DEEPDP: AUTOMATED SYNTHESIS OF RECURRENCES FOR DYNAMIC PROGRAMMING PROBLEMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Dynamic programming (DP) solves many problems efficiently once an appropriate recurrence is known, but discovering that recurrence is often the key algorithmic step. We ask whether this process can be reversed: can small DP tables, constructed by solving each state independently through exhaustive search, provide enough evidence to recover a recurrence that scales to larger instances? We present DEEPDP, which learns a structural prior from synthetic executable rules, represents recurrences through typed DP components, and adapts to new tasks using only table-reconstruction feedback, without target recurrence labels. Across 26 held-out tasks spanning four DP geometries, with target recurrence behaviors excluded from the training rule banks, DEEPDP achieves full functional generalization and improves mean accuracy by up to 49.11 percentage points over program-synthesis and symbolic baselines in the low-information regime. These results show that small exact computations can support the discovery of interpretable, executable algorithmic structure.

1 INTRODUCTION

Dynamic programming (DP) turns many combinatorial problems into efficient algorithms—but only after the right recurrence has been identified. The recurrence is the reusable rule that specifies how each state is computed from smaller states, input values, and boundary conditions; a completed DP table, by contrast, records one solved instance. This raises a natural reverse question: *can we discover the recurrence itself from small, exactly solved DP tables?*

Small tables can often be constructed without knowing any recurrence. Consider 0/1 knapsack as a simple illustration. For a small number of items, each entry $D[i, j]$ can be computed independently by enumerating subsets of the first i items and selecting the best one that fits capacity j. This yields an exact table, but the cost grows rapidly with problem size. If we instead infer the familiar recurrence $D[i, j] = \max(D[i-1, j], D[i-1, j-w_i] + v_i)$, we can then use it directly to fill larger DP tables efficiently, avoiding exhaustive search at every state. For textbook knapsack the recurrence is already known; we use it only to make the recovery problem concrete. Our target setting is one in which small instances can be solved exactly, but the recurrence relation is not known a priori.

Viewed this way, recurrence recovery is a form of *algorithm discovery*: rather than predicting answers, the goal is to infer an explicit executable rule from exact observations—one that explains those observations, can be inspected by humans, and can be executed independently of the learning system once discovered. This provides a controlled setting for AI-driven discovery in mathematics and computer science, with the potential to derive scalable algorithms for new problem variants when exact small-scale computation is available but the recurrence is unknown.

The challenge is that small tables rarely determine a unique recurrence. Many rules may fit the observed entries while behaving differently on unseen inputs, and a valid recurrence may involve predecessor states, input-dependent shifts or costs, aggregation operators, boundaries, branches, or interval splits. Searching over arbitrary programs is therefore enormous and poorly matched to DP structure. Our key observation is that diverse DP algorithms can instead be expressed through a compact set of typed Bellman-style components constrained by the geometry of the state space.

We build on this observation with DEEPDP, a framework for synthesizing executable DP recurrences from small solved tables. DEEPDP learns a structural prior from a synthetic bank of executable recurrences spanning temporal chains, Cartesian state lattices, symbolic alignment lattices, and interval-decomposition lattices. For a new task, it predicts typed recurrence components, assembles

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

them into an executable rule, and tests the rule in a DP simulator. Adaptation requires no target recurrence labels: candidate rules are optimized only by how well their execution reconstructs solved tables. Thus, DP structure constrains the search, while execution provides a direct semantic test of each proposal.

This differs from both code generation and neural algorithmic reasoning. Code-generation and program-synthesis systems produce programs from prompts or examples (Chen et al., 2021; Li et al., 2022; Roziere et al., 2023; Li et al., 2023; Guo et al., 2024), whereas neural algorithmic reasoners typically learn to execute an algorithm whose structure is already fixed (Veličković et al., 2019; 2022; Dudzik & Veličković, 2022; Dudzik et al., 2024). Here, the algorithmic rule itself is hidden. In an illustrative GPT-4o probe, even three solved demonstrations often led to familiar DP templates rather than the precise executable rule supported by the observations.

Contributions. We make three contributions. First, we formulate recurrence recovery from solved DP tables as an interpretable algorithm-discovery problem. Second, we introduce DEEPDP, combining a typed recurrence representation, a learned structural prior from synthetic executable rules, and execution-guided reward-only adaptation. Third, we evaluate 26 held-out tasks across four DP geometries, with target recurrence behaviors excluded from the training rule banks. DEEPDP achieves full held-out functional generalization across all tasks, including behaviorally equivalent recurrences that differ syntactically from the reference, and improves mean accuracy by up to 49.11 percentage points in the low-information regime.

2 BACKGROUND AND PROBLEM FORMULATION

We consider tasks for which exact small-instance DP tables are available but the recurrence is unknown. The objective is to recover an executable rule that explains these observations and generalizes to unseen inputs; when multiple rules are consistent with limited evidence, their executed table behavior provides the criterion for distinguishing them.

2.1 DP RECURRENCES AS EXECUTABLE RULES

A DP algorithm can be written as a Bellman-style update over table states (Bellman, 1966). Let x be the input instance, D the DP table, and s the current state. A recurrence computes $D[s]$ by reading predecessor states, optionally adding input-dependent local costs, and combining the resulting candidates:

$$D[s] = B_r\left(\{D[\tau_{r,m}(s,x)] + c_{r,m}(s,x)\}_{m\in\mathcal{I}_r(s,x)}\right). \tag{1}$$

Here $\tau_{r,m}$ selects a predecessor state, $c_{r,m}$ denotes an optional local cost or bias, $\mathcal{I}_r(s,x)$ is the active candidate set at state s, and B_r is a Bellman combiner such as $\max$, $\min$, $+$, or a logical operator. The recurrence also specifies boundary initialization and invalid-read behavior, which determine how the table is initialized and how out-of-range predecessor accesses are handled.

2.2 STATE-SPACE GEOMETRIES OF DP RECURRENCES

We consider four representative DP geometries that cover common dynamic-programming structures and determine the state space, predecessor patterns, and execution order.

Unary temporal chains. These problems use a one-dimensional state over position or time, with local predecessors such as $i-1$ or $i-2$ and optional local costs; examples include House Robber, Delete and Earn, and Min Cost Climbing Stairs (LeetCode, 2026).

Cartesian state lattices. These problems use a two-dimensional table $D[i,j]$, where one axis tracks progress and the other tracks a resource, target, or spatial coordinate; their recurrences use axis-aligned predecessors and input-dependent shifts such as $j-w_i$. Examples include Knapsack, Coin Change, Subset Sum, and Target Sum (Cormen et al., 2022; Kleinberg & Tardos, 2006; Wang et al., 2025).

Symbolic alignment lattices. These problems use a two-dimensional table over two symbolic sequences, often with equality-gated diagonal transitions such as using $D[i-1,j-1]$ only when

two symbols match; examples include Longest Common Subsequence, Longest Common Substring, and Distinct Subsequences (Hirschberg, 1975; Zhu et al., 2017).

