Transformers as Dynamic Programming Solvers[*]

Zhenrui Zheng[†] **& Zhixuan Tan**[†]
School of Data Science
Chinese University of Hong Kong, Shenzhen
`{225040512, 225040506}@link.cuhk.edu.cn`

Abstract

Large Transformer models exhibit a remarkable capacity for in-context few-shot learning, enabling them to generalize to novel tasks from a few demonstrative examples. Despite extensive research into this phenomenon, its application to emulating structured algorithmic paradigms, particularly dynamic programming (DP), remains underexplored. This work provides empirical evidence that Transformers can function as in-context solvers for problems requiring dynamic programming. We specifically investigate this capability through the lens of the single-source shortest path (SSSP) problem on directed graphs. Our findings indicate that, beyond this specific case, Transformers can learn to effectively implement the recursive logic inherent in DP algorithms, suggesting their potential as general-purpose algorithmic executors for this problem class. Furthermore, we conduct a series of analyses to dissect the factors influencing this emergent capability, with a focus on training data characteristics. A key finding reveals a strong positive correlation between the model's generalization performance and the size of the training dataset (graph diversity); larger training datasets significantly enhance the model's capacity for in-context algorithmic execution. However, the diversity of node label mappings appears unable to improve results, suggesting that "diversity" requires a certain sense of "non-isomorphic" data. However, we also observe substantial variability in generalization performance across different random initializations, indicating that the optimization landscape contains multiple local minima with varying generalization properties.

1 Introduction

The advent of large language models (LLMs) based on the Transformer architecture has introduced a new paradigm of computation: in-context learning (ICL). This emergent capability allows models to rapidly adapt to novel tasks by conditioning on a few demonstrative examples instantiated within the input prompt. A significant line of inquiry has focused on elucidating the underlying mechanisms of ICL, revealing that Transformers can, in certain contexts, approximate computational processes such as learning linear functions and implementing iterative optimization algorithms like gradient descent.

Despite these advances, the capacity of Transformers to emulate more complex, state-dependent algorithmic paradigms remains an open question. Dynamic programming (DP) stands as a cornerstone of algorithm design, offering efficient solutions to a wide array of optimization problems by adhering to the principle of optimality and leveraging memoization of solutions to overlapping subproblems. The recursive and stateful nature of DP presents a formidable challenge for standard Transformer architectures, whose core mechanism is not explicitly designed for maintaining structured memory or

[*]This manuscript serves as the final project for the course DDA6040.
[†]Equal contribution

executing recursive logic. Consequently, whether Transformers can internalize and execute the core principles of DP is a critical, yet underexplored, research frontier.

This paper aims to bridge this gap by systematically investigating the ability of Transformers to function as in-context dynamic programming solvers. We select the single-source shortest path (SSSP) problem on directed graphs—a canonical exemplar of DP—as our primary testbed. The problem is framed as an autoregressive sequence generation task, where the model is trained to produce the step-by-step execution trace of a DP-based algorithm (e.g., a variant of Bellman-Ford or Dijkstra). Our empirical results provide compelling evidence that Transformers can indeed learn to approximate the iterative state updates and decision-making processes inherent to the DP algorithm, successfully solving SSSP instances not seen during training.

Furthermore, we conduct a detailed analysis to delineate the factors conditioning this emergent algorithmic capability. A central finding of our investigation is the pivotal role of data characteristics in enabling generalization. We demonstrate that the model's generalization performance on in-context DP tasks is critically dependent on the size of the training dataset (graph diversity). Models exposed to larger training datasets exhibit significantly enhanced performance, suggesting that increased graph diversity compels the model to learn the abstract, symbolic structure of the algorithm rather than relying on superficial correlations in the input sequences. However, the diversity of node label mappings appears unable to improve results, suggesting that "diversity" requires a certain sense of "non-isomorphic" data. However, we also observe substantial variability in generalization performance across different random initializations, even under identical training conditions, indicating that the optimization landscape contains multiple local minima with varying generalization properties. This insight underscores the importance of data curation and training strategies in unlocking the latent algorithmic reasoning abilities of large language models.

