Simulation of Graph Algorithms with Looped Transformers

Artur Back de Luca [1]   **Kimon Fountoulakis** [1]

Abstract

The execution of graph algorithms using neural networks has recently attracted significant interest due to promising empirical progress. This motivates further understanding of how neural networks can replicate reasoning steps with relational data. In this work, we study the ability of transformer networks to simulate algorithms on graphs from a theoretical perspective. The architecture we use is a looped transformer with extra attention heads that interact with the graph. We prove by construction that this architecture can simulate individual algorithms such as Dijkstra's shortest path, Breadth- and Depth-First Search, and Kosaraju's strongly connected components, as well as multiple algorithms simultaneously. The number of parameters in the networks does not increase with the input graph size, which implies that the networks can simulate the above algorithms for any graph. Despite this property, we show a limit to simulation in our solution due to finite precision. Finally, we show a Turing Completeness result with constant width when the extra attention heads are utilized.

1. Introduction

Recent advancements in neural network models have significantly impacted various domains, most notably vision (Yuan et al., 2021; Khan et al., 2022; Dehghani et al., 2023), and natural language processing (Wei et al., 2022b; Touvron et al., 2023). Transformers (Vaswani et al., 2017), at the forefront of these developments, have become standard for many complex tasks. These successes have also shed light on the capabilities of neural networks in algorithmic reasoning (Veličković & Blundell, 2021), such as basic arithmetic (Lee et al., 2024), sorting (Tay et al., 2020; Yan et al., 2020; Rodionov & Prokhorenkova, 2023), dynamic programming

(Dudzik & Veličković, 2022; Ibarz et al., 2022) and graph algorithms (Veličković et al., 2022; Cappart et al., 2023).

In this work, we focus on algorithmic reasoning on graphs. Current empirical results display a promising degree of scale generalization on graphs (Yan et al., 2020; Tang et al., 2020; Ibarz et al., 2022; Veličković et al., 2022; Bevilacqua et al., 2023; Numeroso et al., 2023). The predominant approach in these studies is to train a neural network to execute a step of a target algorithm and use a looping mechanism to execute the entire algorithm. The looping mechanism is crucial since it allows the execution of long processes on graphs. Motivated by these empirical results, our goal is to study the ability of looped neural networks to simulate algorithms on graphs of varying sizes. Briefly, by simulation, we mean the ability of a neural network to provide the correct output of a step of an algorithm for every step. In all our results we utilize a looped transformer architecture with extra attention heads that interact with the graph. Instead of storing the graph in the input, we encode it using its adjacency matrix which multiplies the attention head. This allows us to access data from the adjacency matrix without scaling the number of parameters of the network with the size of the graph.

Our contributions: We list our contributions below.

1. We demonstrate how a looped transformer architecture can simulate graph algorithms like Breadth-first search (BFS) & Depth-first search (DFS) (Moore, 1959), Dijkstra's shortest path algorithm (Dijkstra, 1959), and Kosaraju's algorithm for identifying strongly connected components (SCC) (Aho et al., 1974). These algorithms were chosen as part of the CLRS benchmark (Veličković et al., 2022). We also use these constructions to create a multitask model capable of simulating BFS, DFS, and Dijkstra's algorithm simultaneously.

2. In our results, the largest dimension of any weight matrix in our network, denoted by width, does not scale with the number of nodes or edges in the graph. This shows that algorithm simulation is possible for graphs of varying sizes, although limited by finite precision. Current results do not consider the looping mechanism and are constrained by the network's depth/width (Loukas, 2020; Xu et al., 2020), or they do not consider graph algorithms (Pérez et al., 2021; Giannou et al., 2023).

---

[1]David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.. Correspondence to: Artur Back de Luca <abackdel@uwaterloo.ca>.

3. We also provide a Turing Completeness result for the looped transformer with $O(1)$ width which uses additional attention heads to interact with the graph.

2. Related work

Empirical: A plethora of papers have been published on the ability of neural networks to execute algorithms on graphs. Notable empirical works include Tang et al. (2020); Yan et al. (2020); Veličković et al. (2020); Ibarz et al. (2022); Bevilacqua et al. (2023); Numeroso et al. (2023); Diao & Loynd (2023); Georgiev et al. (2023a;b); Engelmayer et al. (2023). These works leverage looping mechanisms and report favorable scale-generalization results. For a comprehensive review, see Cappart et al. (2023).

Theoretical: Currently, three types of complementary results study the ability of neural networks to execute algorithms. The first type is simulation results such as Siegelmann & Sontag (1995); Pérez et al. (2021); Giannou et al. (2023); Hertrich & Skutella (2023); Hertrich & Sering (2023). These results are demonstrated by providing analytic expressions of the neural networks that achieve simulation, similar to our results. For example, Siegelmann & Sontag (1995); Pérez et al. (2021) use simulations to prove the Turing Completeness of Recurrent Neural Networks (RNNs) and Transformers, respectively. Giannou et al. (2023) also shows Turing Completeness for Transformers through the simulation of SUBLEQ (Mavaddat & Parhami, 1988). However, these works do not consider graph data. Other works, such as Hertrich & Skutella (2023); Hertrich & Sering (2023) use RNNs and Multilayer-Perceptrons (MLPs) to simulate graph algorithms to solve problems like shortest paths, minimum spanning trees, and maximum flow. Although existing results on graph algorithms or Turing Completeness apply to graph algorithms, these approaches require the graph to be stored in the input data matrix rather than being part of the architecture. This imposes limitations on the size of the input graph, as discussed in Section 4.

The second type of result is using the Probably Approximate Correct learning framework to study the sample complexity of neural networks for executing algorithms. In Xu et al. (2020) the authors show that sample complexity is improved when the neural network is aligned with the structure of the algorithm. We also use the concept of alignment in our constructive proofs. However, our results are about the simulation of algorithms, which hold for any distribution of the test graph. Also, Xu et al. (2020) does not employ a looping mechanism. The third type is impossibility results. In this case, it is shown that neural networks cannot execute certain algorithms when the depth and width of the network are smaller than a lower bound. In Loukas (2020) the author studies the ability of graph neural networks to solve vari-

ous graph problems and also proves Turing Completeness. However, all results are limited by the depth and the width of the neural network which scales with the size of the input graph. In our case, due to the looping mechanism, the depth and width of the network are constant, and a single neural network can solve a particular problem for any input graph, subject to limitations imposed by finite precision.

3. Preliminaries

This section establishes preliminaries for our task of algorithmic simulation on graphs. Recognizing that an algorithm consists of multiple steps, we start by defining simulation for an individual step. Consider $h_F : \mathcal{X} \to \mathcal{Y}$ as the function we aim to simulate and $h_T : \mathcal{X}' \to \mathcal{Y}'$ as the neural network designed for this purpose. Note that h_F and h_T might operate in different representation spaces. For instance, $\mathcal{X}$ and $\mathcal{Y}$ might be the set of natural numbers while $\mathcal{X}'$ and $\mathcal{Y}'$ might be multidimensional reals. To address this, we use mappings $g_e : \mathcal{X} \to \mathcal{X}'$ and $g_d : \mathcal{Y}' \to \mathcal{Y}$ for encoding and decoding, respectively. In our constructions in Section 5.1, g_e organizes terms into our input matrix format and incorporates biases and positional encodings, while g_d simply extracts the appropriate columns from the output.

Definition 3.1 (Simulation). The neural network h_T is a successful simulator of the algorithmic step h_F if for every input $x \in \mathcal{X}$, $h_F(x) = g_d(h_T(g_e(x)))$. A transformer is said to *simulate* an algorithm if it can simulate each step of the algorithm (Giannou et al., 2023; Pérez et al., 2021).

We define a graph G with n nodes by its adjacency matrix $A \in \mathbb{R}_+^{n \times n}$, where $A_{i,j} > 0$ if nodes i and j are connected, otherwise $A_{i,j}$ is zero. In our work, we also use an input matrix $X \in \mathbb{R}^{K \times d}$, where d is the feature dimension and $K \geq n + 1$ is the number of rows that are set according to the simulation. The structure of X is further explained in Section 5.1. Because of the mismatch of length between X and A, we adopt a padded version of A, denoted $\tilde{A} \in \mathbb{R}^{K \times K}$. The entries of the first row and column of $\tilde{A}$ are set to zero. This is to align with the input's top row, which holds data not associated with any node, as further discussed in Section 5.1. The next n rows and columns correspond to the entries of A. Certain implementations require $K > n + 1$. In this case, we further pad $\tilde{A}$ with zeros in the extra rows and columns beyond the entries of A. For example, in our Turing Completeness result in Remark 6.6, K can exceed $n + 1$ because the number of rows may not be directly determined by the number of nodes.

4. The Architecture

We utilize a variation of the standard transformer layer in Vaswani et al. (2017) with an additional attention mechanism that incorporates the adjacency matrix. The additional
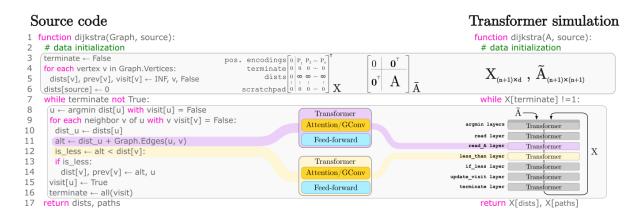
Source code

```
1  function dijkstra(Graph, source):
2    # data initialization
3    terminate ← False
4    for each vertex v in Graph.Vertices:
5      dists[v], prev[v], visit[v] ← INF, v, False
6    dists[source] ← 0
7    while terminate not True:
8      u ← argmin dist[u] with visit[u] = False
9      for each neighbor v of u with v visit[v] = False:
10       dist_u ← dists[u]
11       alt ← dist_u + Graph.Edges(u, v)
12       is_less ← alt < dist[v]:
13       if is_less:
14         dist[v], prev[v] ← alt, u
15     visit[u] ← True
16     terminate ← all(visit)
17   return dists, paths
```

Transformer simulation

```
function dijkstra(A, source):
  # data initialization
```

$$\mathbf{X}_{(n+1)\times d} \;,\; \tilde{\mathbf{A}}_{(n+1)\times(n+1)}$$

```
while X[terminate] !=1:
```

argmin layers	Transformer
read layer	Transformer
read_A layer	Transformer
less_than layer	Transformer
if_less layer	Transformer
update_visit layer	Transformer
terminate layer	Transformer

```
return X[dists], X[paths]
```

Transformer — Attention/GConv — Feed-forward (lines 8–12 block)

Transformer — Attention/GConv — Feed-forward (lines 13–16 block)

Positional encodings, terminate, dists, scratchpad rows forming X; $\begin{bmatrix} 0 & \mathbf{0}^\top \\ \mathbf{0}^\top & A \end{bmatrix} \tilde{A}$

Figure 1. A simplified illustration of the simulation of Dijkstra's algorithm using a looped transformer with extra attention heads that interact with the graph. On the left, we display the pseudocode of Dijkstra's algorithm, serving as the source code. The rightmost section shows the corresponding simulation via a transformer, where each step of the source code in the loop is simulated using one or more transformer blocks. We specifically focus on lines 11 and 12, highlighted to demonstrate the simulation of individual functions, as discussed in Section 5. At the top center of the figure, the encoding of graph information and variable scopes into $\tilde{A}$ and X is depicted. For clarity, X is shown in its transposed format. Throughout the transformer loop, $\tilde{A}$ remains constant, while X is updated in each iteration until the simulation meets its termination criteria. Upon termination, the decoding step extracts columns from X that correspond to the algorithm's desired output.

attention mechanism is described by the following equation:

$$\psi^{(i)}(X, \tilde{A}) := \tilde{A}\, \sigma\left(X W_Q^{(i)} W_K^{(i)\top} X^\top\right) X W_V^{(i)}, \quad (1)$$

where $W_V \in \mathbb{R}^{d\times d}$, W_Q, and $W_K \in \mathbb{R}^{d\times d_a}$ are the value, query, and key matrices, respectively, d_a represents the embedding dimension and σ denotes the hardmax[1,2] function. Note that setting $\tilde{A}$ as the identity matrix reduces equation (1) to a standard attention head. This modification efficiently extracts specific rows from matrix A. Conversely, applying attention to the transpose of $\tilde{A}$ enables the extraction of columns of A. These variations are important in our constructions, and they allow parallel execution of algorithmic subroutines which are further discussed in Section 5.2. We define a complete transformer layer as:

$$f(X, \tilde{A}) = f_{\mathrm{mlp}}(f_{\mathrm{attn}}(X, \tilde{A})) \quad (2)$$

---

[1] The hardmax is defined by $[\sigma(\Phi)]_i := \sum_{k\in K} e_k / |K|$, where e_k is the standard basis vector and $K = \{k \,|\, \Phi_{ik} = \max(\Phi_i)\}$.

