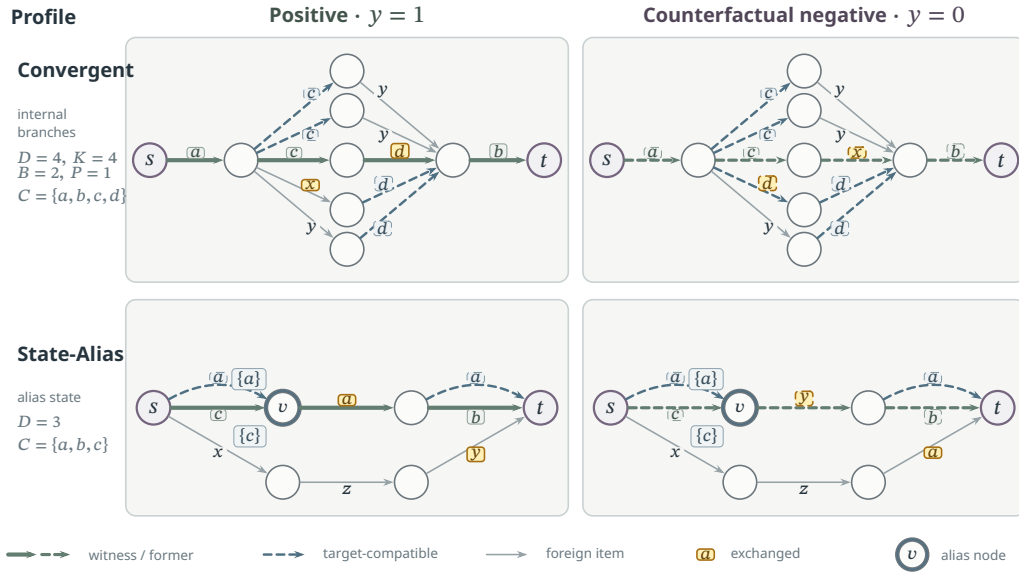


Figure A.4: The Convergent profile (top row) introduces alternative decoy paths that diverge from and later converge with the witness path. The State-Alias profile allows different valid collected-item sets to reach the same alias node  $v$ , shown with a double outline. Therefore, the current graph node alone is not sufficient to describe the search state; the state also requires the cover up to this point. In each row, the right panel is obtained by exchanging the two labels highlighted in yellow and is the solver-verified counterfactual negative of the example on its left.

## A.2 Supplementary Experimental Setup

### A.2.1 Model Configuration

Both ST and UT take a sequence of multi-hot feature vectors as input. The feature dimension depends on the dataset and its vocabulary size. For logical deduction, the feature dimension for an atom vocabulary of size  $N_v$  is  $2N_v + 1$ . For Path-EC, let  $N_i$  be the size of the item vocabulary and  $N_n$  the size of the node vocabulary. The feature dimension is then  $4N_n + N_i$ . For all models, we have a linear feature projection that maps a feature vector of any dimension to a predefined model dimension. The input dimension and the feature projection used by ST and UT are available in Table A.5.

Table A.5: Input feature dimensions and their projection into the latent model space. Both architectures use the same input features and a bias-free linear projection; only the output dimension of the projection differs.

| Dataset                                   | Input dimension | ST projection                                   | UT projection                                   |
|-------------------------------------------|-----------------|-------------------------------------------------|-------------------------------------------------|
| LD Easy ( $N_v = 24$ )                    | 49              | $\mathbb{R}^{49} \rightarrow \mathbb{R}^{192}$  | $\mathbb{R}^{49} \rightarrow \mathbb{R}^{384}$  |
| LD Medium ( $N_v = 32$ )                  | 65              | $\mathbb{R}^{65} \rightarrow \mathbb{R}^{192}$  | $\mathbb{R}^{65} \rightarrow \mathbb{R}^{384}$  |
| LD Hard ( $N_v = 64$ )                    | 129             | $\mathbb{R}^{129} \rightarrow \mathbb{R}^{192}$ | $\mathbb{R}^{129} \rightarrow \mathbb{R}^{384}$ |
| Path-EC profiles ( $N_n = 40, N_i = 16$ ) | 176             | $\mathbb{R}^{176} \rightarrow \mathbb{R}^{192}$ | $\mathbb{R}^{176} \rightarrow \mathbb{R}^{384}$ |

After the input projection, ST uses a latent dimension of 192 and an MLP hidden dimension of 512, whereas UT uses a latent dimension of 384 and an MLP hidden dimension of 1536. The fixed and adaptive variants of each architecture across all datasets retain these dimensions. Tables A.6 and A.7 summarize the main model parameters.

Table A.6: Main model parameters used for ST and ST-Ponder. Both variants use the same configuration, except for the additional halting head used by ST-Ponder.

| Parameter                               | Value                |
|-----------------------------------------|----------------------|
| Latent model dimension                  | 192                  |
| MLP hidden dimension                    | 512                  |
| Expansion-controller layers             | 1                    |
| Attention heads                         | 8                    |
| Key/value dimension per head            | 32/32                |
| Action-pointer heads                    | 8                    |
| Action-pointer key dimension            | 32                   |
| Transition and readout hidden dimension | 512                  |
| Activation function                     | SiLU                 |
| Dropout                                 | 0                    |
| Positional encoding                     | None                 |
| Ponder halting-head hidden dimension    | 512 (ST-Ponder only) |

### A.2.2 Implementation Environment

Table A.8 shows details of the environment used for implementation.

Table A.7: Model parameters used for UT.

| Parameter                    | Value |
|------------------------------|-------|
| Latent model dimension       | 384   |
| MLP hidden dimension         | 1536  |
| Shared Transformer layers    | 1     |
| Attention heads              | 8     |
| Key/value dimension per head | 48/48 |
| Activation function          | SiLU  |
| Dropout                      | 0     |
| Positional encoding          | None  |

Table A.8: Software and hardware environment used for the experiments.

| Component                    | Configuration                                                                                                                |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| Framework                    | PyTorch 2.10, built with CUDA 13.0                                                                                           |
| Python                       | 3.11.14                                                                                                                      |
| Operating system             | Ubuntu 22.04.5 LTS                                                                                                           |
| GPU model                    | NVIDIA A10 24 GB or NVIDIA L40S 48 GB                                                                                        |
| Number of GPUs               | 1                                                                                                                            |
| Experiment configuration     | Hydra 1.3.2                                                                                                                  |
| Dependency management        | uv 0.9.11                                                                                                                    |
| Training monitoring          | Weights & Biases 0.24.2                                                                                                      |
| Data processing and analysis | pandas 3.0.0                                                                                                                 |
| Figure generation            | Matplotlib 3.10.8 and Seaborn 0.13.2                                                                                         |
| Reproducibility              | Seeds are specified, and deterministic settings are enabled. Experiments with $N$ seed runs use seeds $0, 1, \dots, N - 1$ . |

### A.3 Supplementary Experimental Results

This section provides supplementary experimental results for Chapter 5.

#### A.3.1 Depth-wise Logical-Deduction Results

This section supplements the LD Hard analysis in Chapter 5 with depth-wise results for the LD Easy and LD Medium configurations. The figures compare the fixed-computation variants of ST and UT.

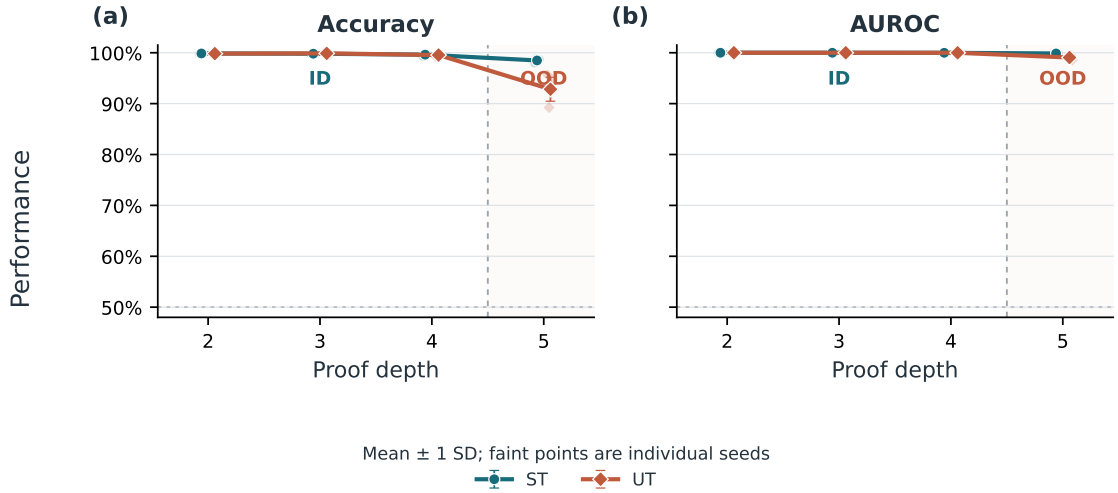


Figure A.5: Depth-wise accuracy and AUROC on LD Easy.