2 Background and Related Work

2.1 In-Context Learning of Algorithms

The capability of Transformers to learn functions in-context has been a major focus of recent theoretical machine learning research.

- **Linear Function Learning:** Garg et al. (2022) and Akyürek et al. (2022) established that Transformers can learn linear regression in-context, effectively implementing OLS or Gradient Descent. This laid the groundwork for viewing ICL as "algorithmic."
- **Complexity Hierarchy:** Bhattamishra et al. (2023) explored the complexity classes Transformers can handle, showing they can learn simple Boolean functions but struggle with complex automata without CoT.
- **Algorithmic Emulation:** Hu et al. (2025) proved that fixed-weight Transformers can emulate a broad class of algorithms via prompting, effectively acting as "prompt-programmable" machines.
- **In-Context TD Learning:** Wang et al. (2024) demonstrated that Transformers can perform Temporal Difference learning in-context, a direct link to the Bellman updates used in DP.

2.2 Chain-of-Thought (CoT) and "Scratchpads"

For DP problems, which require storing intermediate states, standard ICL (predicting the answer directly) is often insufficient.

- **CoT for DP:** Cheng et al. (2023) theoretically and empirically showed that CoT enables Transformers to solve DP problems like Longest Increasing Subsequence (LIS) and Edit Distance. The CoT effectively "unrolls" the DP table into the context window.
- **Scratchpads:** Nye et al. (2021) introduced the "scratchpad" concept, allowing the model to write out intermediate computation steps. This is crucial for SSSP on large graphs, as the path finding requires transitive steps.
- **Recursion of Thought:** Lee et al. (2023) extended this to "Recursion of Thought," using a divide-and-conquer strategy to solve problems too large for a single context.

- **Transformer Learning A^* Algorithm:** Lehnert et al. (2024) showed that Transformers can learn the A^* algorithm in grid world pathfinding and Sokoban problems, showing the potential of Transformers to learn complex algorithms.

2.3 Neural Algorithmic Reasoning (NAR)

- **DeepMind's CLRS:** Veličković et al. (2022) established the CLRS-30 benchmark, pushing for models that align with classical algorithms (Bellman-Ford, Prim's, etc.). They emphasize "Algorithmic Alignment"—the architecture should match the algorithm.
- **Multiple Solutions:** Georgiev et al. (2024) explored NAR for problems with multiple correct solutions (like SSSP on unweighted graphs), arguing that models should learn the distribution of solutions.
- **Knapsack & Pseudo-Polynomial:** Pozgaj et al. (2025) extended NAR to pseudo-polynomial problems like Knapsack, using DP table supervision.

3 Methodology

3.1 Problem Formulation: Single-Source Shortest Path

We focus on the single-source shortest path (SSSP) problem on directed graphs as our testbed for evaluating Transformers as dynamic programming solvers. Formally, given a directed graph $G = (V, E)$ with vertex set V and edge set E, where each edge $(u, v) \in E$ has a non-negative weight $w(u, v) \geq 0$, and a source vertex $s \in V$ and sink vertex $g \in V$, the problem is to find the shortest path from s to g.

This problem can be solved using dynamic programming through Dijkstra's algorithm, which maintains a priority queue and iteratively relaxes edges. The DP formulation for SSSP can be expressed as:

$$d(s) = 0, \tag{1}$$

$$d(v) = \min_{(u,v)\in E}\{d(u) + w(u, v)\} \quad \forall v \in V \setminus \{s\}, \tag{2}$$

where $d(v)$ represents the shortest distance from source s to vertex v. We use graphs with a fixed size of $|V| = 8$ vertices, and edge weights are sampled from the range $[0, 9]$.