Interval-decomposition lattices. These problems define states over intervals, usually by length and start position, and aggregate over split points; examples include Burst Balloons, Minimum-Cost Interval Cutting, and the coupled state tables in Zuker RNA folding (Cormen et al., 2022; Lei et al., 2012). Their non-row-column execution order requires a length–start–split grammar, which can also capture dependencies among multiple coupled interval tables.

2.3 Problem Formulation: Recurrence Recovery

For an unknown DP task f, the learner receives a *support set* of completed small-instance tables and lightweight metadata, and predicts an executable recurrence:

$$(\mathcal{S}_f, \delta_f) \longmapsto \hat{r}_f. \tag{2}$$

The support set $\mathcal{S}_f = \{(x_k, D_f(x_k))\}_{k=1}^{K}$ contains K solved examples, where $x_k \in \mathcal{X}_f$ and $D_f(x_k)$ is the completed DP table. These tables may be generated by brute-force enumeration or another solver for small instances; they reveal the completed computation but not the recurrence itself. The metadata $\delta_f = (G_f, b_f, q_f, T_f)$ provides coarse task information: DP geometry, symbolic-feature flag, input-feature dimension, and terminal readout convention. Thus, solved tables provide behavioral evidence, while metadata constrains the search to recurrences compatible with the task geometry.

We represent $\hat{r}_f$ using typed *rule components*, following program synthesis by searching a constrained language rather than unconstrained text (Solar-Lezama, 2008; Balog et al., 2016; Ellis et al., 2021). Components specify predecessor locations, input-dependent offsets, local costs, aggregation operators, boundary and invalid-read policies, branch conditions, and interval split options. DeepDP therefore performs recurrence recovery within this typed DP language, conditioned on coarse state-space geometry, while inferring the recurrence structure directly from solved tables.

For a held-out input $x' \sim \mathcal{X}_f$, the inferred recurrence should reconstruct the corresponding DP table:

$$\hat{D}(x') = \mathrm{Exec}(\hat{r}_f, x') \approx D_f(x'), \tag{3}$$

where $\mathrm{Exec}(\hat{r}_f, x')$ executes the predicted recurrence using the simulator for geometry G_f. The learner must generalize beyond both the observed support examples and the recurrence functions seen during learner construction.

2.4 Example: 0/1 Knapsack

We instantiate the recurrence-recovery formulation with 0/1 knapsack. Each task instance is $x = (\{(w_i, v_i)\}_{i=1}^{n}, C)$, where w_i and v_i are the weight and value of item i, and C is the capacity. The goal is to choose a subset of items that maximizes total value while keeping total weight at most C. The task uses a Cartesian state lattice, so its metadata is $\delta_f = (\textsc{Cartesian}, 0, 2, \textsc{LastCell})$: no symbolic features, two numeric item features, and a last-cell terminal readout.

The support set $\mathcal{S}_f$ contains completed tables $D_f(x_k)$, where $D_f(x_k)[i, j]$ stores the maximum value obtainable using the first i items under capacity j. These tables may be generated by brute-force enumeration for small n and C, but the recurrence itself is hidden. One recurrence consistent with these tables is

$$D[i, j] = \max\left(D[i-1, j], D[i-1, j-w_i] + v_i\right), \tag{4}$$

with invalid access when $j - w_i < 0$ and zero initialization along the boundary.

In the typed rule-component representation, this recurrence contains two predecessor locations, an input-dependent capacity shift by w_i, a local value bias v_i, a $\max$ aggregation operator, zero boundary initialization, and invalid-state handling for $j - w_i < 0$. The recovery goal is to infer these components from the support tables. Executing the recovered $\hat{r}_f$ on held-out knapsack instances should accurately reconstruct their full DP tables.

3 DeepDP: Dynamic Programming Recurrence Synthesis

Building on the formulation in Sec. 2.3, DeepDP operationalizes recurrence synthesis through the pipeline in Figure 1. A support-table encoder summarizes $\mathcal{S}_f$, typed decoders predict recurrence

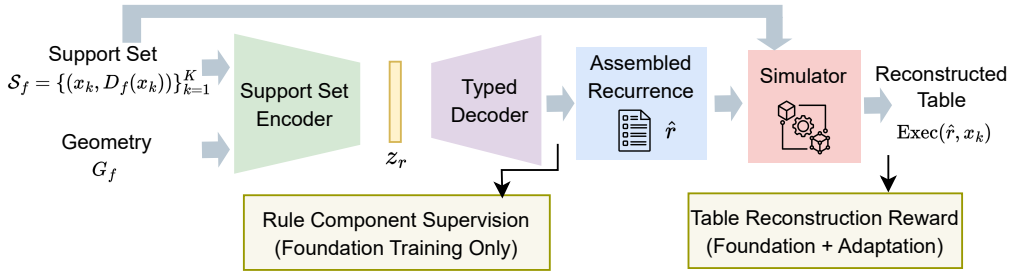

Figure 1: DEEPDP overview: support tables are encoded into a latent recurrence signature, decoded into typed recurrence components, and executed by a simulator for training and adaptation.

components, and a simulator executes the assembled recurrence to measure reconstruction. DEEPDP is first trained as a *foundation recurrence model* on a synthetic bank of executable rules, using both rule-component supervision and execution-based table-reconstruction reward; the resulting foundation model is then adapted to a new task using only solved support tables and reconstruction reward, without target recurrence labels.

3.1 SYNTHETIC RULE BANK CONSTRUCTION

Foundation training requires many recurrences with known rule components, analogous to generating synthetic specifications or examples for neural program synthesis (Devlin et al., 2017; Bunel et al., 2018; Chen et al., 2018). Since real tasks rarely provide labeled recurrence decompositions, DEEPDP constructs a synthetic rule bank $\mathcal{R}$ for each supported DP geometry by enumerating the typed component space and applying geometry-specific validity constraints. Importantly, $\mathcal{R}$ is not a collection of textbook DP problems; it contains arbitrary executable recurrence mechanisms sampled from this component space. The validity constraints remove cyclic dependencies, unsupported boundary or branch configurations, and predecessor patterns that cannot be executed under the chosen geometry. Each retained rule is executable by the simulator and provides both solved DP tables and component labels for training.

Different component choices can sometimes induce the same DP computation, for example due to symmetric predecessors, inactive components, or equivalent boundary behavior. Keeping all such variants would train the model on redundant syntactic labels rather than distinct recurrence behaviors. We therefore filter the bank behaviorally: each candidate rule r is executed on a fixed probe set $\mathcal{P}_{\text{probe}} = \{x_m\}_{m=1}^M$, producing the fingerprint

$$\phi(r) = (\mathrm{Exec}(r, x_1), \mathrm{Exec}(r, x_2), \ldots, \mathrm{Exec}(r, x_M)) \,. \tag{5}$$

The rule bank keeps one representative per fingerprint using diverse probe instances for each geometry. This compression removes redundant labels while preserving broad recurrence coverage.