Figure A.5 shows that both models achieve very high accuracy at ID depths, while ST outperforms UT at the OOD depth 5 in both accuracy and AUROC. In addition, ST shows lower seed variance than UT, indicating better learning stability with ST.

Figure A.6 shows that both architectures perform well on ID test examples, with ST showing slightly better performance at higher depths. At OOD depths, ST clearly shows better average performance across seeds with low seed variability. ST also achieves higher AUROC consistently across depths, indicating stronger class separation that transfers to the OOD depths.

#### A.3.2 Additional Fixed-Budget Results for LD Hard

Figure A.7 shows that longer search generally improves model performance. For depths greater than 7, we see that more search does not really improve the performance. Figure A.7 shows an overall performance improvement when increasing the search budget from 16 to 24 steps. We plot the logit distributions of these two search budgets in Figure A.8. It shows a shift in the logit distribution of positive cases. This means that the model has been able to identify more positive examples with very few false negatives.

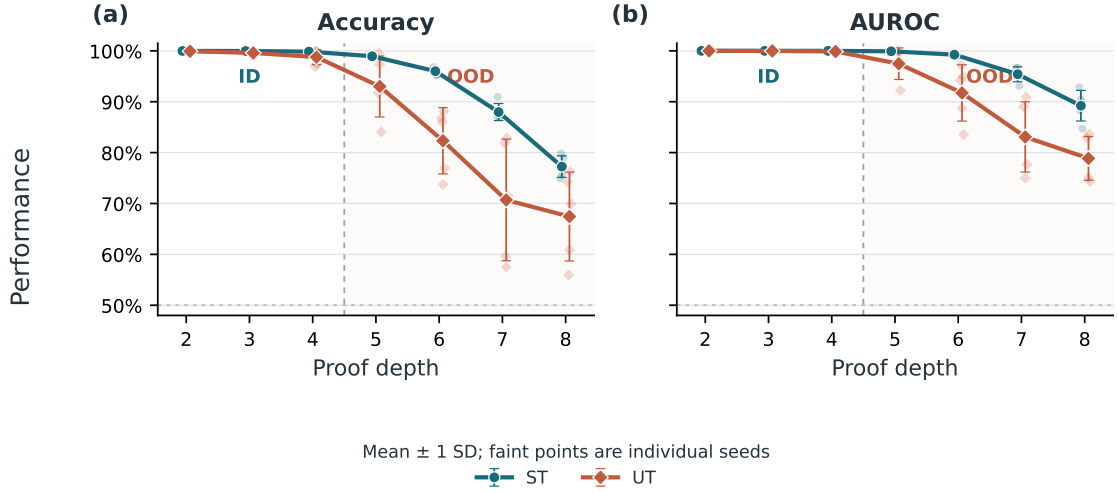


Figure A.6: Depth-wise accuracy and AUROC on LD Medium. The curves report the mean and one standard deviation across five seeds. Proof depths 2–4 belong to the ID test set, whereas depths 5–8 belong to the OOD test set. The horizontal dotted line indicates chance accuracy.

### A.3.3 Additional Fixed-Budget Results for Path Exact-Cover

Figure A.10 shows the logit distributions on the OOD test sets for each run. For depth 7, the model has consistently high AUROC, so it has retained the ability to distinguish the two classes. The AUROC for depth 8 has decreased significantly, indicating the loss of distinguishing ability between the two classes. However, one seed (seed 0) retains a high AUROC for depth 8, indicating that ST can retain class discrimination while still having an issue with threshold calibration for OOD examples.

### A.3.4 Seed Sensitivity on the Convergent Profile

Figure A.11 shows that UT consistently transfers its classification threshold to depth 7 and also shows better class discrimination at depth 8 for most seeds. However, as a post hoc analysis, Figure A.10 shows that the best-performing ST runs have a higher AUROC than all UT runs: seeds 0 and 1 at depth 7, and seed 0 at depth 8. This further highlights that ST, though capable of learning a generalizable discrimination function, often struggles to transfer the matching function responsible for predicting the classification logit to problems not seen during training. We consider learning a generalizable matching function (node classifier) to be future work.

### A.3.5 Additional Per-Seed Results for the State-Alias Profile

Table A.9 provides a detailed performance comparison on the State-Alias profile. Overall, ST shows higher AUROC scores for both OOD depths, while its lower average accuracy results

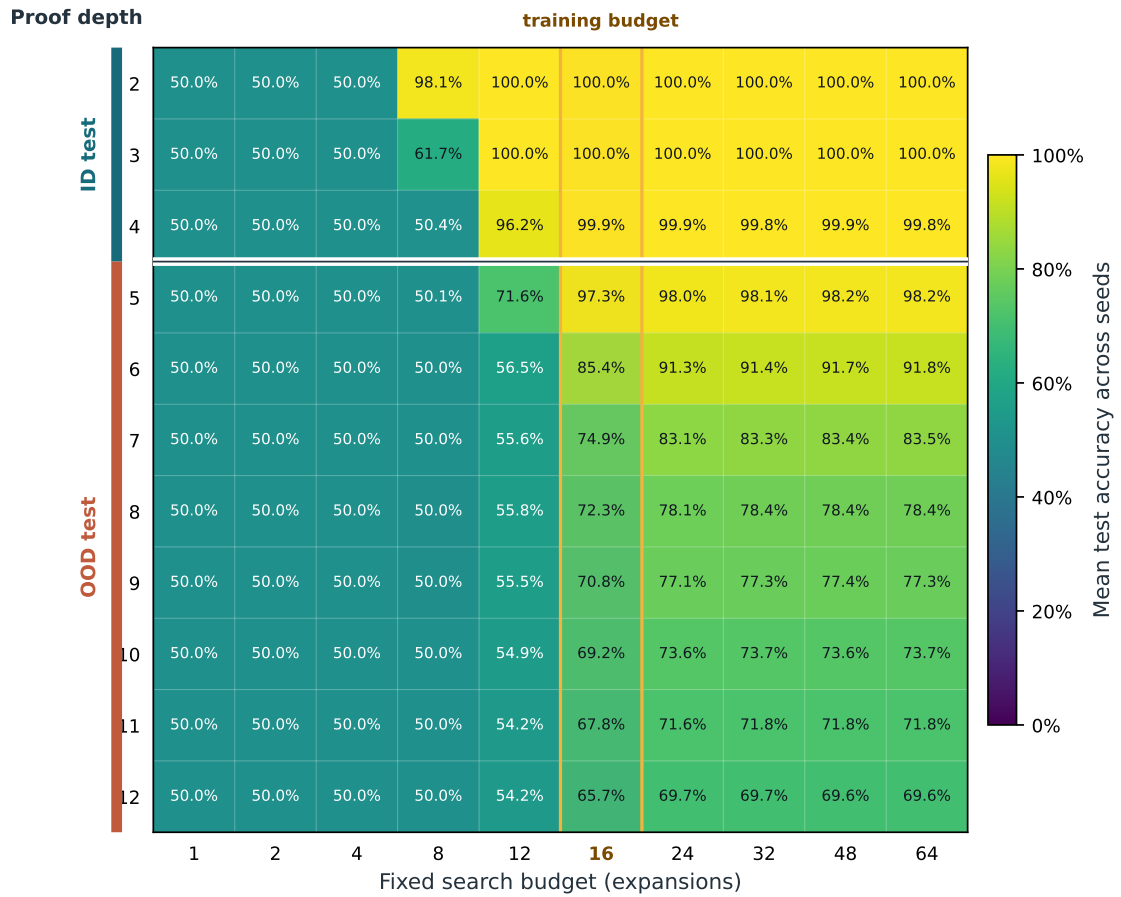


Figure A.7: Heatmap showing the model accuracy when evaluated at different search budgets for each problem depth.

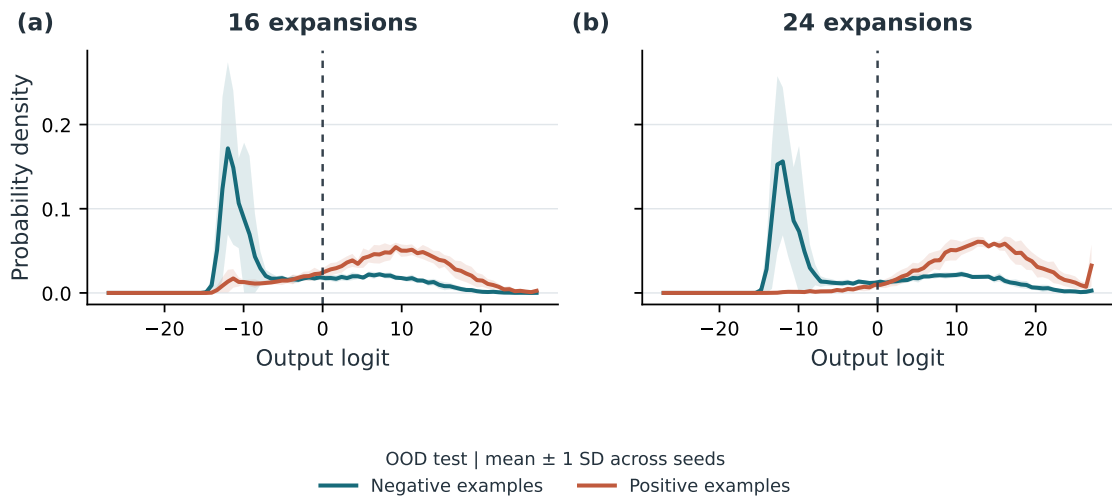


Figure A.8: LD Hard output-logit distributions at different search-step budgets.