3.2 Formulating Shortest Path as Next Token Prediction Problem

To enable the Transformer to learn and execute the Dijkstra algorithm, we formulate the shortest path problem as a next token prediction task. Each problem instance is encoded as a token sequence consisting of two parts: a *prompt* that describes the graph structure and query, and an *events* sequence that represents the step-by-step execution of the Dijkstra algorithm.

Prompt Encoding: The prompt encodes the graph structure and query as a sequence of tokens:

$$\text{prompt} = [\text{start}, \text{node}_s, \text{sink}, \text{node}_g, \text{edge}, \text{node}_i, \text{node}_j, w_{ij}, \ldots], \tag{3}$$

where node_s and node_g denote the source and sink vertices, and each edge (i, j) with weight w_{ij} is represented by the tokens $[\text{edge}, \text{node}_i, \text{node}_j, w_{ij}]$.

Events Encoding: The events sequence captures the execution trace of Dijkstra's algorithm, representing the algorithm's state transitions:

$$\text{events} = [\text{bos}, \text{close}, \text{node}_v, d_v, \text{open}, \text{node}_u, d_u, \ldots, \text{path}, \text{node}_v, d_v, \ldots, \text{eos}], \tag{4}$$

where close events mark vertices being extracted from the priority queue with their final distances, open events mark vertices being added or updated in the queue, and path events trace the shortest path from sink to source[1] by backtracking.

[1]When reconstructing the shortest path, we output the path in a backward manner, from the sink to the source. This is because if we output the path from the source to the sink, finding the next node starting from the source (given the pathfinding process) would have a linear time complexity, which is computationally impossible for transformers (with constant computation). Nevertheless, in practice, we found that both methods work, but we still adopt the more reasonable approach.

Figure 1 illustrates how a directed graph is encoded into a token sequence that the Transformer can process. The encoding transforms the graph structure and query into a sequential format, where nodes, edges, and their weights are represented as discrete tokens, enabling the model to learn the mapping from graph representations to algorithmic execution traces.

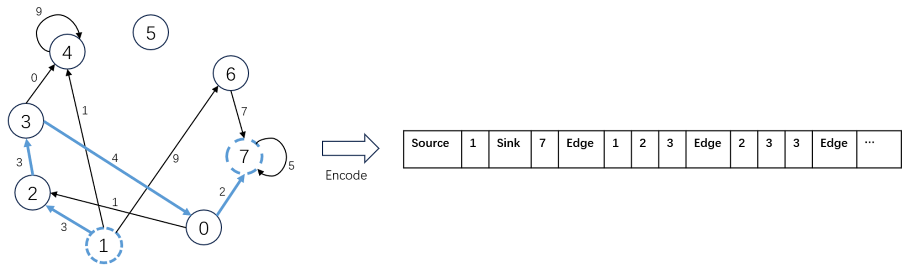

Figure 1: A schematic diagram of encoding a directed graph into a Transformer-processed token sequence. The graph structure (nodes and weighted edges) and queries (source and sink nodes) are converted into a sequential token representation, where each component (node, edge, weight, and algorithmic event) is mapped to discrete tokens that the model can process autoregressively. In this diagram, node 1 is the source, and node 7 is the sink, with the actual shortest blue path highlighted.

Training Format: During training, the model receives the concatenated sequence prompt $\oplus$ events as input, where $\oplus$ denotes concatenation. The model is trained to predict the events sequence autoregressively, with loss computed only on the events portion (the prompt portion is masked out in the loss computation). This design allows the model to learn the mapping from graph structure to algorithmic execution trace.

4 Experiments

4.1 Experimental Setup

We employ a decoder-only Transformer architecture based on the LLaMA model family. The vocabulary consists of special tokens, numerical tokens (0-99), and node label tokens. Node labels are randomly reassigned to different token IDs for each training sample, with the reassignment range controlled by a hyperparameter determining label diversity. During evaluation, node labels are sampled from a held-out label set unseen during training.