[2] The softmax activation function can be used as well, since softmax approximates hardmax as the temperature parameter goes to zero. However, softmax introduces errors in the calculations. To guarantee small errors during the simulation, the temperature parameter will have to be a function of the size of the graph and other parameters related to finite precision. Softmax could further limit the simulation properties of the architecture beyond what we discuss in Section 6. However, this type of limitation is common, for example, see Giannou et al. (2023); Liu et al. (2023). In practice, softmax is preferred over hardmax. For this reason, in our empirical validation results in Appendix B.3 we also use softmax in combination with rounding layers in Appendix B.1.

where

$$f_{\mathrm{attn}}(X, \tilde{A}) = X + \sum_{i\in H} \psi^{(i)}(X, I_{n+1})$$
$$+ \sum_{i\in H_A} \psi^{(i)}(X, \tilde{A}) + \sum_{i\in H_{A^\top}} \psi^{(i)}(X, \tilde{A}^\top)$$

$$f_{\mathrm{mlp}}(X) = Z^{(m)} W^{(m)} + X, \quad Z^{(j+1)} = \phi(Z^{(j)} W^{(j)})$$

for $j = 0, \ldots, m-1$ and $Z^{(0)} = X$. Here, X represents the input matrix and $\tilde{A}$ is the padded adjacency matrix. Additionally, I_{n+1} is the identity matrix of size $n+1$, m and ϕ are the number of layers and activation function of f_{MLP}, respectively. Note that in the formulation of the multi-head attention, we adopt a residual connection and a sum of attention heads ψ in the format described in (2). H, H_A, and $H_{A^\top}$ are the index set of attention heads for the standard attention (Vaswani et al., 2017), as well as for the versions that utilize the adjacency matrix and its transpose, respectively. The total number of attention heads is $|H| + |H_A| + |H_{A^\top}|$. Furthermore, for the MLP, we define ϕ to be the ReLU function, we set $m = 4$, and we use matrices $W^{(i)} \in \mathbb{R}^{d\times d}$ as the parameters of the MLP, where d is the feature dimension of X. We integrate the bias into the linear operations by adding bias columns to X and also introduce a residual connection to f_{mlp}.

Algorithm 1 demonstrates the use of our model h_T, which is constructed by stacking multiple layers f as depicted in (2). In this setting, each forward pass feeds its output to a subsequent pass, creating a looping mechanism. This

loop continues until a pre-defined termination condition is met, such as the activation of a boolean element within the input, denoted by `term` in Algorithm 1. This setting and architecture are used to simulate various graph algorithms, as formally established in the results of Section 6.

Algorithm 1 Looped Transformer (Giannou et al., 2023)

Input: model h_T, matrices X and $\tilde{A}$, column term
1: **while** X[0,term] is **false do**
2:     $X = h_T(X, \tilde{A})$
3: **end while**

4.1. Discussion on the attention head in (1)

While encoding the graph in X is an alternative to our attention head in (1), it is subject to limitations. Specifically, concatenating the adjacency matrix to X results in linear dependence of the width of the network to the number of nodes, thereby limiting simulation to graphs with sizes smaller or equal to the one that the network was set up/trained on. Alternatively, appending the edges of the graph to the input results in the number of positional encodings being dependent on the edge count, which may increase quadratically as the number of nodes grows. This can limit simulation performance when the available positional encodings are limited, for instance, due to finite precision.

Finally, our attention head in (1) can be viewed as a message-passing layer that performs two convolutions. The first convolution corresponds to a complete graph, i.e., standard attention (Vaswani et al., 2017), and the second is a standard graph convolution (Kipf & Welling, 2017). Multiplying the two convolution matrices is important. In particular, the left multiplication of the attention matrix with $\tilde{A}$ in (1) allows for direct and easy access to data from $\tilde{A}$. This is because the attention matrix acts as an indicator matrix that extracts rows from $\tilde{A}$, see Section 5.3 for details. Note that our attention head in (1) differs from Graph Attention Networks (GAT) (Veličković et al., 2018), where the graph is incorporated in the attention mechanism. Although possible to use the attention mechanism in GAT to access the graph data, the proofs will be more elaborate than with (1).

5. Simulation Details

Our simulation results are constructive. We set the parameters of the layer in (2) such that we simulate each sub-routine within each algorithm. We start by describing the input to the model in Section 5.1. We provide two simulation examples of common subroutines among all algorithms in this paper. In Section 5.2 we provide an example of the implementation of the less-than function. This function is common among algorithms in this paper, and it illustrates how the neural network can achieve parallel computation

over the nodes. Next, in Section 5.3 we provide an example for reading information from the graph. Throughout the section, we use Dijkstra's algorithm as an algorithm example (as depicted in Figure 1). With the exemption of our Turing Completeness result in Remark 6.6, it is sufficient to set $K = n + 1$ for all algorithms in our paper. For this reason, and for simplicity, we set $K = n + 1$ in this section.

5.1. Input matrix

The input matrix encompasses all variables used in the algorithm. In the case of Dijkstra's algorithm in Figure 1, the lists of current distances and paths (`dists` and `paths`), and variables like the current node and its distance (`u` and `dist`) are all incorporated into X. Below, we describe in detail the general structure of X. We refer the reader to Figure 2 for a visualization of the structure of the matrix X.

The top row of X stores single variables, encapsulating the global information relevant to the algorithm. The bottom n rows of X are reserved for local variables, such as distances and paths, representing the node-specific data. This storage structure separates the global context from the more specific, node-related information of the algorithm. The input matrix X is augmented with columns for positional encodings, biases, and a scratchpad area. Each node receives a unique positional encoding, denoted by p_i. These concepts have been present in various contexts, see Giannou et al. (2023) for assigning functional blocks to the input and Akyürek et al. (2022); Wei et al. (2022a); Giannou et al. (2023) for the notion of scratch space. Distinct biases are applied to the top and bottom rows to simulate global and local functions effectively. X is equipped with flags indicating global or local states, regulating the algorithm's flow. These flags can either permit or prevent overwriting of fields or signal different stages in the algorithm's execution. For example, in all our simulations each of the last n entries of the node-wise flag named `visit` indicates whether the corresponding node has been visited, while the global variable `term`, signals the terminating condition for the algorithm. Finally, the scratchpad within X serves as a temporary storage space for variable manipulation or intermediate computations.

5.2. Less-than

The purpose of this operation is to determine whether each element in a specific column of X is smaller than its counterpart in another column. The result of this comparison is then recorded in another designated column. In the context of Dijkstra's algorithm, as shown in Figure 1, this function plays a critical role in determining if an alternate path is shorter than the shortest path currently known.

We express this function as: `less-than(X, C, D, E)` = `write(X[:,C] < X[:,D], X[:,E])`. Here, X is the input matrix, while C, D, and E are placeholders

Figure 2. Illustration of the input structure used in the simulation of graph algorithms. On the left, columns indicate node positions using circular positional encodings (Liu et al., 2023), detailed in Section 6. The next blocks on the right denote the global (in red) and local (in green) variables, which occupy the top and last n rows of X, respectively. In these blocks, the first column marks the bias of the corresponding variables. The symbol t indicates the termination flag, while symbols z, and x_i, y_i, are generic local and global variables, respectively. Finally, depicted in the far right block, the scratchpad is used for temporary storage and calculation. Non-shaded areas remain null during execution.

for the specified column indices in X. This simulation involves two key steps. First, we approximate the less-than operator ($<$) using the following expression:

$$X[:,C] < X[:,D] \approx \varepsilon^{-1}\phi\left(X[:,D] - X[:,C]\right) \quad (3)$$
$$-\varepsilon^{-1}\phi\left(X[:,D] - X[:,C] - \varepsilon\right),$$

where $\varepsilon > 0$ is a small tolerance value, and ϕ denotes the ReLU activation function. This operation approximates a step function of the difference $X[:,C]-X[:,D]$, where ε controls the sharpness of the threshold. If this difference is larger than ε, the subtraction of the terms post-ReLU equals ε, which simplifies to 1. If the difference is negative, both ReLU terms become zero. If the difference $X[:,C]-X[:,D]$ is non-negative but less than ε, the approximation linearly interpolates between 0 and 1, with a gradient of ε^{-1}. The second operation is the `write` function. In this operation, the output of the less-than comparison is placed in the specified field, while the original content of that field is erased. This operation is expressed as: $X[:,E] \leftarrow X[:,E] - X[:,E] + X[:,S_1]$, where S_1 is the scratchpad column that holds the result to be put in E.

Combining these two operations, we define the parameters for the layer f. To understand the purpose behind each operation, we start by defining a particular instance of the matrix X, which is used for this example:

$$X = \begin{bmatrix} & B_{\text{global}} & B_{\text{local}} & C & D & E & S \\ \cdots & 1 & 0 & c_0 & d_0 & e_0 & \cdots \\ \cdots & 0 & 1 & c_1 & d_1 & e_1 & \cdots \\ & \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & 0 & 1 & c_n & d_n & e_n & \cdots \end{bmatrix}.$$

Here, C, D and E are placeholders for column indices used

in (3), S is the scratchpad area, and B_{global} and B_{local} represent the biases for global and local variables respectively. First, in the definition of the attention layer, all values are set to 0, maintaining only the residual connection. For the definition of the parameters in f_{mlp}, we introduce the parameters layer by layer, followed by an explanation of what is achieved at each stage. For the first layer, we have:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } (i, j = E) \text{ or } (i = D, j \in \{S_1, S_2\}) \\ -1 & \text{if } i = C,\ j \in \{S_1, S_2\} \\ -\varepsilon & \text{if } i \in \{B_{\text{global}}, B_{\text{local}}\},\ j = S_2 \\ 0 & \text{otherwise,} \end{cases}$$

where S_1 and S_2 are entries in the scratchpad. Here, the parameters of the first layer $W^{(1)}$ keep the entries of the target column E, while it builds the arguments of ϕ in (3), inserting them in the scratchpad. Storing the intermediate values in the scratchpad is beneficial to prevent overwriting issues, as it avoids the use of columns involved in the definition of the equation. The result is given by:

$$XW^{(1)} = \begin{bmatrix} & E & S_1 & S_2 \\ \cdots & e_0 & d_0 - c_0 & d_0 - c_0 - \varepsilon & \cdots \\ \cdots & e_1 & d_1 - c_1 & d_1 - c_1 - \varepsilon & \cdots \\ & \vdots & \vdots & \vdots \\ \cdots & e_n & d_n - c_n & d_n - c_n - \varepsilon & \cdots \end{bmatrix}.$$

Subsequently, the parameters of the second layer $W^{(2)}$ are:

$$(W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } i = E,\ j = i \\ \varepsilon^{-1} & \text{if } i = S_1,\ j = i \\ -\varepsilon^{-1} & \text{if } i = S_2,\ j = S_1 \\ 0 & \text{otherwise.} \end{cases}$$

Here, $W^{(2)}$ is responsible for subtracting the post-ReLU terms and dividing them by ε^{-1}. This result is then stored in the scratchpad entry S_1:

$$Z^{(1)}W^{(2)} \approx \begin{bmatrix} & E & S_1 \\ \cdots & e_0 & c_0 < d_0 & \cdots \\ \cdots & e_1 & c_1 < d_1 & \cdots \\ & \vdots & \vdots \\ \cdots & e_n & c_n < d_n & \cdots \end{bmatrix}.$$

Finally, for the two remaining operations, we have:

$$(W^{(3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{E, S_1\},\ j = i \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(4)})_{i,j} = \begin{cases} -1 & \text{if } i, j = E \\ 1 & \text{if } i = S_1,\ j = E \\ 0 & \text{otherwise.} \end{cases}$$

5

In these two operations, the output of S_1 is preserved and then placed in the desired field E. Additionally, the entries of E, which were kept in the first three layers, are subtracted from the target field in $W^{(4)}$, effectively removing the previous entry in E. Furthermore, to address cases where the input entries are negative, it is necessary to incorporate additional parameters. These parameters should follow the same structure as previously provided, with inverted signs in the parameters $W^{(1)}$ and $W^{(4)}$ for columns C, D, and E. These additional entries are generally placed in the scratchpad.