3.2 SUPPORT-SET ENCODER AND TYPED DECODERS

Given a support set $\mathcal{S}_f = \{(x_k, D_f(x_k))\}_{k=1}^K$, DEEPDP infers a latent signature of the unknown recurrence from the completed DP tables. The goal is not only to encode table values, but to identify the dependency pattern that could have generated them. We view each table as an ordered collection of states, where a state s denotes a table entry under geometry G_f, such as a row–column cell or an interval state. For each support example, the encoder forms a state feature from the observed table value and the input feature aligned to that state:

$$a_{k,s} = \left[D_f(x_k)[s],\ \eta_{G_f}(x_k, s)\right], \qquad u_{k,s} = \mathrm{EncFeat}_{G_f}(a_{k,s}, s). \tag{6}$$

Here $\eta_{G_f}(x_k, s)$ is the input-derived feature available at state s, such as an aligned numeric item feature, a symbolic equality indicator, or an interval-boundary value. The function $\mathrm{EncFeat}_{G_f}$ constructs geometry-specific encoder features using feature projections and positional embeddings when needed. The encoder uses a causal attention mask from the DP execution order, building on attention-based sequence modeling while imposing an algorithm-specific computation order (Vaswani et al., 2017; Veličković et al., 2019). Rather than treating a table as an unordered image, the Transformer follows the corresponding DP geometry's prefix structure:

$$H_k = \mathrm{CausalTr}_\psi\left(\{u_{k,s}\}_{s\in\Omega_k}; \mathcal{M}_{\preceq_{G_f}}\right). \tag{7}$$

Here Ω_k is the state set for support instance k, and $\mathcal{M}_{\preceq_{G_f}}$ allows each state to attend only to states not later in the execution order $\preceq_{G_f}$. This gives the encoder access to completed support tables while preserving the computational direction that the decoded recurrence must follow during execution.

After encoding each support instance, DEEPDP aggregates table-level representations across K examples into a latent recurrence vector z_r. Conditioned on z_r, typed decoders predict rule-component choices for geometry G_f. For each component type $t \in \mathcal{T}_{G_f}$, the decoder produces $p_{\theta,t}(y_t \mid \mathcal{S}_f) = \mathrm{softmax}(W_t z_r + b_t)$, where $y_t \in \mathcal{O}_t$ is a discrete choice for component type t. The predicted choices are assembled into an executable recurrence, and the decoder distributions define the policy used for training and adaptation.

3.3 EXECUTION-GUIDED FOUNDATION TRAINING

Foundation training uses two signals: supervised rule-component prediction and execution-based table reconstruction. For a synthetic training task f with underlying recurrence r_f, let $S_f = \{(x_k, D_f(x_k))\}_{k=1}^{K}$ denote its generated support set. Only semantically active components are supervised. Let $\mathcal{A}(r_f)$ denote this active component set. The supervised loss is

$$\mathcal{L}_{\mathrm{sup}} = \sum_{t \in \mathcal{A}(r_f)} \mathrm{CE}(p_{\theta,t}(\cdot \mid \mathcal{S}_f), y_t)\,, \tag{8}$$

where $p_{\theta,t}$ is the predicted distribution for component type t, and y_t is its ground-truth label.

Component labels teach local recurrence structure, but they do not guarantee that the assembled rule computes the correct table. For executable correctness, a recurrence $\hat{r}$ is sampled from $p_\theta(\cdot \mid \mathcal{S}_f)$, executed on K support inputs, and scored by exact table reconstruction:

$$R_{\mathrm{exec}}(\hat{r}; \mathcal{S}_f) = \frac{1}{K} \sum_{k=1}^{K} \mathbb{1}\left[\mathrm{Exec}(\hat{r}, x_k) = D_f(x_k)\right]. \tag{9}$$

The recurrence probability $p_\theta(\hat{r} \mid \mathcal{S}_f)$ is the product of the decoder probabilities of the component choices used to assemble $\hat{r}$.

The recurrence-selection policy is optimized with a policy-gradient objective (Williams, 1992). We subtract a mini-batch reward baseline $\bar{R}$, computed as the average reconstruction reward of sampled recurrences in the current mini-batch, to reduce variance without changing the expected policy-gradient direction. The full foundation-training objective is

$$\begin{aligned} \mathcal{L}_{\mathrm{RL}} &= -\mathbb{E}_{\hat{r} \sim p_\theta(\cdot \mid \mathcal{S}_f)}\left[\log p_\theta(\hat{r} \mid \mathcal{S}_f)\big(R_{\mathrm{exec}}(\hat{r}; \mathcal{S}_f) - \bar{R}\big)\right], \\ \mathcal{L}_{\mathrm{train}} &= \lambda_{\mathrm{sup}} \mathcal{L}_{\mathrm{sup}} + \lambda_{\mathrm{RL}} \mathcal{L}_{\mathrm{RL}} + \lambda_{\mathrm{ent}} \mathcal{H}(p_\theta). \end{aligned} \tag{10}$$

Here λ_{sup}, λ_{RL}, and λ_{ent} weight the supervised, reconstruction-reward, and entropy terms, respectively; the entropy penalty encourages confident rule-component predictions.

3.4 REWARD-ONLY ADAPTATION

For a new target task, DEEPDP adapts a copy of the foundation recurrence model using only support tables; no target recurrence labels are used. Because DEEPDP forms a task representation by aggregating information across support instances, adaptation is not tied to the support-set size used during foundation training and can operate with fewer solved examples. For geometry G_f, a sampled recurrence $\hat{r}$ is executed by the simulator and scored by reconstruction reward, $R_{\mathrm{adapt}}(\hat{r}; \mathcal{S}_f) = R_{\mathrm{exec}}^{G_f}(\hat{r}; \mathcal{S}_f)$, where $R_{\mathrm{exec}}^{G_f}$ is Equation 9 evaluated under geometry G_f. The adaptation loss is

$$\mathcal{L}_{\mathrm{adapt}} = -\log p_\theta(\hat{r} \mid \mathcal{S}_f)\big(R_{\mathrm{adapt}}(\hat{r}; \mathcal{S}_f) - \bar{R}\big) - \lambda_{\mathrm{adapt-ent}} \mathcal{H}(p_\theta). \tag{11}$$

Because the reward is sparse and the rule space is discrete, DEEPDP uses a small self-distillation mechanism during adaptation, complementing entropy-based exploration for sampled discrete decisions (Williams, 1992; Jang et al., 2016; Maddison et al., 2016). Recurrences with high reconstruction reward are stored in an elite buffer, and their decoded components are reused as temporary teacher targets:

$$\mathcal{L}_{\mathrm{elite}} = \sum_{t \in \mathcal{A}(\hat{r}_{\mathrm{elite}})} \mathrm{CE}\big(p_{\theta,t}(\cdot \mid \mathcal{S}_f), \hat{y}_t^{\mathrm{elite}}\big)\,, \qquad \mathcal{L}_{\mathrm{adapt}}^{+} = \mathcal{L}_{\mathrm{adapt}} + \lambda_{\mathrm{elite}} \mathcal{L}_{\mathrm{elite}}. \tag{12}$$

Table 1: Benchmark taxonomy for recurrence-recovery evaluation.