We generate synthetic directed graphs by sampling edge probability p_{edge} uniformly from $[0, 1]$ and including directed edges with probability p_{edge}. Edge weights are sampled uniformly from integers in $[0, 9]$. Source and sink vertices are randomly selected, and the Dijkstra algorithm generates ground-truth execution traces. The model is trained using standard next-token prediction with cross-entropy loss, with loss computed only on the events portion.

4.2 Experimental Design

We systematically vary two data characteristics: (1) training dataset size (1,000, 10,000, 100,000 samples), and (2) node label diversity (10, 100, 1,000 distinct mappings), yielding nine configurations. For each configuration, we maintain a fixed total number of training steps and repeat with four random seeds, resulting in 36 training runs.

Throughout training, we monitor: (1) *training loss* (cross-entropy on training set), (2) *success rate* (fraction of test instances with valid shortest path solutions), and (3) *perplexity (PPL)* (evaluated on held-out test set using teacher-forcing). The evaluation dataset is 1,000 samples held out from training data.

4.3 Results

Our results demonstrate that Transformers can learn to solve SSSP problems, though generalization performance varies significantly across configurations and random initializations, often despite achieving low training loss.

Figure 2 illustrates these training dynamics. Models that successfully generalize exhibit steady increases in test success rate alongside decreasing test perplexity, while those that fail to generalize maintain high test perplexity and low success rates.

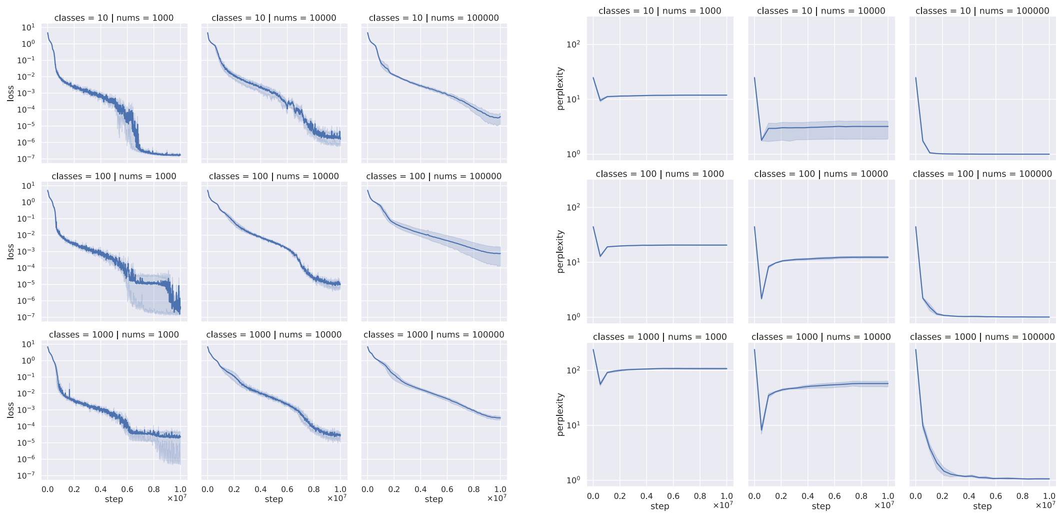

Figure 2: Training dynamics: (left) training loss and (right) test perplexity across different dataset sizes and label diversity settings.

4.3.1 Factors Influencing Generalization: Stochasticity and Data Characteristics

A interesting finding is the significant variability in generalization performance across random seeds, even under identical conditions. Different initializations can lead to dramatically different outcomes, with some seeds achieving near-perfect test performance while others fail to generalize beyond memorized patterns. This suggests the optimization landscape contains multiple local minima with varying generalization properties.