Parallel computation over the nodes: The node-wise format in Section 5.1 is particularly beneficial for operations like less-than in equation (3) because it operates across entire columns. This approach enables parallel processing for all nodes in a graph, enhancing efficiency in algorithms. For example, in Dijkstra's algorithm, the typical implementation involves iterating through each non-visited neighbor of a node. However, by integrating an additional masking function, we can filter out visited or non-neighboring nodes. This allows us to conduct all necessary comparisons and conditional selections for every node simultaneously within the inner loop of the implementation. This modification enhances the algorithm's efficiency by reducing the need for repetitive looping in transformer-based simulations.

5.3. Read row from A

In this section, we provide a configuration of (2) that extracts a row from the adjacency matrix. We describe the function as `read-A(X, Ã, C, D) = write(Ã[C,:], X[:,D])`, where X and $\tilde{A}$ are the input and padded adjacency matrices, while C and D are the row and column indices for the respective matrices. The input structure for this operation requires specialized inputs common to all algorithms presented in this work. This includes a set of columns to store the node-wise positional encodings, denoted as P. Additionally, a global variable representing the positional encoding of the node of interest is stored in a distinct set of columns, marked as P_{cur}. The structure of the input matrix X is given below:

$$
X = \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \begin{array}{cccccccc} & P & P_{\mathrm{cur}} & B_{\mathrm{global}} & B_{\mathrm{local}} & C & D & S \\ \left[\begin{array}{c} \cdots \\ \cdots \\ \vdots \\ \cdots \end{array}\right. & \begin{array}{c} 0 \\ p_1 \\ \vdots \\ p_n \end{array} & \begin{array}{c} p_i \\ 0 \\ \vdots \\ 0 \end{array} & \begin{array}{c} 1 \\ 0 \\ \vdots \\ 0 \end{array} & \begin{array}{c} 0 \\ 1 \\ \vdots \\ 0 \end{array} & \begin{array}{c} c_0 \\ c_1 \\ \vdots \\ c_n \end{array} & \begin{array}{c} d_0 \\ d_1 \\ \vdots \\ d_n \end{array} & \left.\begin{array}{c} \cdots \\ \cdots \\ \vdots \\ \cdots \end{array}\right] \end{array}.
$$

For simplicity, we let the embedding dimension $d_a = 2$, which corresponds to the dimension of positional encodings.

For the attention head for $\tilde{A}$ in (2), we define:

$$
(W_K, W_Q)_{i,j} = \begin{cases} 1 & \text{if } i \in \{(P)_j, (P_{\mathrm{cur}})_j\}, \ j = 1, 2 \\ 0 & \text{otherwise,} \end{cases}
$$

$$
(W_V)_{i,j} = \begin{cases} 2 & \text{if } i = B_{\mathrm{global}}, j = D \\ 0 & \text{otherwise.} \end{cases}
$$

The multiplication $X W_Q W_K^\top X^\top$ results in a matrix where each element is the inner product between positional encodings of nodes. Positional encodings are formulated such that $\langle p_i, p_i \rangle > \langle p_i, p_j \rangle$ whenever $i \neq j$ for $i, j \in [n]$. By leveraging this property, we allow the attention matrix to behave as an indicator function. The result is given below:

$$
\sigma\left(X W_Q W_K^\top X^\top\right) = \begin{bmatrix} 1/2 & 0 & \cdots & 1/2 & \cdots \\ 0 & 1 & \cdots & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \\ 1/2 & 0 & \cdots & 1/2 & \cdots \\ \vdots & \vdots & & \vdots & \ddots \\ 0 & 0 & \cdots & 0 & \cdots \end{bmatrix}.
$$

The result of the multiplication of the attention matrix and $\tilde{A}$ is further multiplied by the input X:

$$
\tilde{A}\,\sigma\left(X W_Q W_K^\top X^\top\right) X = {}^{1\!}/{}_{2} \begin{bmatrix} \cdots & 0 & \cdots \\ \cdots & A_{1i} & \cdots \\ & \vdots & \\ \cdots & A_{in} & \cdots \end{bmatrix} \overset{B_{\mathrm{global}}}{}.
$$

The result in column B_{global} is written in the target column D by W_V. Notice that in this case, the target field must also be erased. This can be done as in Section 5.2, using the f_{mlp} and an intermediate scratchpad field, or by leveraging another attention head to replicate the identity matrix and subtract the values in the target columns.

6. Theoretical analysis

In this section we demonstrate the ability of the architecture in (2) to simulate various graph algorithms. We provide empirical validation of the results in Appendix B.3. We discuss the implications of incorporating attention heads in the form of (1) and we demonstrate that the overall architecture is Turing Complete, even when constrained to constant width. We start by describing the positional encodings we use and other related parameters in our architecture.

6.1. Positional encodings and increment

In our approach, we use circular positional encodings to enumerate the nodes in the graph (Liu et al., 2023). This enumeration is carried out by discretizing the unit circle into intervals of angle $\hat{\delta}$[3]. Each node is then represented by a tuple of sine and cosine values.

---

[3]Numerical issues can arise when representing positions by evenly spaced angles around the unit circle. This is due to the

For node enumeration, we use the formula $p_i = R_{\hat{\delta}}^{\top} p_{i-1}$ for $i \geq 1$. In this context, $\hat{\delta}$ represents the smallest rotation that our system can represent with limited precision. This parameter essentially sets the limit on the number of nodes we can enumerate, which is capped at $\lfloor 2\pi \hat{\delta}^{-1} \rfloor$. This limit is determined by the precision with which we can represent $\hat{\delta}$, thus influencing the maximum number of distinct nodes that can be effectively encoded. This approach inherently compensates for rotational imprecision since the encodings are generated by the rotation matrix and not by the trigonometric functions of each angle. Although this could affect the periodicity of the function, our designs do not rely on this attribute. What is crucial is that the increment function can be executed by a neural network, and the positional encodings satisfy the condition: $\langle p_i, p_i \rangle > \langle p_i, p_j \rangle$ for $i \neq j$ (see Section 5.3). The rotation matrix $R_{\hat{\delta}}$ can be efficiently implemented as a single linear layer within f_{MLP}. Furthermore, since $R_{\hat{\delta}}$ has unitary singular values, all positional encodings have unitary norms, thus by the Cauchy-Schwarz Inequality, the inner product property is preserved.

6.2. Maximum absolute value parameter

All algorithms that we simulate utilize a conditional selection function. The implementation of this function depends on the parameter Ω, which represents the maximum absolute value within a clause. We approximate the conditional selection function in the following manner:

$$\mathtt{if\text{-}else}\,(c_1, c_0, \gamma) \approx \phi\,(c_0 - \gamma\,\Omega) \qquad (4)$$
$$+ \phi\,(c_1 - (1 - \gamma)\,\Omega)\,,$$

where $c_0, c_1 \in [-\Omega, \Omega]$ are the clause values and $\gamma \in \{0, 1\}$ is the condition. When $\gamma = 0$, the function outputs c_0, and c_1 otherwise. The complete construction also replicates the terms for the negative counterparts of the clauses. This approach relies on Ω to cancel any opposite terms in the conditional selection. Therefore, Ω determines the largest absolute value present in the clauses, and any element larger than Ω breaks the simulation.

6.3. Results on simulation of graph algorithms

We now present our results on transformer-based simulations of graph algorithms. We first present the simulation

---

limited precision in representing certain quantities, especially irrational numbers and rational numbers with repeating binary fractions. To mitigate this issue, we start by setting an initial angle p_0 as $(\sin 0, \cos 0) = (0, 1)$ and selecting a minimum increment angle, denoted as δ. We convert the angle δ into its nearest sine and cosine representation that can be exactly represented by the machine. We denote this approximate angle as $\hat{\delta}$ and construct the rotation matrix $R_{\hat{\delta}}$. Given that this matrix is derived from $\hat{\delta}$, it consists of quantities that can be represented by the machine. Importantly, we ensure that $R_{\hat{\delta}}$ maintains the fundamental attribute of orthogonality in the rotation matrix.

results for Dijkstra's algorithm on weighted graphs, accompanied by a brief overview of its proof. We then shift our focus to undirected graphs, examining algorithms such as Breadth-First Search, Depth-First Search, and the identification of Strongly Connected Components. Alongside these results, we provide an outline of the proofs for the related theorems. Lastly, we present our results on multitasking, which combines the constructions of Dijkstra's algorithm with those of Breadth-First Search and Depth-First Search, and discuss the principles behind this construction. The complete proofs are in the Appendix C.

Theorem 6.1. *There exists a looped-transformer h_T in the form of* (2), *with 17 layers, 3 attention heads, and layer width $O(1)$ that simulates Dijkstra's shortest path algorithm for weighted graphs with rational edge-weights, up to $O(\hat{\delta}^{-1})$ nodes and graph diameter of $O(\Omega \varepsilon)$.*

Proof Overview: Theorem 6.1 is proved by construction. Our proof begins by adapting Dijkstra's algorithm to fit our looping structure. This involves unrolling any nested operations to fit within a single loop, as described in Algorithm 1. Although these modifications change Dijkstra's algorithm from its traditional form, they do not alter its core functionality. In conventional programming, loops within the code must be completed before the rest of the code can continue. However, in the absence of additional structures, our unrolling approach would lead to all instructions being executed all at once. This could potentially disrupt the intended sequence of operations. To address this, we also implement binary flags and conditional selections, as explained in (4). These mechanisms are vital for ensuring that certain operations only modify data under specific conditions. A key example in our simulations is the minimum function which has $O(n)$ complexity and runs concurrently with the rest of the algorithm. When this function is active, it prevents modifications by other parts of the algorithm. The modifications can resume once the minimum function has been completed, at which point it becomes inactive. Finally, each step of the rewritten algorithm is implemented as a series of transformer layers in the form of (2), accounting for 17 layers, and three attention heads: two standard and one specialized over A. Some of the core simulation mechanisms of these functions are extensively discussed in Section 4 and Section 6, demonstrating that most of the constructions are based on previously described principles. Furthermore, Theorem 6.1 is restricted to rational edge weights due to the inherent limitations of finite precision in representing edge values. We reweight the edges to circumvent issues arising from the tolerance ε of the less-than function (3), which may not adequately capture minor path differences. This involves dividing all edge weights by the smallest absolute edge weight. Consequently, this ensures that the minimal non-zero path difference is effectively greater than one. The graph size is bounded by the enumeration constraints of $\hat{\delta}$,

as mentioned in Section 6.1. The dependence on the graph diameter is attributed to the maximum distance that can be constructed during the execution of Dijkstra's algorithm. Due to the algorithm's design, which excludes repeating or negative paths, the quantity is determined by the weighted graph's diameter. After the reweighting strategy, the diameter must remain below Ω, leading to a dependence of $O(\Omega\epsilon)$ on the original graph diameter.

We now present our results for algorithms on unweighted graphs, followed by a collective discussion of their proofs.