Geometry	State topology	Recurrence features	# test tables
Unary temporal chain	1D sequence	local dependencies; value-indexed shifts; min/max-plus updates	4,500
Cartesian state lattice	spatial/resource grid	axis shifts; boundary policies; counting, feasibility, and optimization	13,500
Symbolic alignment lattice	sequence grid	equality-gated diagonal transitions; symbolic branching; counting and alignment	16,500
Interval-decomposition lattice	interval and coupled states	split recurrences; interval costs; min/max aggregation; coupled subproblem decomposition	4,500

This stabilizes adaptation by retaining high-reward choices while optimizing reconstruction. Adaptation is a task-level, reward-only specialization of the learned recurrence prior: solved target-task tables refine the search distribution, while recurrence recovery remains conditioned on a single support table without symbolic rule supervision.

4 EVALUATION

4.1 BENCHMARK, MODEL, AND METRICS

We evaluate DEEPDP using the training configuration detailed in Appendix G. During foundation training, the support-table encoder maps four support examples to a normalized latent rule vector ($K = 4$), from which the decoder predicts active rule components. We train geometry-specific foundation models spanning unary temporal chains, Cartesian state lattices, symbolic alignment lattices, and interval-decomposition lattices. Across the foundation models used in our evaluation, our full-scale configuration collectively uses over 800,000 synthetic training examples spanning 14,398 distinct recurrence rules. To test rule-level generalization, all 26 held-out tasks use target recurrences whose executed behavior is excluded from the corresponding synthetic rule bank. During adaptation, each episode contains a single solved support example ($K = 1$) sampled from the task-specific adaptation split. Thus, adaptation can accumulate task-level evidence across optimization episodes, while each episode—and each recurrence-recovery attempt—observes only one completed table. Inference likewise conditions on a single support example per attempt. At no point during adaptation or inference are the reference recurrence or its component labels provided. Each task contains 3,000 task-specific examples, split into 900 training, 600 validation, and 1,500 held-out test examples, yielding 39,000 held-out test examples across the 26 tasks. We further compare DEEPDP against methodologically close neural-symbolic and symbolic program-synthesis baselines—DeepCoder (Balog et al., 2016), DreamCoder (Ellis et al., 2021), and Z3 (De Moura & Bjørner, 2008)—that infer executable symbolic programs from observed examples.

We report two rule-recovery metrics. *Canonical rule agreement* measures exact agreement between the recovered recurrence and the reference recurrence over active rule components. *Generalization accuracy* is the fraction of independent inference attempts for which DEEPDP, given a single support instance, recovers a recurrence that exactly reconstructs all DP tables in the task's full held-out test set. An attempt succeeds only if the recovered recurrence exactly reconstructs every valid state of every test table; a single erroneous state makes the attempt unsuccessful. Detailed metric definitions are provided in Appendix E.

Table 1 summarizes the 26 held-out recurrence-recovery tasks by DP geometry. The benchmark spans combinatorial optimization, counting, sequence alignment, interval decomposition, and structured prediction, including unary chains, Cartesian state lattices, symbolic alignment lattices, and interval decompositions (LeetCode, 2026; Wang et al., 2025; Bellman, 1966; Cormen et al., 2022; Kleinberg & Tardos, 2006; Hirschberg, 1975; Zhu et al., 2017; Guenoche, 2004; Lei et al., 2012). These geometries test whether DEEPDP can recover recurrence structure across diverse state spaces and input regimes.

4.2 HELD-OUT RECURRENCE RECOVERY

Table 2 reports held-out recurrence recovery following foundation training and task-specific adaptation. We evaluate each recovered recurrence using canonical rule agreement, which measures exact correspondence to the reference rule components, and generalization accuracy, which measures whether the recovered recurrence correctly reconstructs all DP tables in the full held-out test set.

Table 2: Held-out recurrence recovery using DEEPDP. Canonical agreement measures exact recovery of rule components, and generalization accuracy measures reconstruction on the full held-out test set. ✓denotes 100%.

Geometry	Task	Canonical rule agreement	Generalization accuracy
Unary temporal chain	House Robber	✓	✓
	Delete and Earn	✓	✓
	Min Cost Climbing Stairs	99.33%	✓
Cartesian state lattice	Unique Paths	✓	✓
	Perfect Squares	✓	✓
	Coin Change	✓	✓
	Partition Equal Subset Sum	0.00%	✓
	Target Sum	✓	✓
	Coin Change II	✓	✓
	Last Stone Weight II	0.00%	✓
	Ways to Express Integer	✓	✓
	Sparse Figurate Decomposition	✓	✓
Symbolic alignment lattice	Distinct Subsequences	0.00%	✓
	Is Subsequence	30.00%	✓
	Delete Operation for Two Strings	✓	✓
	Maximum Length of Repeated Subarray	✓	✓
	Uncrossed Lines	30.67%	✓
	Shortest Common Supersequence	✓	✓
	Longest Common Subsequence	50.67%	✓
	Single-Path Subsequence Counting	0.00%	✓
	Is-Subsequence Alignment	94.00%	✓
	Short-Run Concordance	✓	✓
	Sparse Vocabulary Alignment	74.67%	✓
Interval-decomposition lattice	Burst Balloons	6.67%	✓
	Minimum Cost to Cut a Stick	54.67%	✓
	Zuker RNA Folding	0.00%	✓

Table 3: Generalization accuracy in the low-information regime across all baselines. ✓denotes 100%.

Task	DeepCoder	DreamCoder	Z3	DEEPDP
Sparse Figurate Decomposition	0.00%	1.33%	56.00%	✓
Single-Path Subsequence Counting	66.00%	97.33%	84.67%	✓
Is-Subsequence Alignment	56.67%	79.33%	89.33%	✓
Short-Run Concordance	32.67%	32.00%	96.00%	✓
Sparse Vocabulary Alignment	82.00%	92.00%	28.00%	✓
Zuker RNA Folding	68.00%	99.33%	10.00%	✓

DEEPDP achieves full held-out generalization across all 26 tasks and all four DP geometries. Importantly, this performance does not require exact syntactic reproduction of a canonical rule: multiple recurrence formulations can induce the same computation over the evaluated task domain. For example, several tasks achieve full generalization despite canonical agreement below 50%, demonstrating that recovered rules can reconstruct every held-out DP table correctly even when their components differ substantially from the reference formulation.

For example, in Partition Equal Subset Sum, DEEPDP achieves full generalization despite 0.00% canonical agreement. The reference recurrence combines Boolean predecessors using `or`, whereas a recovered rule may use `max`. Because the DP states are Boolean-valued, these operators induce identical updates. Similar behavioral equivalences arise in sequence-alignment and interval-decomposition tasks. These results demonstrate that DEEPDP can recover functionally correct recurrences without being restricted to a single canonical representation. Further discussion is provided in Section 4.5.

