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, including training data composition and model architecture. A key finding reveals a strong positive correlation between the model's performance and the label diversity of graph nodes within the training corpus; greater diversity significantly enhances the model's capacity for in-context algorithmic execution.

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 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

*This manuscript serves as the final project for the course DDA6040.

†Equal contribution

Preprint.

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 representational diversity in the training data. We demonstrate that the model's generalization performance on in-context DP tasks is critically dependent on the diversity of node labels used during training. Models exposed to a richer variety of label mappings exhibit significantly enhanced performance, suggesting that this diversity compels the model to learn the abstract, symbolic structure of the algorithm rather than relying on superficial correlations in the input sequences. This insight underscores the importance of data curation 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) and Zhou 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.

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.

[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.

3.3 Transformer Architecture for DP Solving

We employ a decoder-only Transformer architecture based on the LLaMA model family, which has been shown to be effective for in-context learning tasks. The model processes input sequences in an autoregressive manner, predicting the next token given all previous tokens through causal attention mechanisms.

Token Vocabulary: The vocabulary consists of special tokens (e.g., pad, bos, eos, start, sink, edge, open, close, path), numerical tokens for edge weights and node distances (0-99), and node label tokens. Crucially, node labels node_0 through node_7 are randomly reassigned to different token ids for each training sample, within a specified range controlled by a hyperparameter that determines the diversity of node label representations. During evaluation, the node labels are sampled from a held-out label set, which is not seen during training. This randomization mechanism enforces the model to learn in-context DP algorithms, rather than memorizing specific node id sequences.

Model Configuration: The model uses a standard causal language modeling objective with cross-entropy loss. The vocabulary size is dynamically adjusted to accommodate the variable node label space. The architecture follows the standard LLaMA design with rotary position embeddings (RoPE) and grouped-query attention mechanisms, enabling efficient processing of variable-length sequences.

3.4 Training and Evaluation Setup

Data Generation: We generate synthetic directed graphs using a random graph generation procedure. For each graph, we sample an edge probability p_{edge} uniformly from $[0, 1]$, and for each pair of vertices (i, j), we include an directed edge with probability p_{edge}. Edge weights are sampled uniformly from integers in the range $[0, 9]$. Source and sink vertices are randomly selected from the vertex set. The Dijkstra algorithm is executed to generate the ground-truth (path finding) events sequence, which includes the complete execution trace from initialization to path reconstruction.

Node Label Diversity Mechanism: A key aspect of our training setup is the node label randomization mechanism. For each training sample, node labels are randomly reassigned to different token ids, with the reassignment range controlled by a hyperparameter. This hyperparameter controls the diversity of node label representations: larger values allow for more diverse label mappings, forcing the model to learn abstract algorithmic patterns rather than memorizing specific node id sequences. This mechanism is applied independently for each sample, ensuring that the same graph structure can appear with different node label encodings across training iterations.

Training Procedure: The model is trained using standard next-token prediction with cross-entropy loss. Training employs gradient accumulation, where gradients are accumulated over multiple mini-batches before performing a parameter update when a target effective batch size is reached. The loss is computed only on the events portion of each sequence, with the prompt portion masked out in the loss computation. We also use cosine learning rate scheduling and gradient clipping for training stability.

Evaluation Metrics: We evaluate the model's performance using several metrics: (1) *Algorithmic correctness*, provide the model with the graph prompt, assessing whether the generated execution trace corresponds to a valid Dijkstra algorithm execution; (2) *Perplexity*, To better evaluate the failure cases, we also evaluate the model with ground-truth evaluate sequences, in teacher-forcing manner, and observing the perplexity of the output sequence, position-wisely. The evaluation dataset is held out from the training data (90% train, 10% evaluation split).

Experimental Variations: To investigate the factors affecting the Transformer's DP-solving ability, we conduct controlled experiments varying: (1) the hyperparameter controlling node label diversity, (2) model size configurations (small, medium, large), (3) training hyperparameters (learning rate, batch size, number of epochs), and (4) graph generation parameters (edge probability distributions, weight ranges).

4 Experiments

4.1 Baseline: Transformer as DP Solver

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

4.2 Effect of Data Characteristics

4.2.1 Impact of Graph Node Label Diversity

4.2.2 Training Data Distribution

4.3 Effect of Model Aspects

4.3.1 Model Size and Capacity

4.3.2 Architecture Variations

5 Results and Analysis

5.1 Performance on Shortest Path Problems

5.2 In-Context Learning Capabilities

5.3 Key Observations

6 Discussion

6.1 Implications for General-Purpose DP Solvers

6.2 Limitations and Future Work

7 Conclusion

A Appendix

You may include other additional sections here.