Figure A.9: Path-EC Convergent accuracy by path depth and search-step budget.

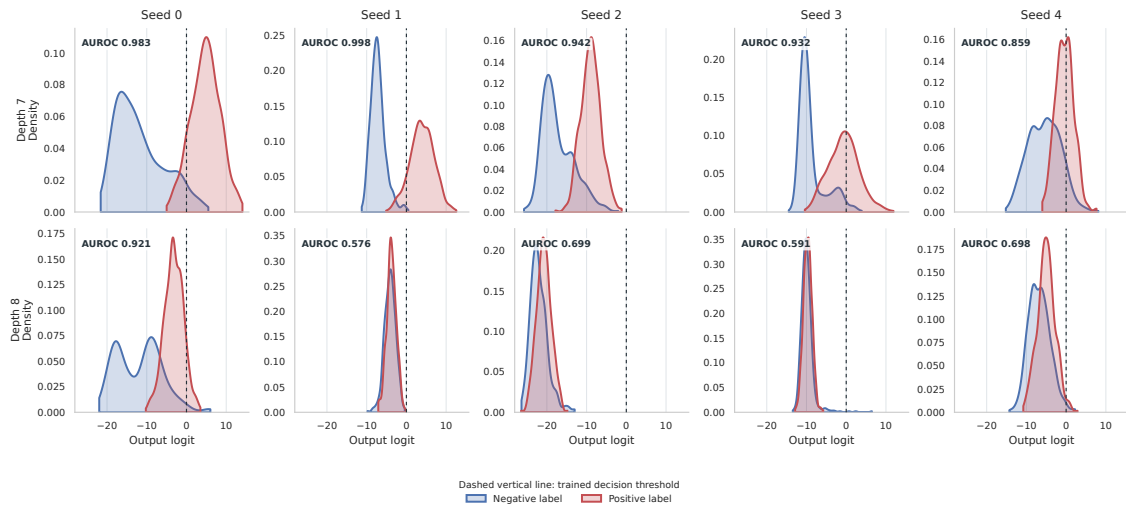


Figure A.10: Class-conditional output-logit distributions of individual ST seeds across the OOD depths of the Path-EC Convergent profile.

from a threshold-transfer issue. This is evident in the seed 3 ST run, where even with an almost perfect AUROC, the accuracy is only 84.4%, which makes the average lower.

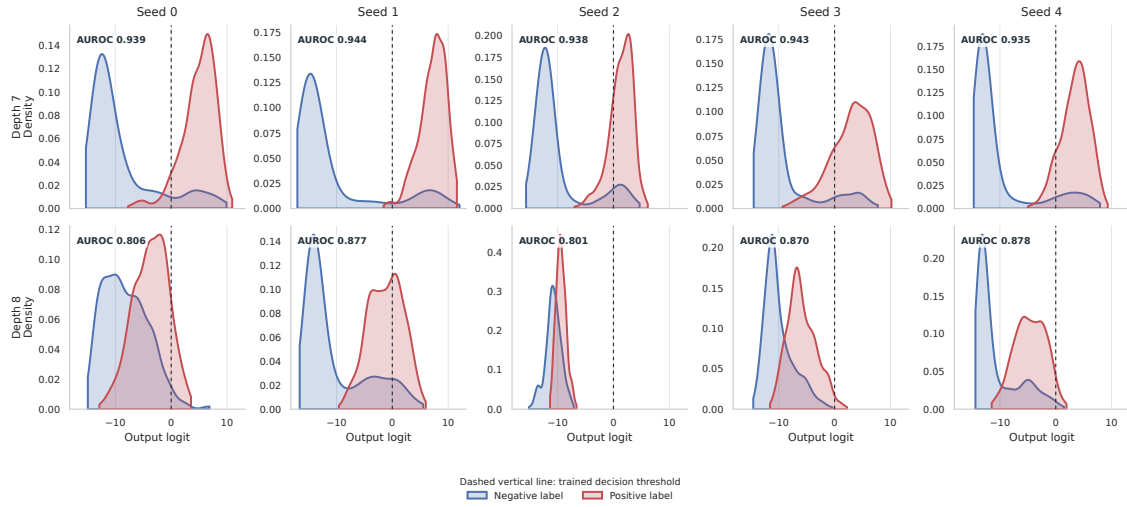


Figure A.11: Class-conditional output-logit distributions of individual UT seeds across the OOD depths of the Path-EC Convergent profile.

Table A.9: Per-seed OOD performance of ST and UT on the Path-EC State-Alias profile. Each depth contains a balanced test set of 896 examples.

| Run   |      | Examples |          |          | ST    |       | UT    |       |
|-------|------|----------|----------|----------|-------|-------|-------|-------|
| Depth | Seed | Total    | Positive | Negative | Acc.  | AUROC | Acc.  | AUROC |
| 7     | 0    | 896      | 448      | 448      | 0.996 | 1.000 | 0.992 | 0.996 |
| 7     | 1    | 896      | 448      | 448      | 0.999 | 1.000 | 1.000 | 1.000 |
| 7     | 2    | 896      | 448      | 448      | 1.000 | 1.000 | 0.978 | 0.989 |
| 7     | 3    | 896      | 448      | 448      | 0.967 | 0.999 | 0.990 | 0.996 |
| 7     | 4    | 896      | 448      | 448      | 0.981 | 1.000 | 0.991 | 0.999 |
| 8     | 0    | 896      | 448      | 448      | 0.981 | 0.997 | 0.990 | 0.994 |
| 8     | 1    | 896      | 448      | 448      | 0.993 | 1.000 | 1.000 | 1.000 |
| 8     | 2    | 896      | 448      | 448      | 0.988 | 1.000 | 0.948 | 0.976 |
| 8     | 3    | 896      | 448      | 448      | 0.844 | 0.997 | 0.987 | 0.998 |
| 8     | 4    | 896      | 448      | 448      | 0.920 | 0.964 | 0.984 | 0.998 |

### A.3.6 Node-Selection Ablation on the State-Alias Profile

Figure A.12 shows that both the accuracy and AUROC of newest-node ST are lower on average for all OOD depths while being close in ID. At depth 7, the accuracy is lower, but the AUROC for some seeds seems to match that of ST with learned selection, although the variance is high. For depth 8, some seeds retain high class separation, but their performance is still below that of ST with learned selection. The variance for newest-node ST is higher across all depths.

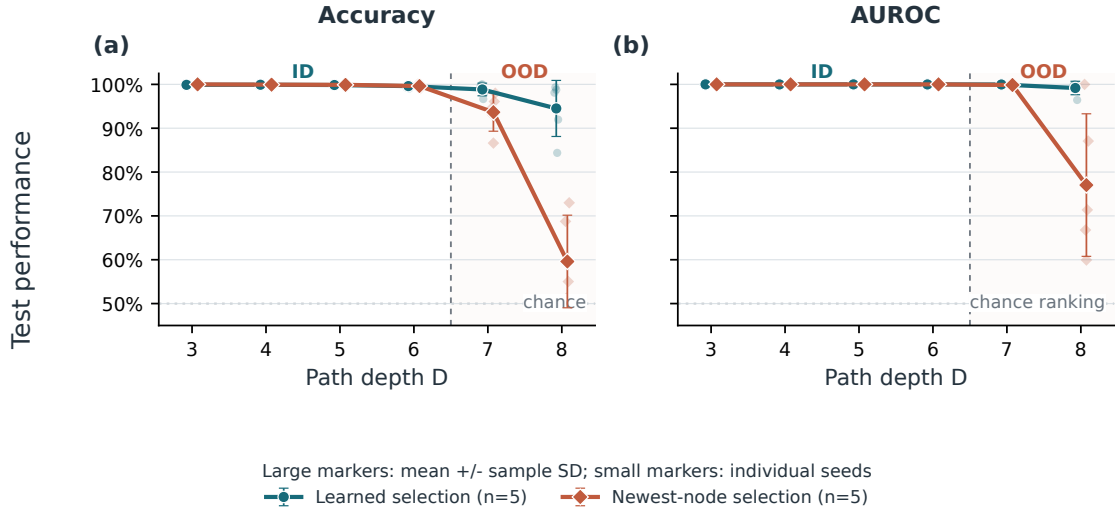


Figure A.12: Depth-wise test performance of the learned node-selection policy and the newest-node policy on the Path-EC State-Alias profile. Panels report (a) accuracy and (b) AUROC for ID depths  $D = 3-6$  and OOD depths  $D = 7-8$ . Small markers show individual seeds, while large markers and error bars show the mean and sample standard deviation across five seeds.