4.3 RECURRENCE RECOVERY IN THE LOW-INFORMATION REGIME

We next isolate the *low-information regime*, where the observed input–table pair reveals only a limited set of informative transitions and therefore provides a particularly stringent test of recurrence recovery. These tasks span Cartesian, symbolic-alignment, and interval-decomposition geometries, with different recurrence mechanisms giving rise to multiple candidates that can be compatible with the available evidence. This setting directly tests whether a method can use learned structural knowledge to identify recurrences that generalize beyond the observed computation. Table 3 compares DEEPDP with DeepCoder, DreamCoder, and Z3 on these low-information tasks. The baselines achieve strong recovery on individual tasks, with mean generalization accuracies of 50.89%, 66.89%, and 60.67%, respectively. DEEPDP, however, achieves full generalization across all evaluated

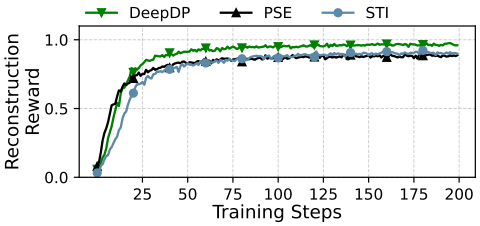

Figure 2: Validation reward in foundation training.

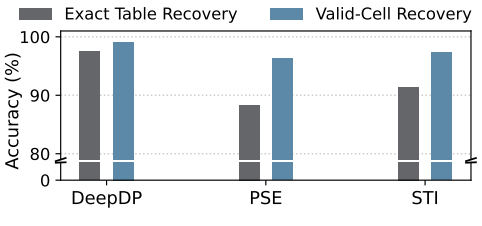

Figure 3: Ablation of the foundation architecture.

Table 4: Ablation of adaptation and self-distillation across all held-out recurrence tasks.

Geometry	w/o adaptation	w/o self-distillation	Full DEEPDP	Best Task-Level Adaptation Gain (pp)
Unary temporal chain	18.83%	99.78%	✓	+83.03
Cartesian state lattice	48.28%	88.89%	✓	+100.00
Symbolic alignment lattice	66.88%	✓	✓	+57.32
Interval-decomposition lattice	0.51%	33.11%	✓	+100.00

low-information tasks, improving mean accuracy by 49.11, 33.11, and 39.33 percentage points over the three baselines. These results demonstrate that the learned structural prior enables DEEPDP to recover generalizable recurrence structure consistently even when the observed support evidence is limited. Appendix D further analyzes how such a prior improves recurrence recovery.

4.4 DISSECTING THE RECURRENCE-RECOVERY PIPELINE

We first ablate the foundation-side design using two alternatives. **Pointwise Support Encoder (PSE)** replaces the DP-causal Transformer with a PointNet-style encoder (Qi et al., 2017), independently encoding support-table cells with a shared MLP before pooling. **Single-Trace Inference (STI)** retains the DEEPDP architecture but restricts foundation-side inference to a single solved support instance, testing the benefit of aggregating multiple support traces (Weir et al., 2022). On Cartesian state lattice tasks, Figures 2 and 3 show that the full foundation design converges faster and achieves stronger held-out reconstruction; DEEPDP reaches 97.54% exact-table recovery and 99.15% valid-cell recovery, demonstrating that its faster convergence translates into more accurate recovery of held-out recurrences.

We next ablate task-specific adaptation and self-distillation across all 26 held-out tasks (Table 4). Task-specific adaptation provides substantial gains across all four geometries, with best task-level improvements ranging from 57.32 to 100 percentage points. Self-distillation provides particularly strong gains for the more challenging Cartesian and interval-decomposition geometries, improving generalization by 11.11 and 66.89 percentage points, respectively. Together, these results show how the learned structural prior, adaptation, and self-distillation complement one another for robust task-specific recurrence recovery. Appendix C further varies synthetic training size and recurrence-rule coverage; recovery improves consistently with both, showing that performance is not tied to an arbitrarily large pretraining configuration.

4.5 QUALITATIVE ANALYSIS

DEEPDP can recover functionally correct recurrences without requiring exact syntactic agreement with a canonical rule, because the same DP computation can admit multiple recurrence forms; similar semantic-equivalence issues are central in program synthesis and analysis (Manna & Waldinger, 1971; Gulwani, 2011; Solar-Lezama, 2008). For example, Boolean-valued tables may make `or` and `max` equivalent, duplicate predecessors may make the aggregator choice irrelevant, and inactive boundary or branch components may differ without affecting valid table entries.

Figure 4 shows an example from UNCROSSED LINES. For the canonical longest-common-subsequence recurrence, when the two symbols match, the diagonal-plus-one candidate attains the same optimum as `max(base,c)`. The two formulations therefore preserve the same computation despite differing syntactically, illustrating how DEEPDP can recover behaviorally equivalent recurrences beyond a single canonical representation.

More broadly, this analysis highlights the advantage of DEEPDP's execution-guided recovery: it need not reproduce a single canonical symbolic formulation. Rather, consistent with our formulation of

Canonical rule

```
# Init: dp_{0,j} ← 0, dp_{i,0} ← 0
for i ∈ 1...N:
  for j ∈ 1...M:
    base ← max(dp_{i,j-1}, dp_{i-1,j})
    c ← dp_{i-1,j-1} + 1
    if v1_{i-1} == v2_{j-1}:
      dp_{i,j} ← c              # <--- DIFFERENT
    else:
      dp_{i,j} ← base
```

Functionally equivalent rule

```
# Init: dp_{0,j} ← 0, dp_{i,0} ← 0
for i ∈ 1...N:
  for j ∈ 1...M:
    base ← max(dp_{i,j-1}, dp_{i-1,j})
    c ← dp_{i-1,j-1} + 1
    if v1_{i-1} == v2_{j-1}:
      dp_{i,j} ← max(base, c) # <--- DIFFERENT
    else:
      dp_{i,j} ← base
```

Figure 4: A functionally equivalent recurrence recovered for UNCROSSED LINES. The recovered rule uses an alternative true-branch formulation while preserving the same computation as the canonical rule.

recurrence recovery, the objective is to identify an explicit executable rule that explains the observed computation and generalizes to unseen inputs. By evaluating candidate recurrences through their executed table behavior, DEEPDP can therefore recover syntactically different formulations that implement the same underlying computation. This makes recurrence recovery robust to representational non-uniqueness while preserving the interpretability and executability of the recovered rule.

5 RELATED WORK

Neural algorithmic reasoning and algorithmic alignment. Neural algorithmic reasoning studies whether neural networks can learn algorithmic computation and generalize beyond training (Reed & De Freitas, 2015; Graves et al., 2016; Kaiser & Bengio, 2016). Benchmarks such as CLRS provide supervised traces for textbook algorithms, while neural execution work trains models to imitate known graph, sorting, string, and dynamic-programming procedures (Veličković et al., 2019; 2020; 2022; Numeroso et al., 2023; Li et al., 2024). Related algorithmic-alignment work analyzes when neural architectures match algorithmic structure, including links between message passing and dynamic programming (Xu et al., 2019; Deac et al., 2021; Bieber et al., 2020; Dudzik & Veličković, 2022; Dudzik et al., 2024). These works mainly test execution or extrapolation of a known algorithm. DEEPDP targets a different setting: the recurrence is unknown and must be inferred from solved DP tables before execution on held-out inputs.