Theorem 6.2. *There exists a looped-transformer h_T in the form of* (2)*, with 17 layers, 3 attention heads, and layer width $O(1)$ that simulates Breadth-First Search for unweighted graphs with up to $\min(O(\hat{\delta}^{-1}), O(\Omega))$ nodes.*

Theorem 6.3. *There exists a looped-transformer h_T in the form of* (2)*, with 15 layers, 3 attention heads, and layer width $O(1)$ that simulates Depth-First Search for unweighted graphs with up to $\min(O(\hat{\delta}^{-1}), O(\Omega))$ nodes.*

Theorem 6.4. *There exists a looped-transformer h_T in the form of* (2)*, with 22 layers, 4 attention heads, and layer width $O(1)$ that simulates Kosaraju's Strongly Connected Components algorithm for unweighted graphs with up to $\min(O(\hat{\delta}^{-1}), O(\Omega))$ nodes.*

Proof Overview: Our proofs for Theorems 6.2-6.4 also follow a constructive approach. The implementations of breadth-first and depth-first search are comparable to Dijkstra's algorithm in terms of the number of layers and attention heads. However, the Strongly Connected Components algorithm utilizes significantly more layers and an additional attention head for A, due to its complexity and the need for a depth-first search over both A and $A^\top$. Despite the absence of edge weights, we encounter an additional bound of $O(\Omega)$ on the number of nodes. This stems from our method of replicating the behavior of queues and stacks, which are used in these algorithms. Similar to our approach with Dijkstra's algorithm, we also incorporate an interaction with a minimum function. During the execution of the algorithms, priority values are assigned to the neighbors of each node visited. These assigned values shape the minimum function's behavior: increasing priority values mimic the functionality of a queue while decreasing values replicate that of a stack. Hence, having more nodes in a graph increases the number of loop iterations, thereby increasing the absolute value of the priority value. Given that these priority values feature in conditional selections, they are also subject to the constraints imposed by Ω. Consequently, the graph's size limitations are independently determined by either $\hat{\delta}$ or Ω. In the implementation of Breadth-First Search (BFS) and Depth-First Search (DFS), since nodes only need to be visited once, Ω is the maximum node count for which accurate simulation of any graph is guaranteed. In the implementation of the Strongly Connected Components (SCC)

algorithm, which involves two distinct DFS operations, the maximum node count is constrained to $\Omega/3$. This specific limit arises from the two-phase process, and also the role of the first DFS in organizing nodes by their finishing times, which is further detailed in Appendix C.

We now present our results on multitasking, followed by a discussion of its construction. The following remark encapsulates the capabilities of our unified model.

Remark 6.5. There exists a looped transformer h_T in the form of (2), with 19 layers, 3 attention heads and layer width $O(1)$ that simulates (i) Depth-First Search and Breadth-First Search for unweighted graphs with up to $\min(O(\hat{\delta}^{-1}), O(\Omega))$ nodes; and (ii) Dijkstra's shortest path algorithm for weighted graphs with rational edge-weights with up to $O(\hat{\delta}^{-1})$ and graph diameter of $O(\Omega\varepsilon)$.

Our unified model can simulate any of the three algorithms with the right input configuration. Although the input structure remains constant across all three algorithms, it only needs minor modifications to specify the algorithm to be executed. This result leverages the construction components of the previous models and thus inherits the limitations presented by the previous results. These algorithms share common structures and functions, which are reused to avoid redundancy, as well as distinct components specific to each algorithm, such as variables and functions that need to be accommodated. For instance, all algorithms use the minimum function as well as a common termination criterion. The accommodation of unique functions for each algorithm is achieved through a conditional selection function. This function determines the variables to be updated, ensuring the execution reflects the intended algorithmic behavior.

Overall, while our guarantees are limited by parameters ε, Ω, and $\hat{\delta}$, an important question remains: What are the implications of using a graph that fails to meet assumptions in our theorems? This situation could result in inaccurate simulation outcomes or even a failure to meet the pre-established termination condition, thus failing to halt as expected.

6.4. Turing Completeness

We now present our result on Turing Completeness of the architecture in (2). It is important to note that using a graph convolution operation within the attention mechanism as shown in (1) might affect the model's expressiveness. We demonstrate that a transformer based on this structure is also Turing Complete. Following the methodology of Giannou et al. (2023), we demonstrate this claim by successfully simulating SUBLEQ, a single-instruction language that, with arbitrary memory resources, is proven to be Turing Complete (Mavaddat & Parhami, 1988).

Remark 6.6. There exists a looped-transformer h_T in the form of (2), which utilizes the modified attention head in

(1), with 11 layers, 3 attention heads, and layer width $O(1)$ that simulates SUBLEQ.

Proof Overview: We prove that the architecture in (2) simulates a modified version of SUBLEQ, which is also Turing Complete. This modification changes the memory structure of SUBLEQ, utilizing two memory blocks instead of one. The first block is the standard memory block as in Mavaddat & Parhami (1988) which can be read and altered. The second is a special read-only memory that only stores the adjacency matrix. We introduce this memory modification to reflect the fact that the architecture in (2) has two inputs, the data matrix X and the padded adjacency matrix $\tilde{A}$. The former is modified at each iteration of the looped transformer architecture, while the latter is only used in the convolution stage, and it never changes. The modified version of SUBLEQ utilizes data from the graph by reading the data from the second block and copying them to the first block. We show that this operation of reading and copying data can be simulated by our read operation in Section 5.3. The rest of SUBLEQ can be simulated using generic attention heads and MLPs similarly to Giannou et al. (2023). The complete proof is provided in Appendix C.

7. Training Limitations in Algorithm Simulation

In this section, we address the inherent challenges of recovering the constructive parameters that simulate the aforementioned algorithms through training. Despite demonstrating the existence of parameters capable of simulation, discovering them through gradient-based training is challenging.

This difficulty arises from the need to approximate discontinuous functions – such as conditional selection or the less-than function – with continuous functions. In such cases, using neural networks to capture the sharp transitions of discontinuities leads to severe ill-conditioning. Ill-conditioning manifests as a sharp and narrow region of the loss function around the desired solution, which hampers parameter recovery during training. Moreover, as empirically demonstrated in Appendix B.4, the finer the simulation parameters (e.g., the smaller ε is in Equation (3)), the stronger the ill-conditioning, thus making recovery increasingly difficult.

However, these findings do not undermine the potential of neural networks to solve such tasks. It simply highlights the issues related to simulating specific algorithms using neural networks. Alternatives to this problem are further discussed in (Hertrich & Sering, 2023), where the authors present algorithms for minimum spanning tree and maximum flow that circumvent the discontinuities associated with more conventional approaches like Kruskal's and Edmonds-Karp's algorithms, thus avoiding approximation issues.

We believe that, with current training methods, the solutions

that demonstrate good generalization capabilities may also indicate the existence of algorithms that can effectively solve these tasks without the use of conditional branching, which is primarily responsible for the observed discontinuities.

8. Conclusion and Future Work

We present a constructive approach where a looped transformer, combined with graph convolution operations, is used to simulate various graph algorithms. This demonstrates the potential of looped transformers for algorithmic reasoning on graphs. This architecture also proves to be efficient, as it has constant network width regardless of the input graph's size, allowing it to handle graphs of varying dimensions.

Outside the scope of simulation, studying looped transformers for graph algorithms within the Probably Approximate Correct (PAC) learning framework offers an exciting research direction. Specifically, the constructive parameters Ω, $\hat{\delta}$, and ε are essential in determining which graphs can be simulated. However, these parameters may also influence the learnability of algorithmic subroutines. An interesting research question involves determining the sample complexity of subroutines as a function of these constructive parameters. Investigating this could further reveal the relationship between the design of solutions and their learning effectiveness under the PAC framework.

Impact Statement

Our results demonstrate the potential of neural networks in handling complex reasoning tasks, a domain traditionally thought to be exclusive to more explicit classical computational approaches. By bridging this gap, our research contributes to the broader comprehension of neural network capabilities. While our work is theoretical, it is important to acknowledge that there may be societal implications stemming from our results. However, at this stage, we do not identify any specific consequences that must be specifically highlighted here.

Acknowledgments

References

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley,

Reading, Mass., 1974.

Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Arora, R., Basu, A., Mianjy, P., and Mukherjee, A. Understanding deep neural networks with rectified linear units. In *International Conference on Learning Representations*, 2018.

Bevilacqua, B., Nikiforou, K., Ibarz, B., Bica, I., Paganini, M., Blundell, C., Mitrovic, J., and Veličković, P. Neural algorithmic reasoning with causal regularisation. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

Cappart, Q., Chételat, D., Khalil, E. B., Lodi, A., Morris, C., and Veličković, P. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2022.

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 1989.

Dehghani, M., Djolonga, J., Mustafa, B., Padlewski, P., Heek, J., Gilmer, J., Steiner, A. P., Caron, M., Geirhos, R., Alabdulmohsin, I., et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning*, pp. 7480–7512, 2023.

Diao, C. and Loynd, R. Relational attention: Generalizing transformers for graph-structured tasks. In *The Eleventh International Conference on Learning Representations*, 2023.

Dijkstra, E. W. A note on two problems in connexion with graphs. *Numer. Math.*, 1959. doi: 10.1007/BF01386390.

Dudzik, A. J. and Veličković, P. Graph neural networks are dynamic programmers. *Advances in Neural Information Processing Systems*, 35:20635–20647, 2022.

Engelmayer, V., Georgiev, D., and Veličković, P. Parallel algorithms align with neural execution. In *The Second Learning on Graphs Conference*, 2023.

Friedman, D., Wettig, A., and Chen, D. Learning transformer programs. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Georgiev, D., Liò, P., Bachurski, J., Chen, J., and Shi, T. Beyond erdos-renyi: Generalization in algorithmic reasoning on graphs. In *NeurIPS 2023 Workshop: I Can't Believe It's Not Better Workshop: Failure Modes in the Age of Foundation Models*, 2023a.

Georgiev, D., Numeroso, D., Bacciu, D., and Liò, P. Neural algorithmic reasoning for combinatorial optimisation. In *The Second Learning on Graphs Conference*, 2023b.

Giannou, A., Rajput, S., Sohn, J.-Y., Lee, K., Lee, J. D., and Papailiopoulos, D. Looped transformers as programmable computers. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202, pp. 11398–11442, 2023.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Hertrich, C. and Sering, L. Relu neural networks of polynomial size for exact maximum flow computation. In *International Conference on Integer Programming and Combinatorial Optimization*, pp. 187–202. Springer, 2023.

Hertrich, C. and Skutella, M. Provably good solutions to the knapsack problem via neural networks of bounded size. *INFORMS journal on computing*, 35(5):1079–1097, 2023.

Hopfield, J. J. and Tank, D. W. "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8.

Ibarz, B., Kurin, V., Papamakarios, G., Nikiforou, K., Bennani, M., Csordás, R., Dudzik, A. J., Bošnjak, M., Vitvitskyi, A., Rubanova, Y., et al. A generalist neural algorithmic learner. In *Learning on Graphs Conference*, pp. 2–1, 2022.

Kaiser, Ł. and Sutskever, I. Neural gpus learn algorithms. In *International Conference on Learning Representations*, 2016.

Khan, S., Naseer, M., Hayat, M., Zamir, S. W., Khan, F. S., and Shah, M. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Lee, N., Sreenivasan, K., Lee, J. D., Lee, K., and Papailiopoulos, D. Teaching arithmetic to small transformers. In *The Twelfth International Conference on Learning Representations*, 2024.

Lin, H. W., Tegmark, M., and Rolnick, D. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 2017.

Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *arXiv preprint arXiv:2301.05062*, 2023.

Liu, B., Ash, J. T., Goel, S., Krishnamurthy, A., and Zhang, C. Transformers learn shortcuts to automata. In *The Eleventh International Conference on Learning Representations*, 2023.

Loukas, A. What graph neural networks cannot learn: depth vs width. In *International Conference on Learning Representations*, 2020.

Mavaddat, F. and Parhami, B. Urisc: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 25(4):327–334, 1988.

Moore, E. F. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959.

Numeroso, D., Bacciu, D., and Veličković, P. Dual algorithmic reasoning. In *The Eleventh International Conference on Learning Representations*, 2023.