Table A.10: Per-seed OOD performance of the learned and newest-node selection policies on the Path-EC State-Alias profile. The variants differ only in their node-selection policy; both use the memory-pointer action representation. Each depth contains 896 examples, balanced between the two labels. All metrics are reported on a 0–1 scale.

| Run   |      | Learned selection |       |       |       | Newest-node selection |       |       |       |
|-------|------|-------------------|-------|-------|-------|-----------------------|-------|-------|-------|
| Depth | Seed | Acc.              | AUROC | TPR   | TNR   | Acc.                  | AUROC | TPR   | TNR   |
| 7     | 0    | 0.996             | 1.000 | 0.998 | 0.993 | 0.945                 | 1.000 | 1.000 | 0.891 |
| 7     | 1    | 0.999             | 1.000 | 1.000 | 0.998 | 0.931                 | 0.999 | 0.864 | 0.998 |
| 7     | 2    | 1.000             | 1.000 | 1.000 | 1.000 | 0.866                 | 0.999 | 0.732 | 1.000 |
| 7     | 3    | 0.967             | 0.999 | 0.998 | 0.935 | 0.961                 | 0.998 | 0.922 | 1.000 |
| 7     | 4    | 0.981             | 1.000 | 1.000 | 0.962 | 0.980                 | 0.997 | 0.984 | 0.975 |
| 8     | 0    | 0.981             | 0.997 | 0.998 | 0.964 | 0.688                 | 1.000 | 1.000 | 0.375 |
| 8     | 1    | 0.993             | 1.000 | 0.989 | 0.998 | 0.512                 | 0.668 | 0.025 | 1.000 |
| 8     | 2    | 0.988             | 1.000 | 1.000 | 0.975 | 0.500                 | 0.599 | 0.000 | 1.000 |
| 8     | 3    | 0.844             | 0.997 | 1.000 | 0.688 | 0.550                 | 0.714 | 0.100 | 1.000 |
| 8     | 4    | 0.920             | 0.964 | 0.915 | 0.924 | 0.730                 | 0.871 | 0.484 | 0.975 |

Table A.10 provides a per-seed breakdown of performance for OOD depths 7 and 8. We see that seed 0 retains high AUROC for both OOD depths, whereas the other seeds show poor performance in both accuracy and AUROC. So, newest-node ST could learn a generalizable solution, but on average, it performs worse than the learned selection.

## A.4 Search Transformer with Token Inputs

This section presents our preliminary study using a Search Transformer architecture closer to that proposed by [22]. This preliminary study motivated changes to the ST architecture, such as the action pointer and readout pooling, as well as to the task representation, which uses a vector encoding of the problem instead of a tokenized problem description.

### A.4.1 General ST Architecture

We refer to the architecture used here as general ST. This architecture is much closer to the conceptual formulation given by Núñez-Molina et al. [22]. The standard ST architecture used in this thesis can be seen as a specialization of this general ST through the following changes:

1. Defining the action as a memory pointer.
2. Simplifying the transition module using an MLP with a gated update.
3. Generating output with a pooling operation rather than with an output transformer.

**Action definition.** The general ST keeps latent action definition that is predicted by the Expansion Transformer along with the selection logit. Expansion transformer is analogous to our selection module where in addition to the selection logit, it also predicts an action vector

for each search node. So, we obtain Expansion transformer by augmenting selection module with an action head that utilizes the contextual representation of each node to predict action. So the selection logit head and action head use the same contextualized representation but predict two different quantities.

**Transition Component.** We adopt the standard transition transformer, which takes the concatenated state–action representation as input. This representation attends to the input tokens and is updated to obtain the successor node. This is the standard Transition Transformer from [22], so we refer to the proposal paper for more details.

**Output Generation.** The output transformer from [22] can generate outputs autoregressively. Since we solve a decision problem, we generate a single output with two possibilities. The output transformer takes the [CLS] token as input and outputs the classification logit. We concatenate the input with the generated search nodes and provide this sequence, together with the final [CLS] token, to the output transformer.

#### A.4.2 Model Configuration

All three components of general ST are standard Transformers. ET is defined as a stack of self-attention, cross-attention, and MLP layers. It takes search nodes as input to self-attention and input embeddings as input to cross-attention. TT uses cross-attention only, where node–action pairs are provided as input and can attend to the problem tokens. OT is a standard Transformer decoder.

#### A.4.3 Dataset

We use a simplified logical-deduction task. Each problem contains a single known fact, and every rule is a unary implication  $a \rightarrow b$ . The task is therefore equivalent to reachability in a directed graph, where the fact is the start node and the query is the target node. We call this the linear-deduction dataset as the proof paths are linear from source to target. Table A.11 shows the dataset properties and split sizes.

Table A.11: Dataset configuration and split sizes of the linear-deduction dataset.

| Property or split       | Depths | Value  |
|-------------------------|--------|--------|
| Atom vocabulary         | –      | 24     |
| Facts per problem       | –      | 1      |
| Unary rules per problem | –      | 1–8    |
| Training examples       | 1–4    | 16,384 |
| ID test examples        | 1–4    | 4,096  |
| OOD test examples       | 5–6    | 10,240 |

The following is one example:

```

Facts: stormy

Rules: stormy -> wrong
      aggressive -> tense
      wrong -> aggressive
      diplomatic -> cruel
      frightened -> helpful
      tense -> glamorous
      cruel -> jittery
      jittery -> frightened

Query: glamorous
Label: 1
Generation depth: 4

```

The label and generation depth in the above example are metadata provided only for illustration, and the model does not have access to them.

We represent this problem as a sequence of tokens as follows:

```

[FACT] stormy [END]

[RULE] stormy [IMPLIES] wrong [END]
[RULE] aggressive [IMPLIES] tense [END]
[RULE] wrong [IMPLIES] aggressive [END]
[RULE] diplomatic [IMPLIES] cruel [END]
[RULE] frightened [IMPLIES] helpful [END]
[RULE] tense [IMPLIES] glamorous [END]
[RULE] cruel [IMPLIES] jittery [END]
[RULE] jittery [IMPLIES] frightened [END]

[QUERY] glamorous [END]

```

The main difference from the standard task formulation is that the inputs are not encoded as vectors, but rather provided as a sequence of tokens to the learnable embedding layer.

#### A.4.4 Training Objective

We use the standard task-level training objective defined in Section 4.5.1 for the control experiment. We also consider intermediate-state supervision, which we describe in this section.

##### Intermediate-State Supervision

In many problems, it might be possible to obtain intermediate states that need to be reached when solving a problem. In such cases, we can add an auxiliary supervision signal by training ST to generate search nodes that represent those intermediate states. The advantages of this approach include better convergence due to the additional learning signal and more interpretable search nodes.

**Generating intermediate supervision targets.** For each positive example, we use a symbolic solver to recover the shortest ordered proof chain from the initial fact to the query. If the shortest proof depth of the query is  $d$ , the chain contains  $d + 1$  atoms: the initial fact followed by the atom derived after each of the  $d$  rule applications, where the final atom is the query. We use these atoms in their derivation order as the intermediate supervision targets. Since negative examples do not have a proof chain for the query, we train them with only the task-level classification objective.

**Projection into a shared representation space.** Let  $A_{\text{int}} = [a_1, \dots, a_m]$  be the ordered list of intermediate target atoms. For each target  $a_i$ , we obtain its embedding  $e_i = E(a_i)$  using the same embedding layer as the input tokens. Since the target embeddings and search nodes may exist in different representation spaces, we use two learned linear layers to project them into a shared matching space and compute their matching cost using negative cosine similarity:

$$c(e_i, n_j) = -\frac{z_i^T s_j}{\|z_i\|_2 \|s_j\|_2}, \quad z_i = W_t e_i + b_t, \quad s_j = W_n n_j + b_n.$$

where  $W_t, W_n \in \mathbb{R}^{d_m \times d_m}$  and  $b_t, b_n \in \mathbb{R}^{d_m}$  are learnable parameters. Both projections preserve the model dimension  $d_m$ .  $z_i$  is the projected target representation and  $s_j$  is the projected search-node representation.