Figure 3 vividly illustrates both this stochasticity and the impact of data characteristics on generalization. For instance, with a dataset size of 10,000 and label diversity of 10, test success rates across four seeds range from approximately 15% to 45%, highlighting high sensitivity to initialization.

Beyond stochasticity, Figure 3 also summarizes the relationship between data characteristics and average test success rate. We observe a strong positive correlation between dataset size and generalization performance. Models trained on 100,000 samples consistently outperform those trained on smaller datasets, with larger datasets also significantly reducing variance across random seeds.

Conversely, increasing node label diversity does not appear to consistently improve generalization. While performance varies across different label diversity settings, this variation does not consistently correlate with label diversity across dataset sizes. This suggests that effective "diversity" for generalization requires variations in graph structure rather than merely in label assignments.

Indeed, the configuration with 100,000 samples achieves the highest performance, with all four seeds achieving test success rates above 90%, irrespective of label diversity, underscoring that graph diversity, primarily driven by dataset size, is the paramount factor influencing generalization.

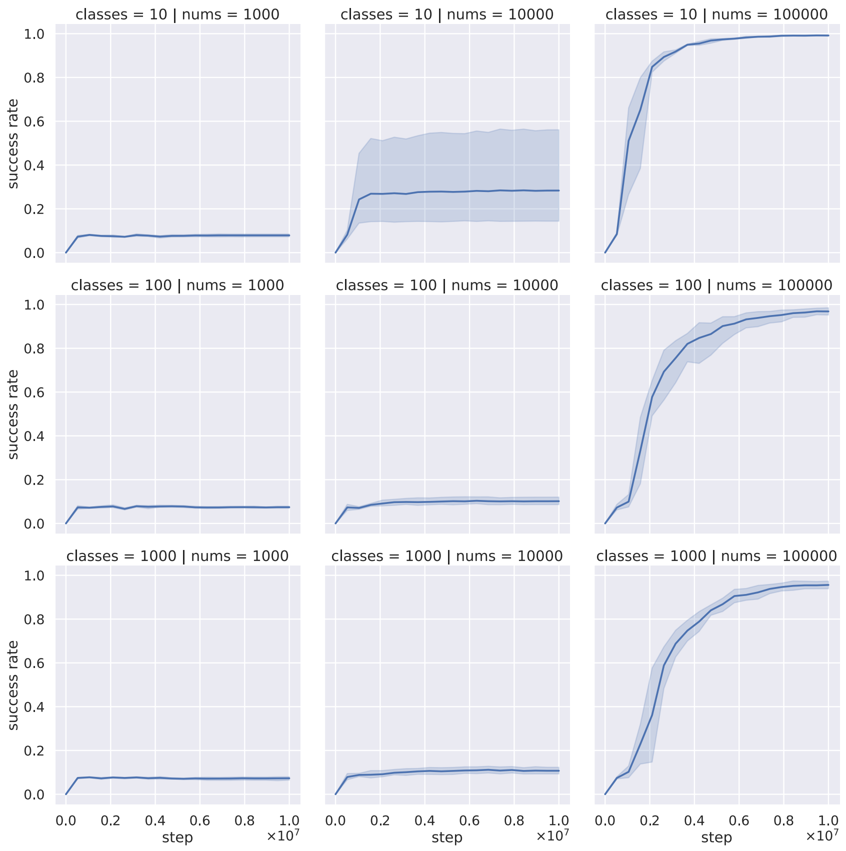

Figure 3: Average test success rate under different dataset sizes (num) and label diversity (classes). This demonstrates the crucial role of graph diversity (dataset size) in achieving generalization, while label diversity shows no positive effect.

5 Discussion

Our findings have several implications for developing general-purpose DP solvers. The observed sensitivity to random initialization suggests training strategies should incorporate multiple random seeds and potentially ensemble methods. The strong correlation between data diversity and generalization performance highlights the critical importance of data curation: simply scaling up training data is insufficient if it lacks representational diversity.