Pérez, J., Barceló, P., and Marinkovic, J. Attention is turing-complete. *Journal of Machine Learning Research*, 22 (75):1–35, 2021.

Reed, S. and De Freitas, N. Neural programmer-interpreters. In *The Fourth International Conference on Learning Representations*, 2016.

Rodionov, G. and Prokhorenkova, L. Neural algorithmic reasoning without intermediate supervision. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Siegelmann, H. and Sontag, E. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50:132–150, 1995.

Smith, K. A. Neural networks for combinatorial optimization: a review of more than a decade of research. *Informs journal on Computing*, 11(1):15–34, 1999.

Tang, H., Huang, Z., Gu, J., Lu, B.-L., and Su, H. Towards scale-invariant graph-related problem solving by iterative homogeneous GNNs. *Advances in Neural Information Processing Systems*, 33:15811–15822, 2020.

Tay, Y., Bahri, D., Yang, L., Metzler, D., and Juan, D.-C. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pp. 9438–9447, 2020.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Veličković, P. and Blundell, C. Neural algorithmic reasoning. *Patterns*, 2(7), 2021.

Veličković, P., Badia, A. P., Budden, D., Pascanu, R., Banino, A., Dashevskiy, M., Hadsell, R., and Blundell, C. The CLRS algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pp. 22084–22102, 2022.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Veličković, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020.

Wei, C., Chen, Y., and Ma, T. Statistically meaningful approximation: a case study on approximating turing machines with transformers. *Advances in Neural Information Processing Systems*, 35:12071–12083, 2022a.

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., and Fedus, W. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022b. ISSN 2835-8856.

Weston, J., Chopra, S., and Bordes, A. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Xu, K., Li, J., Zhang, M., Du, S. S., ichi Kawarabayashi, K., and Jegelka, S. What can neural networks reason about? In *International Conference on Learning Representations*, 2020.

Yan, Y., Swersky, K., Koutra, D., Ranganathan, P., and Hashemi, M. Neural execution engines: Learning to execute subroutines. *Advances in Neural Information Processing Systems*, 33:17298–17308, 2020.

Yuan, K., Guo, S., Liu, Z., Zhou, A., Yu, F., and Wu, W. Incorporating convolution designs into visual transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 579–588, 2021.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S., and Kumar, S. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*, 2020.

Zhou, D.-X. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis*, 2020.

A. Additional related work

A substantial body of research has focused on testing the capability of neural networks to execute algorithms. Early efforts include models like Hopfield Networks (Hopfield & Tank, 1985), which were applied to combinatorial optimization tasks (Smith, 1999). Prior to the emergence of transformers, models such as Neural Turing Machines (NTMs) (Graves et al., 2014) and Memory Networks (Weston et al., 2014) integrated neural networks with external memory, showing strong performance in tasks like copying and sorting. Subsequent advances were seen by adapting NTMs for learning algorithmic tasks (Kaiser & Sutskever, 2016), incorporating a dynamic memory (Graves et al., 2016), and utilizing recurrent models to learn multiple programs (Reed & De Freitas, 2016). These advancements laid the groundwork for concepts used in transformers and this work. For instance, the addressing mechanism in Graves et al. (2014) is akin to the attention mechanism in transformers, as well as concepts like scratch space for intermediate computations, as proposed by Reed & De Freitas (2016).

In the more recent developments related to algorithmic reasoning in transformers, dedicated frameworks have emerged to represent and execute complex algorithmic tasks. An example of this is the Restricted Access Sequence Processing Language (RASP). RASP allows for the expression of sequences of steps in transformers and can be applied to tasks such as token counting and sorting. Along this line, Lindner et al. (2023) developed a compiler for translating between RASP and Transformer models, while Friedman et al. (2023) have designed modified Transformers that can be trained and subsequently converted to RASP instructions.

B. Implementation details

In this section, we describe aspects of the implementation and empirical verification of our simulations. As mentioned in Section 4, despite our constructions being based on hardmax, for the purpose of empirical validation and to better align with the implementation practices in transformers, we utilize the softmax function:

$$\sigma_S(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}},$$

where K represents the number of elements. When using the softmax in attention, we divide all parameters in W_Q and W_K by a temperature parameter T. The value of T is important as it determines the sharpness of the softmax operation. As shown in Giannou et al. (2023) and Liu et al. (2023), as the temperature gets arbitrarily small, the behavior of the softmax approximates that of the hardmax, which is used in our constructions.

However, it is important to acknowledge that in practice the simulation of all algorithmic steps is subject to numerical limitations due to floating-point arithmetic. For example, using the minimum angle $\hat{\delta}$ that a computer can represent accurately is impractical. The reason is that this quantity becomes overshadowed by subsequent operations that introduce imprecision, such as matrix multiplications, and by non-linear elements like the softmax function in the attention stage. In algorithmic tasks, simulations exhibit a very low tolerance for computational errors. If not properly managed, these errors can propagate, leading to inaccurate prediction or non-termination. To address this, we adopt a more conservative approach in setting parameters for $\hat{\delta}$ and ε as described in Appendix B.3. In addition, we incorporate rounding operations for both binary values and positional encodings, following the structure described in (2).

B.1. Rounding functions

We utilize rounding functions in-between the simulated algorithmic steps to ensure that the computation of binary values made by the transformer does not include additional errors that propagate along the simulation. To this end, we adapt the rounding operation presented by Giannou et al. (2023), which can be algebraically expressed as:

$$\text{round}(X[:,C]) \approx \eta^{-1}\left(\phi(X[:,C] - 1/2) - \phi(X[:,C] - (1/2 + \eta))\right), \tag{5}$$

where η is a small tolerance value. This implementation can also be done using the structure presented in (2), noting that the expression in (5) is equivalent to the `less-than` function presented in Section 5.2, with $\varepsilon = \eta$, and $X[:,D]$ equivalent to the quantity $1/2$.

In practice, the increment function described in Appendix C.1.2 introduces noise in the positional encodings. To counter this, we employ a rounding procedure for circular positional encodings. This involves using two transformer layers. The first

layer generates a binary vector based on the current positional encoding We then round this quantity utilizing the procedure described in (5), and finally obtain the corresponding positional encoding based on the index position using the last layer.

The first attention layer is set up as described in (8), where D is set to B_{global}, and S_{PE} denotes the target column in the scratchpad. Another attention head is used to clear the original scratchpad contents, as detailed in (9). The corresponding f_{MLP} in this layer implements Equation (5), with S_{PE} as the source column C.

In the second transformer layer, the matrices $W_K^{(1)}$, $W_Q^{(1)}$, and $W_V^{(1)}$ of the first attention head are defined as follows:

$$(W_K^{(1)}, W_Q^{(1)})_{i,j} = \begin{cases} \tilde{T}^{-1/2} & \text{if } i = S_{\text{PE}} \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(1)})_{i,j} = \begin{cases} 2 & \text{if } i = (P)_k, j = (E)_k, \ k \in 0,1 \\ 0 & \text{otherwise.} \end{cases}$$

In this layer, E denotes the target columns for inserting positional encodings, and $\tilde{T}$ is an annealed temperature parameter of softmax, which is set higher than in other implementations (e.g., $T = 10^{-5}$). This ensures that minor differences in this attention head are minimized, thereby reducing errors in reading positional encodings. Additionally, we use the configuration of (9) in another attention head to clear the contents of E and S_{PE}. Lastly, the implementation of f_{MLP}, as defined in (10), clears the last n entries of the target field E.

B.2. Writing prevention after termination

The constructions in the empirical section slightly differ from the ones presented in Appendix C due to an additional consideration. While the theoretical constructions require delivering accurate outputs upon reaching the termination condition, one could continue running the looped transformer after the termination flag is activated. To ensure that the solution is not modified post-termination, we introduce extra constraints in the form of flags that deactivate write permissions once the termination condition is met. These are articulated through algebraic expressions, easily integrating into the transformer implementation. In practice, the writing permission flag is not determined solely by the minimum function's termination flag (term_{min}) for the implementation of Theorem 6.1 and similar algorithms. Instead, we construct a flag defined as `write = term_min and (not term)`, which can be implemented as `X[:,write] = φ(X[:,term_min]-X[:,term])`. This condition ensures that once the algorithm reaches the termination state, writing is forbidden.

Furthermore, we employ a conditional selection mechanism – as explained in Appendix C.1.1 – to maintain the termination flag, which is activated if the terminating condition is met but the simulation continues running. This is performed by first moving the calculation of the termination condition to a temporary storage area, denoted as S_{term}. Then, at the end of each iteration, we execute the layer: `cond-select(X, S_term, term, S_term, term)`. This means that if the recalculated termination condition in S_{term} is true, then the `term` variable is updated; otherwise, it remains unchanged, effectively managing the looped transformer's operation post-termination.

B.3. Empirical validation

We validate our theoretical results using graphs in the CLRS Algorithmic Reasoning Benchmark (Veličković et al., 2022). This benchmark offers data created for multiple tasks, including sorting, searching, and geometry, among others. Additionally, it encompasses tasks specific to graphs, including those simulated in this study. These tasks are divided into segments with varying input sizes. For graph-related tasks, the training and validation sets include graphs with 16 nodes, while the test set contains graphs with 64 nodes. In Table 1, we show that our simulations achieve perfect accuracy across all tested instances,

Table 1. Accuracy over algorithmic tasks on CLRS

Split (size)	BFS	DFS	Dijkstra	SCC
Train (1000)	100%	100%	100%	100%
Validation (32)	100%	100%	100%	100%
Test (32)	100%	100%	100%	100%

where we also observe the same performance for the multitask model. In our empirical verifications, we consistently apply specific parameters: an angular increment δ set at 10^{-2}, a maximum value Ω of 10^5, and a temperature parameter T at 10^{-7}. Additionally, in the masking operations such as Appendix C.2.1, we address numerical imprecision during execution

by using a value that is an order of magnitude smaller than Ω. This precaution is crucial because minor inaccuracies that slightly increase these values above Ω could lead to cumulative errors in the conditional selection function, potentially impacting the integrity of our results.

Furthermore, beyond the practical considerations discussed in the previous sections, the implementation of breadth-first search in the CLRS benchmark slightly differs from the one presented in Algorithm 4. This is because, as mentioned in Veličković et al. (2022), the results of the algorithm are built on top of the concept of parallel execution, which implies that for algorithms such as breadth-first search, there is no concept of order in the discovery of neighbor nodes, as outlined in Cormen et al. (2022). Instead, the order of discovery in CLRS is established by the value of the node index. Therefore, to address this particularity, the implementation of the conditional selection function in step (9) of Algorithm 4 has another pair of clauses, which can be expressed by `cond-select(X, val`$_{\text{best}}$`, order, term`$_{\text{min}}$`, order)`. Essentially, this variation in the implementation is designed to capture the priority value from the parent node, thereby ensuring uniform priority assignment across all nodes within the same neighborhood level.

B.4. Ill-conditioning effects on recovering simulation parameters

In this section, we discuss the challenges that hinder the discovery of simulation parameters for any of the algorithms presented through training. We begin by examining an important distinction in simulation results: exact versus approximate representation. When analyzing the expressivity of neural networks, most results focus on their ability to approximate desired target functions (Cybenko, 1989; Hornik et al., 1989; Lin et al., 2017; Zhou, 2020; Yun et al., 2020). However, only certain functions can be *exactly* represented by neural networks, i.e., with no approximation error. In (Arora et al., 2018), the authors demonstrate that feedforward neural networks with ReLU activation functions are inherently limited to exactly representing continuous and piecewise-linear functions. This limitation does not prevent such neural networks from representing other functions, albeit only approximately. This issue extends beyond the specific architecture studied in (Arora et al., 2018), as it can be argued that any neural network using continuous functions can only approximate discontinuous functions. Our simulation results directly observe this limitation: the construction of discontinuous functions, such as if-else or less-than, are subject to a degree of approximation controlled by the parameters Ω and ε, as shown in Equation (4) and Equation (3), respectively.