**Greedy Monotonic Assignment.** We use a greedy monotonic assignment algorithm where we choose best matching search nodes for each intermediate state greedily. We also experimented with the Hungarian matching algorithm, but we are not including it here as the simple monotonic matching provides similar convergence while providing faster computation. The matching preserves the given state order across the generated nodes, such that a state occurring later in the derivation is matched only to a node generated after those matched to earlier states. Let  $j_i$  denote the index of the node matched to  $z_i$ , with  $j_0 = 0$ . We sequentially assign each intermediate state to the closest available node as

$$j_i = \arg \min_{k \in \{j_{i-1}+1, \dots, T-m+i\}} c(z_i, n_k), \quad i = 1, \dots, m,$$

where ties are resolved by selecting the earliest node. The upper bound  $T - m + i$  reserves enough nodes for the remaining intermediate states. Consequently,

$$1 \leq j_1 < j_2 < \dots < j_m \leq T,$$

and the matched nodes are given by

$$M_{\text{int}} = [n_{j_1}, \dots, n_{j_m}].$$

**Auxiliary objective.** Given the matched nodes  $M_{\text{int}} = [\mu_1, \dots, \mu_m]$ , where  $\mu_i = n_{j_i}$ , we define the auxiliary loss as

$$\mathcal{L}_{\text{int}}(Z_{\text{int}}, M_{\text{int}}) = \frac{1}{m} \sum_{i=1}^m c(z_i, \mu_i). \quad (\text{A.1})$$

**Combined objective.** For the fixed-step experiments considered here, the task and intermediate-state objectives at training step  $t$  are combined as

$$\mathcal{L}(t) = \mathcal{L}_{\text{task}} + \lambda_{\text{int}}(t) \mathcal{L}_{\text{int}}, \quad (\text{A.2})$$

where  $\mathcal{L}_{\text{task}}$  is the final binary classification loss and  $\lambda_{\text{int}}(t) \in \mathbb{R}_{\geq 0}$  controls the weight of the auxiliary loss. We initialize  $\lambda_{\text{int}}(0) = 1.0$  and anneal it to  $10^{-4}$  over the first 8,000 training steps. We then keep it fixed at  $10^{-4}$  for the remaining training steps.

#### A.4.5 Results

We compare the performance of general ST with and without intermediate-state supervision. In addition, we also train a similar sized UT as a diagnostic measure. The UT here does not use intermediate-state supervision, so its equivalent comparison is with the general ST with only the task level supervision.

Table A.12: Performance comparison of ST, ST with state supervision, and UT on the linear-deduction dataset with token inputs. Each entry reports the mean and standard deviation over three training seeds. *Comp.* denotes the number of runs reaching the final curriculum rung out of three.

| Model                       | In-distribution   |                   | Out-of-distribution |                   | Comp. |
|-----------------------------|-------------------|-------------------|---------------------|-------------------|-------|
|                             | Acc.              | AUROC             | Acc.                | AUROC             |       |
| UT                          | $1.0 \pm 0.0$     | $1.0 \pm 0.0$     | $0.998 \pm 0.002$   | $1.0 \pm 0.0$     | 3/3   |
| ST                          | $0.519 \pm 0.017$ | $0.541 \pm 0.022$ | $0.503 \pm 0.007$   | $0.492 \pm 0.025$ | 0/3   |
| ST (with state supervision) | $0.985 \pm 0.012$ | $0.999 \pm 0.001$ | $0.873 \pm 0.105$   | $0.989 \pm 0.014$ | 3/3   |

Table A.12 shows that UT reliably learns the task using only decision label supervision, achieving near-perfect ID and OOD performance across all three runs. In contrast, the general ST remains close to chance in both accuracy and AUROC, and none of its runs reaches the final curriculum rung. Adding intermediate-state supervision substantially changes its learnability: all three runs complete the curriculum and achieve an ID accuracy of  $98.5 \pm 1.2\%$  with an AUROC of  $0.999 \pm 0.001$ . Its average OOD accuracy falls to  $87.3 \pm 10.5\%$  and AUROC to  $0.989 \pm 0.014$ , showing high variance in accuracy across seeds. The three state-supervised ST runs achieve OOD accuracies of 75.14%, 93.7%, and 92.9%, and AUROCs of 0.99, 0.97, and 0.99, respectively. This shows that intermediate-state supervision can produce solutions with strong OOD class separation, while accuracy is not reliable across seeds.

We treat this as a preliminary experiment, rather than a direct comparison with the main findings, because it uses a simplified dataset and solver-generated intermediate targets. Nevertheless, these results suggest that intermediate-state supervision is a promising approach for improving the learnability of ST.

# List of Acronyms

|             |                                   |
|-------------|-----------------------------------|
| <b>ACT</b>  | Adaptive Computation Time         |
| <b>BCE</b>  | Binary Cross-Entropy              |
| <b>CoT</b>  | Chain-of-Thought                  |
| <b>DL</b>   | Deep Learning                     |
| <b>ES</b>   | Embedded Space                    |
| <b>I2A</b>  | Imagination-Augmented Agents      |
| <b>IBP</b>  | Imagination-Based Planner         |
| <b>IS</b>   | Instance Space                    |
| <b>KL</b>   | Kullback–Leibler                  |
| <b>LLM</b>  | Large Language Model              |
| <b>LSTM</b> | Long Short-Term Memory            |
| <b>LT</b>   | Looped Transformer                |
| <b>MCTS</b> | Monte-Carlo Tree Search           |
| <b>MIL</b>  | Multiple Instance Learning        |
| <b>MLP</b>  | Multilayer Perceptron             |
| <b>RNN</b>  | Recurrent Neural Network          |
| <b>ROC</b>  | Receiver Operating Characteristic |
| <b>ST</b>   | Search Transformer                |
| <b>TNR</b>  | True Negative Rate                |
| <b>ToT</b>  | Tree of Thoughts                  |
| <b>TPR</b>  | True Positive Rate                |
| <b>UT</b>   | Universal Transformer             |
| <b>VPN</b>  | Value Prediction Network          |

# List of Symbols

## Learning and Prediction

|                             |                                                                      |
|-----------------------------|----------------------------------------------------------------------|
| $\mathcal{D}$               | dataset of input-label pairs                                         |
| $x_i = (E_i, q_i)$          | $i$ -th encoded problem instance, with context $E_i$ and query $q_i$ |
| $y_i$                       | binary target decision label for input $x_i$                         |
| $\hat{y}$                   | binary model prediction                                              |
| $\mathcal{L}_{\text{task}}$ | binary cross-entropy task loss                                       |
| $\mathcal{L}_{\text{ac}}$   | ST-Ponder objective: expected task loss plus KL regularization       |

## State-Space Search

|               |                                                                        |
|---------------|------------------------------------------------------------------------|
| $\mathcal{S}$ | state space                                                            |
| $\mathcal{A}$ | action space in the state-space search formulation                     |
| $s_0$         | initial state                                                          |
| $S_g$         | set of goal states                                                     |
| $\mathcal{T}$ | state-transition function mapping a state-action pair to its successor |

## Latent Recurrent Transformers

|               |                                                                       |
|---------------|-----------------------------------------------------------------------|
| $\mathcal{V}$ | Transformer representation space                                      |
| $d_m$         | Transformer model dimension                                           |
| $S_\tau$      | latent state at recurrent step $\tau$                                 |
| $n_\tau$      | number of latent state vectors at recurrent step $\tau$               |
| $F_\phi$      | parameter-shared Transformer component applied at each recurrent step |
| $u$           | latent-state update operator                                          |

## Search Transformer

|                |                                                                       |
|----------------|-----------------------------------------------------------------------|
| $\mathcal{E}$  | space of encoded problem elements                                     |
| $\mathcal{N}$  | space of search-node representations                                  |
| $E$            | unordered set of encoded elements representing the problem context    |
| $e_i$          | $i$ -th encoded element of the problem context                        |
| $q$            | encoded query or goal element                                         |
| $d$            | shared dimension of problem elements, query vectors, and search nodes |
| $\mathbf{0}_d$ | $d$ -dimensional all-zero vector                                      |
| $\mathbf{1}_d$ | $d$ -dimensional all-ones vector                                      |
| $K$            | number of elements in the problem context                             |
| $t$            | search-step index                                                     |
| $T$            | configured number of search steps                                     |

|                           |                                                                                                  |
|---------------------------|--------------------------------------------------------------------------------------------------|
| $n_0$                     | initial search node                                                                              |
| $n_i, n_t$                | search node indexed by $i$ ; $n_t$ is appended at search step $t$                                |
| $N_t$                     | ordered collection of search nodes available for selection at search step $t$                    |
| $f_{\text{sel}}$          | function producing selection logits over the search nodes in $N_t$                               |
| $w_{\text{sel},i}^{(t)}$  | soft selection weight assigned to node $n_i$ at search step $t$                                  |
| $\pi_{i,k}^{\text{base}}$ | cached base probability of selecting context element $e_k$ as the action for node $n_i$          |
| $\pi_{i,k}^{(t)}$         | probability of selecting $e_k$ as the action for node $n_i$ after masking at search step $t$     |
| $a_i^{(t)}$               | soft action representation obtained for node $n_i$ by a weighted context read at search step $t$ |
| $\tau_{\text{act}}$       | temperature controlling the sharpness of the action-pointer distribution                         |
| $f_{\text{tran}}$         | deterministic shared transition function mapping a node-action pair to a candidate successor     |
| $\hat{n}_i^{(t)}$         | candidate successor generated from node $n_i$ at the current search step, before aggregation     |
| $r_{i,k}^{(t)}$           | remaining availability of node-action pair $(n_i, e_k)$ after its update at search step $t$      |
| $c_{i,k}^{(t)}$           | joint soft contribution weight of node-action pair $(n_i, e_k)$ at search step $t$               |
| $\alpha$                  | strength of the action-availability penalty                                                      |