The distinction between training convergence and generalization success indicates that training loss alone is inadequate for assessing whether a model has learned a generalizable algorithm. Evaluation must explicitly test generalization to held-out distributions. The success of Transformers in learning DP algorithms, despite lacking explicit mechanisms for structured memory or recursive logic, suggests the in-context learning paradigm may be more powerful than previously recognized.

Our investigation is limited to SSSP on graphs of fixed size (8 vertices). Extending to other DP problems (longest common subsequence, edit distance, knapsack) and larger graph sizes would strengthen our conclusions. Future work could investigate how architectural choices interact with data characteristics, analyze the optimization landscape to understand why certain seeds lead to generalizable solutions, and explore more efficient training strategies.

6 Conclusion

This work demonstrates that Transformers can function as in-context solvers for dynamic programming problems, specifically learning and executing the single-source shortest path algorithm. Our investigation reveals that generalization is highly sensitive to both training data characteristics and random initialization.

A central finding is the critical role of graph diversity (dataset size): higher graph diversity significantly increases the likelihood of learning generalizable algorithmic patterns. However, node label diversity appears unable to improve results, suggesting that "diversity" requires a certain sense of "non-isomorphic" data. We also observe substantial variability across random seeds, suggesting the optimization landscape contains multiple local minima with varying generalization properties.

These findings underscore the importance of careful data curation and robust evaluation protocols that explicitly test generalization. While challenges remain, our results suggest Transformers may hold promise as general-purpose executors of dynamic programming algorithms.

References

Garg, S. et al. (2022) Transformers as Algorithms: Generalization and Stability in In-context Learning. arXiv:2301.07067. `https://arxiv.org/abs/2301.07067`

Akyürek, E. et al. (2022) The Effects of Pretraining Task Diversity on In-Context Learning of Ridge Regression. OpenReview. `https://openreview.net/pdf?id=EshX_qlA3o`

Bhattamishra, R. et al. (2023) Understanding In-Context Learning in Transformers and LLMs by Learning to Learn Discrete Functions. arXiv:2310.03016. `https://arxiv.org/abs/2310.03016`

Hu, Y. et al. (2025) In-Context Algorithm Emulation in Fixed-Weight Transformers. arXiv:2508.17550. `https://arxiv.org/abs/2508.17550`

Wang, Y. et al. (2024) Transformers Learn Temporal Difference Methods for In-Context Reinforcement Learning. OpenReview. `https://openreview.net/pdf?id=mEqddgqf5w`

Cheng, Y. et al. (2023) Towards Revealing the Mystery behind Chain of Thought: A Theoretical Perspective. arXiv:2305.15408. `https://arxiv.org/html/2305.15408v5`

Nye, M. et al. (2021) Show Your Work: Scratchpads for Intermediate Computation with Language Models. OpenReview. `https://openreview.net/forum?id=iedYJm92o0a`

Lee, J. et al. (2023) Recursion of Thought Prompting: Solving Complex Tasks Beyond Context Limits. LearnPrompting.org. `https://learnprompting.org/docs/advanced/decomposition/recursion_of_thought`

Lehnert, L., Sukhbaatar, S., Su, D., Zheng, Q., Mcvay, P., Rabbat, M., & Tian, Y. (2024) Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping. arXiv:2402.14083. `https://arxiv.org/abs/2402.14083`

Veličković, P. et al. (2022) The CLRS Algorithmic Reasoning Benchmark. arXiv:2205.15659. `https://arxiv.org/abs/2205.15659`

Georgiev, N. et al. (2024) Neural Algorithmic Reasoning with Multiple Correct Solutions. arXiv:2409.06953. `https://arxiv.org/html/2409.06953v4`

Pozgaj, D. et al. (2025) KNARsack: Teaching Neural Algorithmic Reasoners to Solve Pseudo-Polynomial Problems. arXiv:2509.15239. `https://arxiv.org/abs/2509.15239`