Relying on approximations for discontinuous functions introduces a significant challenge: ill-conditioning. In discontinuous functions, small input increments can cause sharp changes in outputs. Capturing this in the linear operations of neural networks, which must approximate these sharp variations, leads to ill-conditioning. The more precise the approximation, the more ill-conditioned the solution becomes, directly affecting the discovery of such parameters through training. Even with strong supervision (Veličković et al., 2022; Xu et al., 2020) in the form of ground-truth inputs and outputs for every layer, recovering such approximations is challenging.

Figure 3. Condition number of linear layers in the constructions of the if-else (left) and less-than (right) functions. The x-axes indicate the value of the construction parameters Ω and ε of Equation (4) and Equation (3), respectively. A higher Ω improves the simulation quality for the if-else function, while a smaller ε improves the simulation quality for the less-than function.

To illustrate this, we use our construction of the if-else function (Equation (4)), controlled by the parameter Ω, which limits the highest absolute value in any clause, determining the range of the approximation. We then estimate the Hessian with respect to the loss function as a function of Ω, as shown in Figure 3. Analyzing the condition number of this Hessian provides insight into the sharpness of the region surrounding the optimal solution: a higher condition number indicates a sharper and narrower region around the local optimum, which complicates recovery during training.

Our empirical results show that once Ω exceeds 50, the condition number of the Hessian rises above 10^{16}. This indicates an optimization landscape so sharp that finding the solution through any learning routine becomes virtually impossible. To further support this claim, we conducted an additional experiment using the if-else function. We initialized the layer parameters with the construction values, adding a very small quantity of random noise (of magnitude 10^{-7}). Despite conducting numerous experiments, initializing the parameters very close to the optimal target, and utilizing various optimization strategies—including second-order methods—the parameters failed to converge to the target solution.

This experiment further illustrates that any modern architecture relying on continuous functions to approximate discontinuities will struggle to discover parameters through training that achieve perfect (or even good) scale generalization. The better the scale generalization required, the more severe the ill-conditioning becomes, making it increasingly difficult to discover good parameters.

C. Proofs of Theorems

C.1. Preparations for the proofs

In this section, we first introduce key coding primitives that are essential to all the algorithmic simulations presented. We illustrate these coding primitives through the algorithm of the minimum function. This function is designed to identify the smallest value in a list, along with its index and other pertinent information. As outlined in Section 6, for each simulated algorithm, we adapt their formulations to align with the looped architecture of the transformer. Although this architecture inherently does not support inner loops, we simplify the presentation of the pseudocode by using inner loops solely to depict functions that are operated in parallel by the transformer. As further depicted in Algorithms 3-6, the iteration of the minimum function described in Algorithm 2 is executed inside the main loop of the transformer, along with the other functions of the corresponding algorithms. The algorithm description is also accompanied by comments that enumerate the steps that are implemented using the layer formulation in (2). Each comment is also linked to the section that provides a more detailed explanation of its implementation.

Algorithm 2 Iteration of the minimum function (get_minimum)

Input: data x, size n *List of values to find the minimum*
Input: data x_{SCC}, size n

1: $\text{scc}_{\text{cur}}, \text{scc}_{\text{best}} = 0, 0$ *Only used for SCC*
2: $\text{idx}_{\text{cur}}, \text{val}_{\text{cur}} = 0, 0$
3: $\text{idx}_{\text{best}}, \text{val}_{\text{best}} = 0, \Omega$
4: $\text{visit}_{\text{min}} = $ array of size n
5: **for** $i = 1$ **to** n **do**
6:     $\text{visit}_{\text{min}}[i] = $ **false**
7: **end for**
8: $\text{term}_{\text{min}} = $ **false**

9: **if** term_{min} is **true then**
10:     $\cdots$
11:     $\text{term}_{\text{min}} = $ **false**   *(1) Re-initialize variables [C.1.1]*
12: **end if**
13: $\text{idx}_{\text{cur}} = \text{idx}_{\text{cur}} + 1$ *(2) Increment position [C.1.2]*
14: $\text{val}_{\text{cur}} = x[\text{idx}_{\text{cur}}]$ *(3) Read value at current position [C.1.3]*
15: $\text{scc}_{\text{cur}} = x_{\text{SCC}}[\text{idx}_{\text{cur}}]$
16: $\text{condition} = \text{val}_{\text{cur}} < \text{val}_{\text{best}}$ *(4) Compare values [C.1.4]*
17: **if** condition **then**
18:     $\text{val}_{\text{best}}, \text{idx}_{\text{best}} = \text{val}_{\text{cur}}, \text{idx}_{\text{cur}}$ *(5) Update variables [C.1.1]*
19:     $\text{scc}_{\text{best}} = \text{scc}_{\text{cur}}$
20: **end if**
21: $\text{visit}_{\text{min}}[\text{idx}_{\text{cur}}] = $ **true**   *(6) Visit current position [C.1.5]*
22: $\text{term}_{\text{min}} = $ **not** (**false** in $\text{visit}_{\text{min}}$)   *(7) Trigger termination [C.1.6]*

The lines [1-8] consist of the initialization of the data, while the remaining represent the algorithm executed inside the loop. Note that lines 1, 15, and 19 are highlighted in gray, and are only defined in the case of the Strongly Connected Components algorithm. As described in Section 5.1, the single variables (e.g. `val_cur`) occupy the top row, while lists (e.g. `visit_min`) occupy the last n rows of X. Furthermore, the implementation of indexes requires two different representations in the input matrix X.

In Algorithm 2, these correspond to `idx_cur` and `idx_best` as well as the variables for the Strongly Connected Component implementation. In X, these indices are represented both as integer numbers and positional encodings. The latter is vital for algorithm functionality, while integer numbers are key in the decoding phase, forming the output data. Consequently, ambiguous variables in proofs are referred to using their positional encoding. For example, if `idx_cur` points to the list's first element during execution, it is represented as $p_1 = (\sin(\theta_1), \cos(\theta_1))$ — the positional encoding for the first node. The integer number representation is emphasized when relevant. Finally, the comments that enumerate and describe the steps executed are the basis for the functions implemented by our architecture.

17

C.1.1. CONDITIONAL SELECTION: STEPS (1) AND (5)

This section details the implementation of the conditional selection functions outlined in steps (1) and (5) of Algorithm 2. We reference the structure of this implementation in the subsequent proofs. In our constructions, we express this operation as: `cond-select(X,V₁,V₀,C,E)` `= write(if-else(X[:,V₁], X[:,V₀], X[:,C]), X[:,E])`. Here, X is the input matrix; V_1 and V_0 are columns for clause values; C and E are the condition and target field, respectively. The approximation of the function `if-else` is shown in (4).

Next, we describe how this function is implemented. As with other implementations, we utilize a scratchpad area, denoted as S, to store intermediate calculations. Each column in this space is indexed by S_i. During implementation, we also introduce more abstract indices for the scratchpad, such as $S_{\text{idx}_{\text{cur}}}$. While it is understood that this index correlates to a specific integer number, we adopt this approach to enhance clarity and comprehension. By using such descriptive indexing, we aim to convey the intended purpose of each allocation in the scratchpad.

As also described in the implementation of the `less-than` function (see Section 5.2), we set all parameters to zero in the attention stage, only maintaining the residual connection. Then, for the values of the f_{MLP}, we define:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \left\{ \begin{smallmatrix} V_0, V_1, C, E, \\ B_{\text{global}}, B_{\text{local}} \end{smallmatrix} \right\}, \ j = i, \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \left\{ \begin{smallmatrix} (E,E),(V_0,S_1) \\ (V_1,S_2) \end{smallmatrix} \right\} \\ -\Omega & \text{if } (i,j) \in \left\{ \begin{smallmatrix} (C,S_1),(B_{\text{global}},S_2) \\ (B_{\text{local}},S_2) \end{smallmatrix} \right\} \\ \Omega & \text{if } i = C, j = S_2 \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(3)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \left\{ \begin{smallmatrix} (E,E),(S_1,S_1) \\ (S_2,S_1) \end{smallmatrix} \right\} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = E, j = S_1 \\ -1 & \text{if } i, j = E \\ 0 & \text{otherwise.} \end{cases} \qquad (6)$$

In the general construction of the conditional selection function, the first layer, $W^{(1)}$, functions as an identity operator. The second layer, $W^{(2)}$, constructs the arguments of ϕ as detailed in (4). The third layer, $W^{(3)}$, sums the terms post-ReLU, and the last layer, $W^{(4)}$ is designed to transfer the term, initially located in the scratchpad, to the specified column E. Our complete construction also accounts for the negative counterparts of the clauses. For simplicity, the adaptation for these counterparts is not included in the proofs, as it simply involves inverting the signs of the inputs V_0, V_1 and E in $W^{(1)}$ and $W^{(4)}$. While we present an implementation for a single pair of clauses V_0 and V_1, this function can accommodate additional variables following the same mechanism as in (4), as is done for steps (1) and (5) of Algorithm 2. For example, in the implementation of Algorithm 2, the conditional selection clause represented in step (5) also needs to update the integer representation of the index, therefore another pair of clauses is utilized in (5).

Essentially, the general implementation of the conditional selection function operates by choosing a value from two columns in the input matrix and inserting it into a designated field. However, the implementation varies in the context of the re-initialization function. In this scenario, the conditional selection is executed between the current values in the algorithm and their reset versions. Specifically for this function, the reset values must be generated in the first layer and then placed in the scratchpad, which assumes the role of the V_1 field in the aforementioned formulation. This adjustment ensures that the function appropriately handles the re-initialization process within the algorithmic framework.

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i = \{B_{\text{global}}, B_{\text{local}}, \text{idx}_{\text{cur}}, \text{val}_{\text{cur}}, ..., \text{visit}_{\min}\}, j = i \\ (p_0)_k & \text{if } i = (\text{idx}_{\text{cur}})_k, j = (S_{\text{idx}_{\text{cur}}})_k, \ k \in \{1,2\} \\ \Omega & \text{if } i = \text{val}_{\text{best}}, j = S_{\text{val}_{\text{best}}} \\ 0 & \text{if } i = \text{val}_{\text{cur}}, ..., \text{visit}_{\min}, j = S_{\text{zero}} \\ 0 & \text{otherwise.} \end{cases} \qquad (7)$$

In this implementation, the variable for the current index is reset to a specific value, $p_0 = (\sin(0), \cos(0)) = (0, 1)$. This value does not correspond to any node enumerated and is intentionally reserved in our constructions to represent the equivalent of the index 0 in integer numbers. This approach ensures that p_0 serves as a unique and identifiable starting point or reset state within the algorithm. For the remaining layers, the placeholder of field V_1 is then occupied by the targets of $W^{(1)}$, i.e. $S_{\text{idx}_{\text{cur}}}$, $S_{\text{val}_{\text{best}}}$ and S_{zero} in their corresponding operations. Moreover, it is important to note that regardless of the

established condition, the status of term$_{\text{min}}$ is invariably set to false. Consequently, this field is consistently cleared in every iteration, similar to the way the target field E is cleared.