### Adaptive Computation

|                          |                                                                               |
|--------------------------|-------------------------------------------------------------------------------|
| $\ell_t$                 | pooled output logit after $t$ search steps                                    |
| $\lambda_t$              | probability of halting at search step $t$ , conditioned on reaching that step |
| $p_t$                    | unconditional probability of halting exactly at search step $t$               |
| $\tilde{p}$              | normalized finite-horizon halting distribution used during training           |
| $\tilde{p}_G(\lambda_p)$ | normalized finite-horizon geometric prior over halting steps                  |
| $c_t$                    | cumulative probability of halting by search step $t$                          |
| $t_{\text{halt}}$        | search step at which adaptive inference stops                                 |
| $T_{\text{max}}$         | maximum search horizon allowed during inference                               |
| $\delta_{\text{halt}}$   | cumulative-probability threshold used to halt inference                       |
| $\lambda_p$              | per-step halting-probability parameter of the geometric prior                 |
| $\beta$                  | weight of the KL regularizer on the halting distribution                      |

### Reasoning Tasks

|               |                                                 |
|---------------|-------------------------------------------------|
| $\text{At}$   | finite vocabulary of atoms in logical deduction |
| $F$           | set of known logical-deduction facts            |
| $\mathcal{R}$ | set of logical-deduction rules                  |
| $\mathcal{C}$ | finite set of possible Path-EC edge-label items |
| $C$           | target item set in Path-EC                      |

|     |                                                                                         |
|-----|-----------------------------------------------------------------------------------------|
| $B$ | number of deceptive paths constructed in each direction in the Disjoint Path-EC profile |
| $P$ | length of the valid target-item prefix of a deceptive Path-EC path                      |

# List of Figures

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |    |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1 | The components of a Transformer. A Transformer block combines multi-head attention with residual connections, normalization, and a feed-forward network. Each attention head converts query–key compatibility scores into weights $A_h$ used to combine the values. . . . .                                                                                                                                                                                                                                                                                               | 8  |
| 2.2 | Forward-chaining logical deduction with query $Q$ . <b>(a)</b> A positive instance with facts $\{A, B, C\}$ , where applying $r_1$ , $r_2$ , and $r_3$ derives $H_0$ , $H_1$ , and $Q$ . <b>(b)</b> A negative instance with facts $\{A, C, D\}$ . The missing fact $B$ blocks $r_1$ , so $H_0$ and $Q$ cannot be derived, while $r_2$ and the distractor rule $r_d$ remain applicable. Dashed elements show the blocked part of the proof. . . . .                                                                                                                       | 19 |
| 2.3 | A representative example from the Disjoint Path-EC profile with target set $C = \{a, b, c\}$ . In the positive instance (left), the highlighted path from $s$ to $t$ collects $a$ , $b$ , and $c$ exactly once. The paired counterfactual negative (right) is obtained by exchanging two edge labels. This preserves the graph topology and global item counts but removes every valid exact-cover path. . . . .                                                                                                                                                          | 20 |
| 4.1 | One search step works by reading information available on the tape, performing some computation, and writing a new item to the first available space on the right. The query $q$ , problem context $E$ , and the starting node $n_0$ are already populated on the tape. A search step $t$ appends a node $n_t$ . The search process is iterative and is carried out until (1) the search budget runs out, (2) a solution (either positive or negative) is found, or (3) a halting condition is met, where each of these conditions depends on the implementation. . . . . | 31 |
| 4.2 | Selection over the frontier nodes. Self-attention contextualizes the frontier nodes, cross-attention reads the problem memory, and a position-wise MLP produces one selection logit for each node. . . . .                                                                                                                                                                                                                                                                                                                                                                | 33 |
| 4.3 | Candidate expansion and aggregation at search step $t$ . Left: the transition module maps one node–action pair $(n_i, a_i^{(t)})$ to a candidate successor $\hat{n}_i^{(t)}$ . Right: the shared transition module shown on the left, abbreviated as $f_{\text{tran}}$ , expands all frontier nodes in parallel. The selection weights $w_{\text{sel},i}^{(t)}$ aggregate the candidate successors into the single successor node $n_t$ , which is appended to the frontier. . . . .                                                                                      | 36 |
| 4.4 | Query-conditioned readout. Each generated search node is scored against the query, and the resulting node logits are pooled into a single prediction logit. . . . .                                                                                                                                                                                                                                                                                                                                                                                                       | 40 |

|      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |    |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 5.1  | Depth-wise performance comparison on LD Hard. Plots report the mean and one standard deviation across five seeds. (a) reports accuracy, and (b) reports the area under the ROC curve. . . . .                                                                                                                                                                                                                                                                                           | 52 |
| 5.2  | Average performance on the OOD test set over different seeds. Small points represent individual seeds, while large markers and whiskers show the mean and one standard deviation across seeds. . . . .                                                                                                                                                                                                                                                                                  | 52 |
| 5.3  | Performance comparison on test problems obtained with query substitution. (a) Overall recall for derivable queries and specificity for non-derivable queries. (b) Derivable query recall for each proof depth. Small points denote five independently trained seeds; large markers or lines show the mean, and whiskers indicate one standard deviation. ST maintains higher recall with substantially lower seed sensitivity, whereas UT becomes less reliable on deeper proofs. . . . | 54 |
| 5.4  | Performance on the validation set at different training steps. (a) Balanced accuracy on the validation set over the number of training examples seen. The faint lines represent individual-seed performance. The bold curves and shaded areas show the mean and one standard deviation across seeds. (b) The number of examples required to exceed 90% validation accuracy for each seed. We report the median across the five seeds. . . . .                                           | 55 |
| 5.5  | Distribution of the search step at which the prediction first becomes stably positive for LD Hard problems, grouped by proof depth. Each cell shows the mean probability across seeds. $N$ denotes problems for which the prediction never becomes stably positive. The upper and lower panels show the ID and OOD test sets, respectively. . . . .                                                                                                                                     | 55 |
| 5.6  | Performance comparison on the Disjoint dataset. All five ST runs and only a single UT run completed the curriculum. We include results only from completed runs here. (a) shows accuracy, and (b) shows pair accuracy. . . . .                                                                                                                                                                                                                                                          | 56 |
| 5.7  | Pair accuracy on the Path-EC Disjoint profile by depth, branch count, and prefix length. . . . .                                                                                                                                                                                                                                                                                                                                                                                        | 57 |
| 5.8  | True negative rate on original and path-splice negative test examples. Five completed runs were used for ST and a single run for UT. . . . .                                                                                                                                                                                                                                                                                                                                            | 58 |
| 5.9  | The distribution of the positions at which the prediction logits for positive examples exceed 0 and remain above 0. . . . .                                                                                                                                                                                                                                                                                                                                                             | 59 |
| 5.10 | Per-seed accuracy and AUROC performance of ST and UT on the Path-EC Convergent profile. . . . .                                                                                                                                                                                                                                                                                                                                                                                         | 60 |
| 5.11 | Class-conditional logit distributions for the best- and worst-performing ST checkpoints on depth-7 Path-EC Convergent examples. . . . .                                                                                                                                                                                                                                                                                                                                                 | 60 |