Program synthesis, execution feedback, and code generation. Neural program-synthesis systems such as DeepCoder and DreamCoder recover executable programs from examples or specifications (Balog et al., 2016; Ellis et al., 2021), while modern code-generation models extend this idea to natural-language prompts and large programming corpora (Chen et al., 2021; Li et al., 2022; Feng et al., 2020; Ahmad et al., 2021; Li et al., 2023; Guo et al., 2024). These systems usually synthesize complete programs satisfying examples or unit tests (Austin et al., 2021; Chen et al., 2018). For DP, this only indirectly supervises the recurrence because the Bellman update is entangled with loops, indexing, and output handling. DEEPDP instead uses full DP tables as supervision, directly predicts typed recurrence components, and evaluates the induced computation by table reconstruction.

Declarative dynamic-programming systems. Declarative DP languages and systems, such as Dyna and Bellman's GAP, help users specify, compile, and optimize dynamic programs (Eisner et al., 2004; 2005; Sauthoff et al., 2013; Vieira et al., 2017). Other work develops algebraic or declarative views of DP and analyzes written DP programs to derive more efficient variants (Giegerich et al., 2004; Vieira, 2026). These systems assume that a recurrence or declarative program has already been provided. DEEPDP is complementary: it attempts to recover such a recurrence from solved tables before any DP program has been written.

6 CONCLUSION

We present DEEPDP, a framework for recovering executable DP recurrences from small solved tables. Rather than predicting final answers or generating unconstrained code, DEEPDP learns a structural prior over typed recurrence components from synthetic executable rules and adapts to new tasks using only table-reconstruction reward. Across 26 held-out tasks spanning four DP geometries, DEEPDP achieves full held-out generalization, including cases where the recovered rule differs syntactically from the canonical formulation. These results show that execution-guided recurrence recovery can transform small brute-force-completed tables into reusable and interpretable DP rules while remaining robust to representational non-uniqueness.

AI USE STATEMENT

We used generative AI tools during manuscript preparation for language editing, including proofreading, rewording, and improving presentation and clarity. These tools also provided occasional support for code modification and debugging. The authors examined and revised AI-assisted writing and validated AI-assisted code through review and testing. All reported results, analyses, and conclusions were independently checked by the authors, who remain fully responsible for the contents of this paper.

ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. The research does not involve private, sensitive, or personally identifiable data. Given the scope and methodology of the study, we are not aware of specific ethical issues or significant potential for harmful societal impact.

REPRODUCIBILITY STATEMENT

We include the DEEPDP implementation and experimental scripts in the supplementary materials as a ZIP archive. The accompanying materials and Appendix provide the hyperparameter configurations, instructions for running the code, and experimental details needed to replicate the experiments.

REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 2655–2668, 2021.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Richard Bellman. Dynamic programming. *science*, 153(3731):34–37, 1966.

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems*, 33:8626–8637, 2020.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Andreea-Ioana Deac, Petar Veličković, Ognjen Milinkovic, Pierre-Luc Bacon, Jian Tang, and Mladen Nikolic. Neural algorithmic reasoners are implicit planners. *Advances in Neural Information Processing Systems*, 34:15529–15542, 2021.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Andrew J Dudzik and Petar Veličković. Graph neural networks are dynamic programmers. *Advances in neural information processing systems*, 35:20635–20647, 2022.

Andrew Joseph Dudzik, Tamara von Glehn, Razvan Pascanu, and Petar Veličković. Asynchronous algorithmic alignment with cocycles. In *Learning on Graphs Conference*, pp. 3–1. PMLR, 2024.

Jason Eisner, Eric Goldlust, and Noah A Smith. Dyna: A language for weighted dynamic programming. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pp. 218–221, 2004.

Jason Eisner, Eric Goldlust, and Noah A Smith. Compiling comp ling: Weighted dynamic programming and the dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pp. 281–290, 2005.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Robert Giegerich, Carsten Meyer, and Peter Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476, 2016.

Alain Guenoche. Supersequences of masks for oligo-chips. *J. Bioinform. Comput. Biol.*, 2(3): 459–470, 2004.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Łukasz Kaiser and Samy Bengio. Can active memory replace attention? *Advances in Neural Information Processing Systems*, 29, 2016.

Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

LeetCode. Leetcode. `https://leetcode.com/`, 2026.

Guoqing Lei, Yong Dou, Wen Wan, Fei Xia, Rongchun Li, Meng Ma, and Dan Zou. Cpu-gpu hybrid accelerating the zuker algorithm for rna secondary structure prediction applications. *BMC genomics*, 13(Suppl 1):S14, 2012.

Hefei Li, Chao Peng, Chenyang Xu, and Zhengfeng Yang. Open-book neural algorithmic reasoning. *Advances in Neural Information Processing Systems*, 37:9819–9836, 2024.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

Danilo Numeroso, Davide Bacciu, and Petar Veličković. Dual algorithmic reasoning. *arXiv preprint arXiv:2302.04496*, 2023.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 652–660, 2017.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Georg Sauthoff, Mathias Möhl, Stefan Janssen, and Robert Giegerich. Bellman's gap—a language and compiler for dynamic programming in sequence analysis. *Bioinformatics*, 29(5):551–560, 2013.

Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

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

Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.

Petar Veličković, Lars Buesing, Matthew Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *Advances in neural information processing systems*, 33: 2232–2244, 2020.

Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pp. 22084–22102. PMLR, 2022.

Tim Vieira. Automating the analysis and improvement of dynamic programming algorithms with applications to natural language processing. *arXiv preprint arXiv:2603.13242*, 2026.

Tim Vieira, Matthew Francis-Landau, Nathaniel Wesley Filardo, Farzad Khorasani, and Jason Eisner. Dyna: Toward a self-optimizing declarative language for machine learning applications. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 8–17, 2017.

Zengkai Wang, Weizhi Liao, Youzhen Jin, and Zijia Wang. Performance guarantees of recurrent neural networks for the subset sum problem. *Biomimetics*, 10(4):231, 2025.

Nathaniel Weir, Xingdi Yuan, Marc-Alexandre Côté, Matthew Hausknecht, Romain Laroche, Ida Momennejad, Harm Van Seijen, and Benjamin Van Durme. One-shot learning from a demonstration with hierarchical latent language. *arXiv preprint arXiv:2203.04806*, 2022.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

Daxin Zhu, Lei Wang, Tinran Wang, and Xiaodong Wang. A space efficient algorithm for the longest common subsequence in k-length substrings. *Theoretical Computer Science*, 687:79–92, 2017.

Table 5: Effect of foundation-training scale on Cartesian recurrence recovery. Data scaling varies the number of training examples while retaining the complete rule bank; rule/data scaling jointly increases recurrence coverage and training-set size.