C.1.2. INCREMENT POSITION: STEP (2)

After the re-initialization process, the first step that follows is obtaining the next index in the list. In this case, since we utilize circular positional encodings, all that is needed is to utilize a rotation matrix. The rotation matrix is predetermined by $\hat{\delta}$, as discussed in Section 6.1. We express this operation as `increment(X, C, D) = write(rotate(X[:, C]), X[:,D])`, where C and D are the source and target columns. In the implementation of step (2), both C and D are the variable `idx`$_{\text{cur}}$. Below, we detail the specific set of parameters used to implement this function. Again, in this case, we only utilize the entries of f_{MLP}, while the parameters of f_{attn} are set to zero.

$$
(W^{(1)})_{i,j} = \begin{cases} \cos(\hat{\delta}) & \text{if } i = (C)_k, \ j = (S_C)_k, \ k \in \{1,2\} \\ -\sin(\hat{\delta}) & \text{if } i = (C)_1, \ j = (S_C)_2 \\ \sin(\hat{\delta}) & \text{if } i = (C)_2, \ j = (S_C)_1 \\ 1 & \text{if } i \in D, \ j = i \\ 0 & \text{otherwise,} \end{cases}
$$

$$
(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{D, S_C\}, \ j = i \\ 0 & \text{otherwise,} \end{cases} \quad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = (S_C)_k, j = (D)_k, k \in \{1,2\} \\ -1 & \text{if } i \in D, j = i \\ 0 & \text{otherwise.} \end{cases}
$$

In this context, the initial entries of the matrix $W^{(1)}$ correspond to the rotation matrix $R_{\hat{\delta}}$. Additionally, this matrix includes a unique non-zero entry, specifically designed to reset the target field of the final layer. It is also important to note that the variables C and D are represented by a tuple of sine and cosine values, and as such, its coordinates must be accurately aligned. Moreover, the implementation must account for the fact that positional encodings can possess negative values. The parameters for these added entries are essentially the inverse of the signs present in $W^{(1)}$ and $W^{(4)}$. To facilitate this process, the implementation also incorporates additional scratchpad entries, which serve to temporarily hold these intermediate results.

C.1.3. READ VALUE: STEP (3)

Once the node index is updated, we need to retrieve its associated value from the input list, possibly accompanied by other relevant information. This operation can be expressed as `read-X(X, C, D, E) = write(X[C, D], X[1,E])`, where C and D denote the coordinates of the target entry and E is the placeholder for the target field. In a more basic form, this operation involves extracting information from a local variable and transferring it into a global variable. For the implementation of step (3), we utilize this function to extract the entry from the list of values (denoted by x in Algorithm 2) as well as from the list that encodes the indexes with integer numbers.

The principle for this operation is similar to the extraction of rows or columns of A, as presented in Section 5.3. We employ a standard attention mechanism (no graph convolution) leveraging the positional encoding of the idx$_{\text{cur}}$ variable to obtain the entries in the corresponding row.

$$
(W_K^{(1)}, W_Q^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(P)_j, (C)_j\}, \ j = 1,2 \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(1)})_{i,j} = \begin{cases} 2 & \text{if } i = D, j = E \\ 0 & \text{otherwise.} \end{cases} \tag{8}
$$

To ensure the target column E is prepared for the insertion of the desired value, we apply a distinct attention mechanism. This mechanism employs a standard attention head but sets the first entry of XW_K and XW_Q to p_0. This configuration guarantees that, when executing a hardmax operation, the attention matrix effectively functions as an identity matrix. Consequently, this facilitates the removal of any existing entries in the target column E.

The matrices $W_K^{(2)}$, $W_Q^{(2)}$, and $W_V^{(2)}$ are defined as follows:

$$
(W_K^{(2)}, W_Q^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \{((P)_1, 1), ((P)_2, 2), (B_{\text{global}}, 2)\} \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(2)})_{i,j} = \begin{cases} -1 & \text{if } i, j = E \\ 0 & \text{otherwise.} \end{cases} \tag{9}
$$

This results in the following attention matrix:

$$\sigma\left(XW_Q^{(2)}W_K^{(2)\top}X^\top\right) = \sigma\left(\left[\begin{array}{ccccc} p_0^\top p_0 & p_0^\top p_1 & \cdots & p_0^\top p_n \\ \hline p_1^\top p_0 & p_1^\top p_1 & \cdots & p_1^\top p_n \\ \vdots & \vdots & \ddots & \vdots \\ p_n^\top p_0 & p_n^\top p_1 & \cdots & p_n^\top p_n \end{array}\right]\right) = \left[\begin{array}{c|cccc} 1 & 0 & \cdots & 0 \\ \hline 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{array}\right]$$

By employing the value matrix W_V, we can clear the target column E to insert new values. It is important to note that the attention head in (8) also adds terms to the last n rows of E, which are undesirable. To counteract this, we use the parameters of the f_{MLP}:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i,j = E \\ -\Omega & \text{if } i = B_{\mathrm{global}}, j = E \\ 0 & \text{otherwise,} \end{cases} \quad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i,j = E \\ 0 & \text{otherwise,} \end{cases} \quad (W^{(4)})_{i,j} = \begin{cases} -1 & \text{if } i,j = E \\ 0 & \text{otherwise,} \end{cases} \quad (10)$$

In this construction, the first layer removes the top row entry of E, which is then passed along the second and third layers, and finally subtracted from the target, removing all entries in the bottom rows.

C.1.4. COMPARE VALUES: STEP (4)

In this stage, we focus on comparing the current value ($\mathtt{val_{cur}}$) with the best value found so far ($\mathtt{val_{best}}$). The implementation of this function is detailed in Section 5.2. In this context, the operation is expressed as $\mathtt{less\text{-}than(X,}$ $\mathtt{val_{cur},}$ $\mathtt{val_{best},}$ $\mathtt{condition)}$.

C.1.5. VISIT NODE: STEP (6)

Each iteration of the minimum function sets the current node as visited. This action is crucial for the function's termination criteria, which checks if all nodes have been visited.

Essentially, the role of this function is to assign a predefined value to a specific entry $(X)_{i,j}$. We express this operation as $\mathtt{write\text{-}row(X,\ C,\ D,\ E)} = \mathtt{write(X[0,D].e}_{X[0,C]}\mathtt{,\ X[:,E])}$. Here, e_i symbolizes the standard basis vector in an $n+1$ dimensional space. The variables C, D, and E in the expression represent columns that contain the row to be written, the writing content, and the target field, respectively. This writing procedure is analogous to the read function, previously described in Appendix C.1.3.

The matrices W_K, W_Q, and W_V are defined as follows:

$$(W_K^{(1)}, W_Q^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(P)_j, (C)_j\},\ j = 1,2 \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(1)})_{i,j} = \begin{cases} 2 & \text{if } i = D, j = E \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

In the application of step (6) of Algorithm 2, we set C to be $\mathtt{idx_{cur}}$, D to B_{global}, and E to the $\mathtt{visit_{min}}$ variable. Additionally, as also pointed out in Appendix C.1.3, the extra entries generated by the attention head in (11) must also be eliminated. Contrary to the read function, in this instance, the unwanted entry appears in the top row of the target field. To address this, we adopt the same strategy as in the read function, utilizing an additional attention head that functions as the identity function, deleting the top row entry using the global bias B_{global}.

$$(W_K^{(2)}, W_Q^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \{((P)_1, 1), ((P)_2, 2), (B_{\mathrm{global}}, 2)\} \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(2)})_{i,j} = \begin{cases} -1 & \text{if } i = B_{\mathrm{global}},\ j = E \\ 0 & \text{otherwise.} \end{cases}$$

To define f_{MLP}, we set all its parameters to zero and retain only the residual connection. This approach essentially disregards any contribution from f_{MLP}.

C.1.6. TERMINATE: STEP (7)

At the end of each step of the minimum function, as well as in the other algorithms, we must verify if all nodes have been visited. If all nodes are visited, this indicates termination, therefore the termination flag must be activated. This process can

be formulated as `all-one(X, C, E) = write(all(X[2:,C]), X[1,E])`, where `all` is a function indicating whether all elements are non-zero, while $X[2:, C]$ captures the last n rows of the column C.

We start by defining the attention stage:

$$(W_K^{(1)}, W_Q^{(1)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \{((P)_1, 1), ((P)_2, 2), (B_{\text{global}}, 2)\} \\ 0 & \text{otherwise,} \end{cases} \qquad (W_V^{(1)})_{i,j} = \begin{cases} -1 & \text{if } i = E, \ j = i \\ 1 & \text{if } i = B_{\text{global}}, \ j = E \\ 0 & \text{otherwise.} \end{cases}$$

The first attention head, defined as the read function in Appendix C.1.3, is tasked with clearing fields used in the writing process. Additionally, we define:

$$(W_K^{(2)})_{i,j} = \begin{cases} -1 & \text{if } i = B_{\text{global}} \\ 1 & \text{if } i = B_{\text{local}} \\ 0 & \text{otherwise,} \end{cases} \qquad (W_Q^{(2)})_{i,j} = \begin{cases} 1 & \text{if } i = B_{\text{global}} \\ -1 & \text{if } i = B_{\text{local}} \\ 0 & \text{otherwise,} \end{cases} \qquad (W_V^{(2)})_{i,j} = \begin{cases} \Omega & \text{if } i = C, \ j = E \\ -\Omega & \text{if } i = B_{\text{local}}, \ j = E \\ 0 & \text{otherwise.} \end{cases}$$

This results in the following attention matrix:

$$\sigma\left(X W_Q^{(2)} W_K^{(2)\top} X^\top\right) = \sigma\left(2 \begin{bmatrix} -1 & 1 & \cdots & 1 \\ \hline 1 & -1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & -1 & \cdots & -1 \end{bmatrix}\right) = \begin{bmatrix} 0 & 1/n & \cdots & 1/n \\ \hline 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{bmatrix} \tag{12}$$

The top row of the attention matrix is crucial as it aggregates the last n terms of the matrix X. In addition, since our construction only uses columns with null values at the top row, the remaining n rows in (12) do not add any contribution in any field. We first define the terms for f_{MLP}, followed by an overview of how this implementation simulates the desired behavior.

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i = E, \ j \in \{E, S_E\} \\ -1 & \text{if } i = E, \ j = S_{\tilde{E}} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{E, S_E, S_{\tilde{E}}\}, \ j = i \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{E, S_{\tilde{E}}\}, \ j = E \\ -1 & \text{if } i = S_E, \ j = E \\ 0 & \text{otherwise.} \end{cases}$$

In the implementation of the f_{MLP}, the entries for S_E and $S_{\tilde{E}}$ are used to remove the entries pre-MLP, in both positive and negative cases. In summary, the entire operation performs as follows: $\phi(1 - \Omega + \Omega/n \sum_i (X[i, C]))$. When the sum of elements in C is less than n, the term $-\Omega$ dominates, ensuring a resulting flag of zero. However, once all nodes are visited, $X[:, C]$ offsets the negative influence of Ω, thereby activating the flag.

C.2. Proof of Theorem 6.1

In this section, we introduce the constructions utilized to simulate Dijkstra's shortest path algorithm (Dijkstra, 1959). This algorithm aims to determine the shortest path between any node of a weighted graph and a given source node. To this end, we first present an overview of the implementation strategy, which accompanies the limitations that are described in Theorem 6.1. These are then followed by the specific construction of each step of the algorithm.

Implementation overview: Executing Dijkstra's algorithm consists of an iterative process that detects the node with the current shortest distance from the source node, updating the shortest path distances to neighboring nodes and marking the given node as visited. This process continues until all nodes have been visited. In our implementation, we simulate a minimum function that operates in the column that stores the distances. This operation is alternated with the main execution of the algorithm, that is, the section that reads the adjacency matrix and updates the distances. This is made possible by the conditional selection mechanism and the minimum function's termination flag, which conditions any writing operations in the main execution of the algorithm.

Specifications of the architecture: In the following pseudocode description, we show the steps required to execute the algorithm. Each of the steps has an associated transformer layer in the form of (2) that implements the corresponding routine. In the case of Dijkstra's algorithm, there is a total of 17 steps, and therefore our implementation requires 17 layers. Furthermore, throughout the implementation details of each step, no configuration uses parameters whose count scales with the size of the graph, thus accounting for constant network width.

Limitations: Beside the enumeration and the minimum edge limitations explained in Section 6, the diameter of the graph also imposes a limitation on the set of graphs our constructions can represent. Unlike the implementation of the other algorithms, the priority for selecting a node is determined by its current distance from the source node. As explained in Section 6.2, the constructions of conditional selection functions are limited by Ω, the largest absolute value that can be utilized in a conditional clause. This constructive limitation constrains the distances that can be represented. More specifically, the greatest distance between any pair of vertices in the graph is bounded by Ω. Because of the reweighing scheme discussed in Section 6, which guarantees that the smallest difference between any paths is lower bounded by ε, the reweighted edges will be greater than their original values. Therefore, bounding the greatest distance of the reweighted graph is equivalent to bounding the diameter of the original graph to be $O(\Omega\epsilon)$.