|      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |    |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 5.12 | Performance after threshold calibration for the weakest ST seed at depth 7. At the trained threshold $t = 0$ , accuracy is 50.0%. Selecting a threshold $t^* = -12.73$ on the OOD validation set increases accuracy to 88.3%, with a TPR of 92.9% and a TNR of 83.7%. The best seed achieves 94.4% accuracy at $t = 0$ . . . . .                                                                                                                                                                                                                                                                  | 61 |
| 5.13 | Comparison of accuracy and AUROC for ST and UT on the OOD State-Alias dataset. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 62 |
| 5.14 | ST performance for different search steps: (a) compares accuracy for ID and OOD, and (b) shows the AUROC result. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | 63 |
| 5.15 | Performance on the Disjoint profile across different inference-time computation budgets. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  | 63 |
| 5.16 | Comparison of fixed-step ST and ST-Ponder on LD Hard. Curves show the mean and one standard deviation across five seeds, with faint markers denoting individual seeds. (a) compares the accuracy of ST and ST-Ponder, and (b) shows the expected number of steps taken for each depth. . . . .                                                                                                                                                                                                                                                                                                    | 64 |
| 5.17 | Accuracy and AUROC comparison of the standard learned node-selection policy and the newest-node policy on the Path-EC State-Alias profile. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                | 66 |
| A.1  | Feature encoding of logical-deduction objects for the ordered example vocabulary $\mathcal{V} = (a, b, c, d)$ . The body block is multi-hot, the head block is one-hot, and the final scalar marks the query object. Facts and queries have empty bodies. For a vocabulary of size $N_v$ , each feature vector has dimension $2N_v + 1$ . . . . .                                                                                                                                                                                                                                                 | 73 |
| A.2  | Feature encoding of a Path-EC edge and query for the ordered node vocabulary $\mathcal{V}_n = (s, t)$ and item vocabulary $\mathcal{V}_i = (a, b)$ . The edge $s \xrightarrow{a} t$ activates its item, source, and target positions. The query $(s, t, \{a\})$ activates the goal, target-item, and query-start positions. Each feature vector has dimension $4N_n + N_i$ . . . . .                                                                                                                                                                                                              | 78 |
| A.3  | Path-EC Disjoint example families at depth $D = 3$ . In each row, the negative example is obtained by exchanging the two labels highlighted in yellow. This keeps the graph structure, query, and global item counts fixed but removes the exact-cover witness. Increasing $B$ replaces immediately rejectable decoy paths with paths that begin or end with target items. . . . .                                                                                                                                                                                                                | 79 |
| A.4  | The Convergent profile (top row) introduces alternative decoy paths that diverge from and later converge with the witness path. The State-Alias profile allows different valid collected-item sets to reach the same alias node $v$ , shown with a double outline. Therefore, the current graph node alone is not sufficient to describe the search state; the state also requires the cover up to this point. In each row, the right panel is obtained by exchanging the two labels highlighted in yellow and is the solver-verified counterfactual negative of the example on its left. . . . . | 80 |
| A.5  | Depth-wise accuracy and AUROC on LD Easy. . . . .                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 | 83 |

|      |                                                                                                                                                                                                                                                                                                                                                                            |    |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| A.6  | Depth-wise accuracy and AUROC on LD Medium. The curves report the mean and one standard deviation across five seeds. Proof depths 2–4 belong to the ID test set, whereas depths 5–8 belong to the OOD test set. The horizontal dotted line indicates chance accuracy. . . . .                                                                                              | 84 |
| A.7  | Heatmap showing the model accuracy when evaluated at different search budgets for each problem depth. . . . .                                                                                                                                                                                                                                                              | 85 |
| A.8  | LD Hard output-logit distributions at different search-step budgets. . . . .                                                                                                                                                                                                                                                                                               | 85 |
| A.9  | Path-EC Convergent accuracy by path depth and search-step budget. . . . .                                                                                                                                                                                                                                                                                                  | 86 |
| A.10 | Class-conditional output-logit distributions of individual ST seeds across the OOD depths of the Path-EC Convergent profile. . . . .                                                                                                                                                                                                                                       | 86 |
| A.11 | Class-conditional output-logit distributions of individual UT seeds across the OOD depths of the Path-EC Convergent profile. . . . .                                                                                                                                                                                                                                       | 87 |
| A.12 | Depth-wise test performance of the learned node-selection policy and the newest-node policy on the Path-EC State-Alias profile. Panels report (a) accuracy and (b) AUROC for ID depths $D = 3-6$ and OOD depths $D = 7-8$ . Small markers show individual seeds, while large markers and error bars show the mean and sample standard deviation across five seeds. . . . . | 88 |

# List of Tables

|     |                                                                                                                                                                                                                                                                                                                                                                  |    |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 4.1 | Ordered curriculum rungs used for each training configuration. In the Disjoint profile, $B$ is the deceptive branch count and $D$ is the problem depth. . . . .                                                                                                                                                                                                  | 44 |
| 5.1 | Dataset configurations used for training and evaluation. The in-distribution (ID) depths are the problem depths used for training, whereas the out-of-distribution (OOD) depths correspond to problems requiring more steps to solve than the ID problems. . . . .                                                                                               | 47 |
| 5.2 | Base training and optimization settings, listed in approximate order of importance for interpreting and reproducing the training procedure. . . . .                                                                                                                                                                                                              | 49 |
| 5.3 | Overall accuracy comparison of ST and UT on different dataset profiles. $Acc.$ is the model accuracy and $Comp.$ is the number of completed runs out of five seeds. Accuracy entries report the mean and standard deviation across completed runs; when only one run completed, only its accuracy is shown. . .                                                  | 51 |
| 5.4 | Results of the combined node-selection and action/transition-design ablation. Performance entries report the mean and standard deviation across five seeds. For runs that did not reach the final curriculum rung, performance is taken from the final available checkpoint. $Comp.$ gives the number of runs that completed the curriculum out of five. . . . . | 65 |
| 5.5 | Ablation results comparing the latent-action and no-action variants on LD Medium. The reference row reports the standard ST configuration over five seeds, while the ablation rows report the mean and standard deviation over three seeds. $Comp.$ gives the number of completed runs. . . . .                                                                  | 67 |
| 5.6 | Readout paradigm ablation on LD Hard. Each entry reports the mean and standard deviation over three training seeds. . . . .                                                                                                                                                                                                                                      | 68 |
| A.1 | Logical-deduction configurations used in the experiments. . . . .                                                                                                                                                                                                                                                                                                | 71 |
| A.2 | Logical-deduction split sizes. The datasets contain the same number of examples for each depth and label. The OOD totals differ because the three configurations cover different numbers of OOD depths. . . . .                                                                                                                                                  | 71 |
| A.3 | Configurations used to generate the Path Exact-Cover benchmark profiles. Here, $D$ is the target-set size and the required path length, $K$ is the number of decoy paths, $B$ is the number of deceptive paths in each direction, and $P$ is the length of a deceptive path’s valid segment. . . . .                                                             | 77 |

|      |                                                                                                                                                                                                                                                                                                                                                                                            |    |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| A.4  | Path-EC split sizes. Each family contains one positive example and its paired counterfactual negative, and therefore contributes two examples. The two per-depth columns apply to every profile; the profile columns report the total number of examples across the corresponding depth range. The three Disjoint conditions are combined when training the single Disjoint model. . . . . | 77 |
| A.5  | Input feature dimensions and their projection into the latent model space. Both architectures use the same input features and a bias-free linear projection; only the output dimension of the projection differs. . . . .                                                                                                                                                                  | 81 |
| A.6  | Main model parameters used for ST and ST-Ponder. Both variants use the same configuration, except for the additional halting head used by ST-Ponder. . . . .                                                                                                                                                                                                                               | 81 |
| A.7  | Model parameters used for UT. . . . .                                                                                                                                                                                                                                                                                                                                                      | 82 |
| A.8  | Software and hardware environment used for the experiments. . . . .                                                                                                                                                                                                                                                                                                                        | 82 |
| A.9  | Per-seed OOD performance of ST and UT on the Path-EC State-Alias profile. Each depth contains a balanced test set of 896 examples. . . . .                                                                                                                                                                                                                                                 | 87 |
| A.10 | Per-seed OOD performance of the learned and newest-node selection policies on the Path-EC State-Alias profile. The variants differ only in their node-selection policy; both use the memory-pointer action representation. Each depth contains 896 examples, balanced between the two labels. All metrics are reported on a 0–1 scale. . . . .                                             | 89 |
| A.11 | Dataset configuration and split sizes of the linear-deduction dataset. . . . .                                                                                                                                                                                                                                                                                                             | 90 |
| A.12 | Performance comparison of ST, ST with state supervision, and UT on the linear-deduction dataset with token inputs. Each entry reports the mean and standard deviation over three training seeds. <i>Comp.</i> denotes the number of runs reaching the final curriculum rung out of three. . . . .                                                                                          | 93 |

# List of Algorithms

|   |                                         |    |
|---|-----------------------------------------|----|
| 1 | Fixed-Step Search Transformer . . . . . | 41 |
|---|-----------------------------------------|----|