	Data Scaling		Joint Rule/Data Scaling		
Examples	**Exact (%)**	**Cell (%)**	**Rule Coverage**	**Exact (%)**	**Cell (%)**
50K	89.92	96.55	25%	77.58	91.69
100K	94.44	97.98	50%	90.72	96.38
150K	96.39	98.71	75%	95.14	98.19
200K	97.54	99.15	100%	97.54	99.15
250K	97.75	99.13	–	–	–

A Illustrative LLM Recurrence-Induction Probe

We evaluate GPT-4o on 2,000 capacity-DP episodes spanning 2,000 distinct canonical recurrence rules. Each episode provides three solved input–DP-table demonstrations and asks the model to infer the underlying recurrence from the same rule grammar used by our simulator. GPT-4o returns only the recurrence; we then freeze and independently execute the predicted rule on a fourth, unseen query input. Despite the additional demonstrations, GPT-4o exactly reconstructs only 6 of 2,000 query tables, yielding 0.30% table-exact accuracy and 20.00% cell accuracy. Exact recovery of the canonical recurrence is only 0.10%.

The predictions instead exhibit a strong tendency toward a small set of familiar recurrence patterns. Although the evaluation contains 2,000 distinct target rules, GPT-4o produces only 177 distinct predicted rules; the three most frequent predictions alone account for 42.45% of all episodes, and 91.20% of predictions use a `max` aggregator. This concentration suggests that the model often maps solved tables to plausible DP templates rather than identifying the precise executable recurrence supported by the demonstrations. These results highlight the difficulty of recurrence recovery: identifying a familiar DP pattern is not enough; the exact recurrence structure must also be inferred correctly from the solved tables.

B Choice of support size.

The foundation model benefits from multiple support traces because a single solved table can leave several recurrence candidates consistent with the observed behavior. Increasing K provides complementary evidence across inputs, helping disambiguate transition structure, initialization, and input-dependent conditions. However, once the support set is sufficiently informative, additional traces provide diminishing returns while increasing encoder input size and computation. We therefore choose the support size of $K = 4$ for foundation training. As a representative validation, Figure 5 sweeps K from 1 to 5 on Cartesian state-lattice recurrence recovery. Exact-table reconstruction rises from 91.34% at $K = 1$ to 97.54% at $K = 4$, while $K = 5$ yields a similar saturated result. This indicates that $K = 4$ provides sufficient complementary evidence for reliable recurrence identification while maintaining a compact support set.

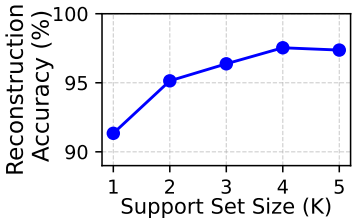

Figure 5: Table reconstruction versus foundation support size K.

C Effect of Foundation-Training Scale

To directly examine the contribution of foundation training, we vary both the amount of synthetic training data and the diversity of recurrence rules in a representative Cartesian state-lattice setting (Table 5). With the complete rule bank fixed, increasing the training set from 50K to 200K examples improves exact-table recovery from 89.92% to 97.54% and valid-cell recovery from 96.55% to 99.15%. Increasing the dataset further to 250K provides only a marginal additional gain in exact-table recovery, indicating that performance is already close to saturation at the scale used in our main experiments.

Recurrence diversity provides an additional benefit beyond simply increasing the number of examples. When rule-bank coverage and training-set size are jointly increased from 25%/50K to 100%/200K,

exact-table recovery rises from 77.58% to 97.54%, while valid-cell recovery increases from 91.69% to 99.15%. The intermediate settings improve consistently as coverage expands. These controlled scaling results show that the foundation stage contributes systematically to recurrence recovery: both sufficient training data and broad rule coverage strengthen the learned structural prior, while the diminishing returns beyond 200K examples indicate that the selected training scale is sufficient rather than unnecessarily oversized.

D Mathematical Analysis of the Learned Prior

We formalize why a sufficiently informative learned structural prior can improve recovery when limited support evidence leaves multiple recurrences consistent with the observed DP tables.

Let $\mathcal{R}$ denote the space of behaviorally distinct executable recurrences, and let $S = \{(x_j, D_j)\}_{j=1}^{m}$ denote the observed support set, where x_j is an input and D_j is its corresponding DP table. Let $r^\star \in \mathcal{R}$ denote the unknown target recurrence, such that

$$D_j = \mathrm{Exec}(r^\star, x_j), \forall j \in \{1, \ldots, m\}. \tag{13}$$

Define the set of support-consistent recurrences as

$$V(S) = \{r \in \mathcal{R} : \mathrm{Exec}(r, x_j) = D_j,\ \forall j \in \{1, \ldots, m\}\}. \tag{14}$$

It follows that $r^\star \in V(S)$. When the support evidence is insufficient, $|V(S)| > 1$: multiple behaviorally distinct recurrences agree on all support examples but may differ on unseen inputs.

We view the target recurrence as drawn from a task distribution, so the probabilities below are posterior probabilities under this distribution conditioned on the observed support set. For each $r_i \in V(S)$, let

$$q_i = \Pr(r^\star = r_i \mid S), \qquad s.t. \sum_{r_i \in V(S)} q_i = 1, \tag{15}$$

where q_i is the posterior probability that r_i is the correct generalizing recurrence.

Let $\hat{r}_{\mathrm{sym}}$ denote the recurrence returned by a consistency-based symbolic procedure, and define

$$a_i = \Pr(\hat{r}_{\mathrm{sym}} = r_i \mid S), \qquad s.t. \sum_{r_i \in V(S)} a_i = 1. \tag{16}$$

Conditioned on S, the symbolic procedure has no information about $r^\star$ beyond the support constraints. Modeling its selection among support-consistent recurrences as conditionally independent of $r^\star$, its probability of selecting the correct recurrence is

$$\begin{aligned} P_{\mathrm{sym}}(S) &= \Pr(\hat{r}_{\mathrm{sym}} = r^\star \mid S) && (17)\\ &= \sum_{r_i \in V(S)} \Pr(\hat{r}_{\mathrm{sym}} = r_i,\ r^\star = r_i \mid S) && (18)\\ &= \sum_{r_i \in V(S)} a_i q_i \le \max_{r_i \in V(S)} q_i. && (19) \end{aligned}$$

The inequality follows because $\sum_i a_i q_i$ is a convex combination of the posterior probabilities $\{q_i\}$.

A learned-prior method instead selects

$$\hat{r}_{\mathrm{prior}} \in \arg\max_{r_i \in V(S)} \pi_\theta(r_i \mid S), \tag{20}$$

where π_θ is the support-conditioned recurrence distribution induced by foundation training. Its probability of selecting the correct generalizing recurrence is

$$P_{\mathrm{prior}}(S) = \Pr(r^\star = \hat{r}_{\mathrm{prior}} \mid S). \tag{21}$$

If foundation training enables the learned-prior method to select a posterior-maximal support-consistent recurrence,

$$\hat{r}_{\mathrm{prior}} \in \arg\max_{r_i \in V(S)} q_i, \tag{22}$$

then

$$P_{\text{prior}}(S) = \max_{r_i \in V(S)} q_i \geq P_{\text{sym}}(S). \tag{23}$$

Equation 23 establishes a conditional Bayes-optimality result: when multiple recurrences remain support-consistent, a preference better aligned with the task posterior increases the likelihood of selecting the recurrence that generalizes correctly.