## List of References

- [1] J. Amores, “Multiple instance classification: Review, taxonomy and comparative study,” *Artificial intelligence*, vol. 201, pp. 81–105, 2013.
- [2] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” *arXiv preprint arXiv:1607.06450*, 2016.
- [3] A. Banino, J. Balaguer, and C. Blundell, *PonderNet: Learning to Ponder*, Sep. 2021. DOI: 10.48550/arXiv.2107.05407 arXiv: 2107.05407 [cs]. Accessed: Aug. 1, 2025.
- [4] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, “Curriculum learning,” in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML ’09, New York, NY, USA: Association for Computing Machinery, Jun. 14, 2009, pp. 41–48, ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553380 Accessed: Feb. 7, 2026. [Online]. Available: <https://dl.acm.org/doi/10.1145/1553374.1553380>
- [5] X. Chen, J. Xu, T. Liang, Z. He, J. Pang, D. Yu, L. Song, Q. Liu, M. Zhou, Z. Zhang, et al., “Do not think that much for  $2+3=?$  on the overthinking of o1-like llms,” *arXiv preprint arXiv:2412.21187*, 2024.
- [6] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Ding, H. Xin, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Wang, J. Chen, J. Yuan, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, S. Ye, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Zhao, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Xu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu,

- Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang, *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, Jan. 2025. DOI: 10.48550/arXiv.2501.12948 arXiv: 2501.12948 [cs]. Accessed: Aug. 14, 2025.
- [7] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and Ł. Kaiser, *Universal Transformers*, Mar. 2019. DOI: 10.48550/arXiv.1807.03819 arXiv: 1807.03819 [cs]. Accessed: Jun. 27, 2025.
- [8] Y. Fan, Y. Du, K. Ramchandran, and K. Lee, “Looped transformers for length generalization,” in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: <https://openreview.net/forum?id=2edigk8yoU>
- [9] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, *TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep Reinforcement Learning*, Mar. 2018. DOI: 10.48550/arXiv.1710.11417 arXiv: 1710.11417 [cs]. Accessed: Jun. 27, 2025.
- [10] J. Geiping, S. McLeish, N. Jain, J. Kirchenbauer, S. Singh, B. R. Bartoldson, B. Kailkhura, A. Bhatele, and T. Goldstein, *Scaling up Test-Time Compute with Latent Reasoning: A Recurrent Depth Approach*, Feb. 2025. DOI: 10.48550/arXiv.2502.05171 arXiv: 2502.05171 [cs]. Accessed: Aug. 1, 2025.
- [11] A. Giannou, S. Rajput, J.-Y. Sohn, K. Lee, J. D. Lee, and D. Papailiopoulos, “Looped transformers as programmable computers,” in *Proceedings of the 40th International Conference on Machine Learning*, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., ser. Proceedings of Machine Learning Research, vol. 202, PMLR, 23–29 Jul 2023, pp. 11 398–11 442. [Online]. Available: <https://proceedings.mlr.press/v202/giannou23a.html>
- [12] A. Graves, *Adaptive Computation Time for Recurrent Neural Networks*, Feb. 2017. DOI: 10.48550/arXiv.1603.08983 arXiv: 1603.08983 [cs]. Accessed: Jul. 31, 2025.
- [13] C. Grimm, A. Barreto, S. Singh, and D. Silver, “The value equivalence principle for model-based reinforcement learning,” in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS ’20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.
- [14] S. Hao, S. Sukhbaatar, D. Su, X. Li, Z. Hu, J. Weston, and Y. Tian, *Training Large Language Models to Reason in a Continuous Latent Space*, Dec. 2024. DOI: 10.48550/arXiv.2412.06769 arXiv: 2412.06769 [cs]. Accessed: Jun. 27, 2025.
- [15] A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, et al., “Openai o1 system card,” *arXiv preprint arXiv:2412.16720*, 2024.
- [16] E. Jang, S. Gu, and B. Poole, “Categorical reparameterization with gumbel-softmax,” *arXiv preprint arXiv:1611.01144*, 2016.

- 
- [17] Ł. Kaiser and I. Sutskever, *Neural GPUs Learn Algorithms*, Mar. 2016. DOI: 10.48550/arXiv.1511.08228 arXiv: 1511.08228 [cs]. Accessed: Aug. 7, 2025.
  - [18] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” *ArXiv*, vol. abs/2205.11916, 2022. [Online]. Available: <https://api.semanticscholar.org/CorpusID:249017743>
  - [19] I. Loshchilov and F. Hutter. “Decoupled Weight Decay Regularization.” arXiv: 1711.05101 [cs.LG], Accessed: Aug. 3, 2026. [Online]. Available: <http://arxiv.org/abs/1711.05101> pre-published.
  - [20] C. J. Maddison, A. Mnih, and Y. W. Teh. “The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables.” arXiv: 1611.00712 [cs], Accessed: Jan. 5, 2026. [Online]. Available: <http://arxiv.org/abs/1611.00712> pre-published.
  - [21] W. Merrill and A. Sabharwal, “The Parallelism Tradeoff: Limitations of Log-Precision Transformers,” *Transactions of the Association for Computational Linguistics*, vol. 11, pp. 531–545, Jun. 12, 2023, ISSN: 2307-387X. DOI: 10.1162/tac1\_a\_00562 Accessed: Dec. 30, 2025. [Online]. Available: [https://direct.mit.edu/tac1/article/doi/10.1162/tac1\\_a\\_00562/116413/The-Parallelism-Tradeoff-Limitations-of-Log](https://direct.mit.edu/tac1/article/doi/10.1162/tac1_a_00562/116413/The-Parallelism-Tradeoff-Limitations-of-Log)
  - [22] C. Núñez-Molina, I. P. Merino, P. Mesejo, and J. Fernández-Olivares, “Learning to reason with transformers via search inductive biases: A proposal,” in *AAAI 2025 Workshop LM4Plan*, 2025.
  - [23] C. Núñez-Molina, P. Mesejo, and J. Fernández-Olivares, “A Review of Symbolic, Subsymbolic and Hybrid Methods for Sequential Decision Making,” *ACM Computing Surveys*, vol. 56, no. 11, pp. 1–36, Nov. 2024, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3663366 Accessed: Jul. 9, 2025.
  - [24] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, et al., “Show your work: Scratchpads for intermediate computation with language models,” *arXiv preprint arXiv:2112.00114*, 2021.
  - [25] J. Oh, S. Singh, and H. Lee, “Value prediction network,” in *Neural Information Processing Systems*, 2017. [Online]. Available: <https://api.semanticscholar.org/CorpusID:26464771>
  - [26] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia, *Learning model-based planning from scratch*, Jul. 2017. DOI: 10.48550/arXiv.1707.06170 arXiv: 1707.06170 [cs]. Accessed: Aug. 12, 2025.

- 
- [27] S. Racanière, T. Weber, D. Reichert, L. Buesing, A. Guez, D. Jimenez Rezende, A. Puigdomènech Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. Silver, and D. Wierstra, “Imagination-augmented agents for deep reinforcement learning,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2017/file/9e82757e9a1c12cb710ad680db11f6f1-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2017/file/9e82757e9a1c12cb710ad680db11f6f1-Paper.pdf)
  - [28] N. Saunshi, N. Dikkala, Z. Li, S. Kumar, and S. J. Reddi, *Reasoning with Latent Thoughts: On the Power of Looped Transformers*, Feb. 2025. DOI: 10.48550/arXiv.2502.17416 arXiv: 2502.17416 [cs]. Accessed: Jun. 27, 2025.
  - [29] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, “Mastering Atari, Go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 2020, ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4 Accessed: Aug. 8, 2025.
  - [30] Y. Sui, Y.-N. Chuang, G. Wang, J. Zhang, T. Zhang, J. Yuan, H. Liu, A. Wen, S. Zhong, H. Chen, and X. Hu, *Stop Overthinking: A Survey on Efficient Reasoning for Large Language Models*, Apr. 2025. DOI: 10.48550/arXiv.2503.16419 arXiv: 2503.16419 [cs]. Accessed: Jun. 23, 2025.
  - [31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf)
  - [32] O. Vinyals, M. Fortunato, and N. Jaitly. “Pointer Networks.” arXiv: 1506.03134 [stat.ML], Accessed: Jun. 22, 2026. [Online]. Available: <http://arxiv.org/abs/1506.03134> pre-published.
  - [33] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, Jan. 2023. DOI: 10.48550/arXiv.2201.11903 arXiv: 2201.11903 [cs]. Accessed: Sep. 5, 2025.
  - [34] L. Yang, K. Lee, R. Nowak, and D. Papailiopoulos, *Looped Transformers are Better at Learning Learning Algorithms*, Mar. 2024. DOI: 10.48550/arXiv.2311.12424 arXiv: 2311.12424 [cs]. Accessed: Jun. 27, 2025.
  - [35] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*, Dec. 2023. DOI: 10.48550/arXiv.2305.10601 arXiv: 2305.10601 [cs]. Accessed: Sep. 30, 2025.

- [36] B. Zhang and R. Sennrich, “Root mean square layer normalization,” *Advances in neural information processing systems*, vol. 32, 2019.
- [37] H. Zhu, S. Hao, Z. Hu, J. Jiao, S. Russell, and Y. Tian, “Reasoning by superposition: A theoretical perspective on chain of continuous thought,” *ArXiv*, vol. abs/2505.12514, 2025. [Online]. Available: <https://api.semanticscholar.org/CorpusID:278740606>