E METRICS

Canonical Rule Agreement. Canonical rule agreement is a strict symbolic diagnostic that compares the decoded recurrence against one canonical reference encoding. For inference attempt a on target task f, we define

$$\text{RuleExact}_f^{(a)} = \mathbb{1}\left[\exists \pi \in \Pi_{\text{comm}} \text{ s.t. } \forall t \in \mathcal{A}(r_f),\ \hat{y}^{(a)}_{\pi(t)} = y_t\right].$$

Here, r_f is the reference recurrence, $\mathcal{A}(r_f)$ is its set of semantically active rule components after masking grammar-fixed or unused components, y_t and $\hat{y}_t^{(a)}$ are the reference and predicted labels for component t, and Π_{comm} contains the identity permutation and, when applicable, symmetry permutations such as swapping predecessor slots under a commutative aggregator. With A independent inference attempts, the reported canonical rule agreement is

$$\text{CanonicalAcc}_f = \frac{1}{A}\sum_{a=1}^{A} \text{RuleExact}_f^{(a)}.$$

Generalization Accuracy. Generalization accuracy measures how often an independently recovered recurrence generalizes perfectly to the full held-out test set. For inference attempt a on target task f, let $\hat{r}_f^{(a)}$ denote the recovered recurrence and let $\mathcal{D}_f^{\text{test}}$ denote the held-out test set. We define the success of attempt a as

$$\text{GenSuccess}_f^{(a)} = \mathbb{1}\left[\forall (x, D_f(x)) \in \mathcal{D}_f^{\text{test}},\ \forall z \in \Omega_f(x),\ \text{Exec}(\hat{r}_f^{(a)}, x)[z] = D_f(x)[z]\right].$$

Here, $D_f(x)$ is the reference DP table, $\Omega_f(x)$ is the set of valid DP states, and $\text{Exec}(\hat{r}_f^{(a)}, x)$ is the table produced by executing the recovered recurrence with the simulator for the corresponding geometry. The task-level generalization accuracy is the fraction of successful inference attempts:

$$\text{GenAcc}_f = \frac{1}{A}\sum_{a=1}^{A} \text{GenSuccess}_f^{(a)}.$$

An attempt contributes 1 only if its recovered recurrence exactly reconstructs every valid state of every held-out test table. In the implementation, repeated recovered recurrences are deduplicated before test-set execution to avoid redundant computation, and the resulting test-set outcome is assigned back to each attempt that produced that recurrence.

Exact Table Recovery. Exact table recovery measures whether an executed recurrence reconstructs each individual evaluation table exactly:

$$\text{TableExact} = \frac{1}{\sum_{e\in\mathcal{E}} K_e}\sum_{e\in\mathcal{E}}\sum_{k=1}^{K_e} \mathbb{1}\left[\forall z \in \Omega_f(x_k),\ \hat{D}_f(x_k)[z] = D_f(x_k)[z]\right].$$

Here, $\mathcal{E}$ is the evaluation set, K_e is the number of support tables in evaluation item e, and $\hat{D}_f(x_k)$ denotes the table generated by the decoded recurrence on support input x_k. Each table contributes one binary decision; any incorrect valid state makes the table incorrect.

Valid-Cell Recovery. Valid-cell recovery measures reconstruction accuracy over individual valid DP states:

$$\text{CellAcc} = \frac{\sum_{e\in\mathcal{E}}\sum_{k=1}^{K_e}\sum_{z\in\Omega_f(x_k)} \mathbb{1}\left[\hat{D}_f(x_k)[z] = D_f(x_k)[z]\right]}{\sum_{e\in\mathcal{E}}\sum_{k=1}^{K_e} |\Omega_f(x_k)|}.$$

Here, $\Omega_f(x_k)$ denotes the valid states of the reference table; for interval-decomposition tasks, padded or structurally invalid interval states are excluded. This metric captures partial functional correctness when the recovered recurrence is nearly correct but fails on only a subset of states.

F Decoded Components

Table 6 lists the learnable rule components for each DP geometry. Only components required by the recurrence are learned; grammar-fixed or inactive fields are masked during training and evaluation.

Table 6: Active decoder heads.

Mode	Learned symbolic heads
Unary temporal chain / Cartesian state lattice	Two predecessor tuples: row delta, column coefficient, column bias, value coefficient, and value bias; aggregation operator, invalid sentinel, and boundary initialization.
Symbolic alignment lattice	Three predecessor tuples with row delta, column bias, and constant bias; primary aggregation operator, secondary aggregation operator, branch mode, false-branch coefficient, invalid sentinel, and initialization.
Interval-decomposition lattice	Aggregation, initialization, invalid-state handling, boundary or local-cost terms, source-table choices, interval offsets, split-point bounds, and optional component-selection controls for single or coupled interval recurrences.

G Training Configuration

We use the same foundation-training recipe across the regular modes unless otherwise specified. The values shown in Table 7 are the default settings used for the held-out evaluations, with model selection performed on the validation split.

For each held-out task, adaptation is performed on task-specific solved support examples generated using the corresponding recurrence-independent brute-force solver. The adaptation training split is used for reward-only policy updates, the validation split is used to select the adapted checkpoint, and the test split is reserved for the final post-adaptation evaluation; ground-truth recurrence labels are not used for adaptation. The elite-buffer parameters control the self-distillation term: a small number of high-reward selected recurrences are kept, evaluated, and reused as temporary pseudo-targets during adaptation. Table 8 presents the detailed parameter settings.

Table 7: Foundation training configurations.

Setting	Value
Optimizer	Adam
Learning rate	10^{-3}
Rule CE weight	1.0
RL weight	1.0
Entropy weight	10^{-3}
Gradient clipping	0.5
Data-loader workers	4

Table 8: Adaptation configurations.

Setting	Value
Optimizer	Adam
Adapt learning rate	10^{-3}
Adapt entropy weight	5×10^{-2}
Elite buffer size	5
Elite CE weight	0.10
Elite candidates per batch	2
Elite evaluation batches	4

For the baseline comparison, all methods operate over the same geometry-specific recurrence space and are evaluated using the same saved support and test splits. Each generalization attempt uses one solved support instance ($K = 1$) to synthesize a recurrence, which is then frozen and evaluated on the same 1,500-example held-out test set; the test split is never used for synthesis, adaptation, or model selection. DeepCoder and DreamCoder are pretrained on the corresponding synthetic foundation data and adapted without target recurrence labels, then perform neural-guided recurrence search at inference. DreamCoder additionally uses its wake–sleep library-learning procedure, while Z3 searches the same recurrence space directly from the observed support table using symbolic constraints. All methods are evaluated over the same 150 generalization attempts.

H LIMITATIONS AND FUTURE WORK

DEEPDP currently assumes a provided DP geometry, input-feature interface, and terminal readout convention, enabling structured recurrence inference within a finite geometry-specific rule space. This formulation provides several directions for extension, including jointly inferring task metadata, supporting partially observed or noisy tables, and expanding the recurrence grammar to broader DP formulations. Hierarchical or staged decoding could help control the resulting synthesis space, while confidence-aware recovery could further support recurrence synthesis when the available evidence does not uniquely determine a single exact rule.