In the remaining parts of the proof, we start by presenting the revised version of the algorithm, followed by the construction of each step that has not been covered in previous algorithms.

Algorithm 3 Dijkstra's algorithm for shortest path

Input: integer start
Input: matrix A, size $n \times n$

1: prev, visit, dists, dists$_{\text{masked}}$, changes, is_zero, candidates = arrays of size n
2: **for** $i = 1$ **to** n **do**
3:     visit[i], dists[i], prev[i] = **false**, $\hat{\Omega}$, i
4: **end for**
5: visit[start] = 0
6: term = **false**
7: $\cdots$ *Initialization of min-variables*

8: **while** term is **false do**
9:     **for** $i = 1$ **to** n **do**
10:         **if** visit[i] is **true then**
11:             dists$_{\text{masked}}$[i] = Ω *(1) Mask visited nodes* [C.2.1]
12:         **else**
13:             dists$_{\text{masked}}$[i] = dists[i]
14:         **end if**
15:     **end for**

16:     get_minimum(dists$_{\text{masked}}$)   *(2-8) Find minimum value* [C.1]

17:     **if** term$_{\text{min}}$ is **true then**
18:         node = idx$_{\text{best}}$
19:         dist = val$_{\text{best}}$ *(9) Get minimum values* [C.2.2]
20:     **end if**
21:     candidates = A[node, :]   *(10) Get row of A* [C.2.3]
22:     **for** $i = 1$ **to** n **do**
23:         is_zero[i] = (candidates[i] $\leq$ 0)   *(11) Mark non-neighbors* [C.2.4]
24:     **end for**
25:     **for** $i = 1$ **to** n **do**
26:         candidates[i] = candidates[i] + dist   *(12) Build distances* [C.2.5]
27:     **end for**
28:     **for** $i = 1$ **to** n **do**
29:         changes[i] = candidates[i] < dists[i]   *(13) Identify updates* [C.2.6]
30:     **end for**
31:     **for** $i = 1$ **to** n **do**
32:         **if** term$_{\text{min}}$ is **false** and is_zero[i] is **true then**
33:             changes[i] = 0   *(14) Mask updates* [C.2.7]
34:         **end if**
35:     **end for**
36:     **for** $i = 1$ **to** n **do**
37:         **if** changes[i] is **true then**
38:             prev[i], dists[i] = node, candidates[i]   *(15) Update variables* [C.2.8]
39:         **end if**
40:     **end for**
41:     visit[node] = visit[node] + term$_{\text{min}}$ *(16) Visit node* [C.2.9]
42:     term = **not** (**false** in visit)   *(17) Trigger termination* [C.2.10]
43: **end while**
     return prev, dists

C.2.1. MASK VISITED NODES: STEP (1)

In the execution of Dijkstra's algorithm, visited nodes are excluded from future iterations. Our implementation achieves this by masking distances, ensuring that masked values are ignored during the minimum function execution.

Before delving into the parameter configuration of this layer, it is crucial to address the initialization of the `dists` variable, which stores current distances. In Dijkstra and other algorithms, this variable is initialized to $\hat{\Omega} = \Omega - \epsilon$. This setup ensures that, during the minimum function phase, there is a preference for unvisited nodes - that is, nodes not adjacent to any visited node at a certain point in the execution - over those already visited. Conversely, initializing all nodes to Ω would result in the masking operation treating unvisited and visited nodes equally, thereby disrupting the accuracy of the simulation.

Implementing the masking operation in our context closely resembles the conditional selection function in Appendix C.1.1, with a key difference in its initial setup. This variation primarily involves the definition of the first two layers, which we define as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \left\{ \substack{V_0,C,E, \\ B_{\text{global}}, B_{\text{local}}} \right\}, j = i \\ \Omega & \text{if } i = B_{\text{local}}, j = S_{V_1} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \left\{ \substack{(E,E),(V_0,S_1) \\ (S_{V_1},S_2)} \right\} \\ -\Omega & \text{if } (i,j) \in \left\{ \substack{(C, S_1),(B_{\text{global}}, S_2), \\ (B_{\text{local}}, S_2)} \right\} \\ \Omega & \text{if } i = C, j = S_2 \\ 0 & \text{otherwise,} \end{cases}$$

Unlike the approach outlined in (6), the masking operation does not have a V_1 counterpart already calculated. To this end, we first generate create a mask of magnitude Ω over all nodes, and insert it in a scratchpad space denoted by S_{V_1}. Then for the second layer, we simply redirect the definition presented in (6), from the dependency from V_1 to the dedicated column S_{V_1}. These adjustments are processed by the remaining layers of the implementation, ensuring the correct nodes are selected for further operations in the algorithm. For the application in step (1) of Algorithm 3, we set V_0 to `dists`, C to `visit`, and E to $\text{dists}_{\text{masked}}$.

C.2.2. GET MINIMUM VALUES: STEP (9)

The purpose of this function is to update the values `node` and `dist` once the minimum function terminates. To this end, we also utilize the formulation of the conditional selection function, expressing it as `cond-select(X, [idx`$_{\text{best}}$`, val`$_{\text{best}}$`], [node, dist], term`$_{\text{min}}$`, [node, dist])`. Note that in this formulation, the fields V_0, V_1, and E take in multiple values. The implementation for this case is similar to that presented in Appendix C.1.1, matching the order of the elements in each of these fields.

C.2.3. GET ROW OF THE ADJACENCY MATRIX: STEP (10)

Once the variable `node` is updated, the corresponding row of the adjacency matrix A must be extracted. To this end, we refer to the implementation described in Section 5.3, setting P_{cur} to `node`.

C.2.4. MARK NON-NEIGHBOR NODES: STEP (11)

In this operation, all non-neighbor nodes must be indicated by 1 in their corresponding coordinates of the `is_zero` variable. For this implementation, we set all attention parameters to zero, and define f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) = \{(B_{\text{local}}, S_{\text{zero}}), (\text{is\_zero}, \text{is\_zero})\} \\ -\Omega & \text{if } i = \text{candidates}, \ j = S_{\text{zero}} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i = \{S_{\text{zero}}, \text{is\_zero}\}, \ j = i \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_{\text{zero}}, \ j = \text{is\_zero} \\ -1 & \text{if } i, j = \text{is\_zero} \\ 0 & \text{otherwise.} \end{cases}$$

In effect, we create a list of ones, which are then subtracted by all non-zero entries of `candidates`, multiplied by $-\Omega$. This ensures that all non-neighbor nodes are set to one, while the remaining are zero. In addition, the existing entries in the variable `is_zero` are erased.

C.2.5. BUILD DISTANCES: STEP (12)

In this step, the distance to the current node must be combined with the weighted edges. To explain this process further, the operation copies a value from the top row across the bottom n rows of X. This operation can be formally expressed as `repeat-n(X, C, D) = write(`$\mathbf{1}_{n+1}$`X[1,C], X[:,D])`. Here, $\mathbf{1}_{n+1}$ is the all-ones vector of size $n + 1$, which is multiplied by the first entry of the column C. The result is then written in column D.

The first attention head is defined as:

$$(W_K^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i = B_{\text{global}} \\ 0 & \text{otherwise,} \end{cases} \quad (W_Q^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{B_{\text{global}}, B_{\text{local}}\} \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i = C, \ j = E \\ 0 & \text{otherwise.} \end{cases}$$

This configuration leads to the following attention matrix:

$$\sigma\left(XW_Q^{(1)}W_K^{(1)\top}X^\top\right) = \sigma\left(2 \begin{bmatrix} \begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{array} \end{bmatrix}\right) = \begin{bmatrix} \begin{array}{c|ccc} 1 & 0 & \cdots & 0 \\ \hline 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 0 \end{array} \end{bmatrix}. \tag{13}$$

Equation (13) demonstrates that the operation on X captures elements from the first row and distributes them across its subsequent rows The second attention head is defined as follows:

$$(W_K^{(2)}, W_Q^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \{((P)_1, 1), ((P)_2, 2), (B_{\text{global}}, 2)\} \\ 0 & \text{otherwise,} \end{cases} \quad (W_V^{(2)})_{i,j} = \begin{cases} -1 & \text{if } i = C, E, j = E \\ 0 & \text{otherwise.} \end{cases} \tag{14}$$

The attention head in (13) also generates unwanted values in the top row, which are eliminated by retrieving the column C that contains the top row value replicated earlier. As for the configuration of f_{MLP}, all its values are set to 0, only utilizing the residual connection.

For the specific application to step (12) in Algorithm 3, this is applied to val_{cur} so that its value is distributed along all candidates entries in X. However, since this field already contains the edges from the adjacent nodes, we set $(W_V^{(2)})_{(\text{val}_{\text{cur}}, \text{val}_{\text{cur}})} = 0$, effectively ignoring the deletion of the target field. Furthermore, notice that this operation builds the distance for all nodes, including both neighbors and non-neighbors. The subsequent step in Algorithm 3 addresses the non-neighbors by applying a masking technique. Moreover, two additional variables are replicated: the minimum function's termination flag, utilizing repeat-n(X, term_{min}, $\text{term}_{\text{min\_n}}$), as well as the integer representation of the current node, through repeat-n(X, node_{int}, $\text{node}_{\text{int\_n}}$).

The repeated integer representation of the node ($\text{node}_{\text{int\_n}}$) is employed in step (15) for updating the previously visited node. On the other hand, the repeated termination flag ($\text{term}_{\text{min\_n}}$) is utilized in step (16) to mask operations if the minimum function is still in progress.

C.2.6. IDENTIFY UPDATES (13)

In this step, the distances computed in candidates are compared against the current distances. To this end, we utilize the implementation described in Section 5.2 and write: less-than(X, candidates, dists, changes).

C.2.7. MASK UPDATES: STEP (14)

After performing the comparison, the variable changes indicates which nodes require updates. It is important to note that the comparison results should be set to false for non-neighbor nodes and when the termination flag of the minimum function is not active. We express this condition as follows: changes[i] = changes[i] and term_{min} and (not is_zero[i]).

In terms of implementation, we begin by setting the parameters of f_{attn} to zero and define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i,j \in \left\{ \begin{array}{c} (\text{changes, changes}), (\text{term}_{\text{min\_n}}, \text{changes}), \\ (\text{changes}, S_{\text{changes}}) \end{array} \right\} \\ -1 & \text{if } i \in \left\{ \begin{array}{c} \text{is\_zero}, \\ B_{\text{global}}, B_{\text{local}} \end{array} \right\}, \ j = \text{changes} \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{changes}, S_{\text{changes}}\}, j = i \\ 0 & \text{otherwise,} \end{cases} \quad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i,j = \text{changes} \\ -1 & \text{if } i = S_{\text{changes}}, j = \text{changes} \\ 0 & \text{otherwise.} \end{cases}$$

In this implementation, we write the necessary logical condition as $\phi(\texttt{changes[i]} + \texttt{term}_{\min} - \texttt{is\_zero[i]} - 1)$. This ensures the result equals 1 only when all conditions are satisfied. Additionally, in this implementation, the variable $\texttt{term}_{\min\_n}$ is determined in step (12) by repeating the status of the flag $\texttt{term}_{\min}$ along the last n rows of X.

C.2.8. UPDATE VARIABLES: STEP (15)

Once all the conditions have been imposed on the variable `changes`, the next step consists in updating the necessary values based on this decision variable. The expression in line [38] of Algorithm 3 is implemented as `cond-select(X,` `[node`$_{\texttt{int\_n}}$`, candidates], [prev, dists], changes, [prev, dists]`, utilizing the conditional selection function described in Appendix C.1.1. Here, the variable $\texttt{node}_{\texttt{int\_n}}$ is the list containing the repeated entries of the current node in its integer form, as executed in step (12).

C.2.9. VISIT NODE: STEP (16)

The principle for marking a node as visited follows the same implementation outlined in Appendix C.1.5. We write this function as: `write-row(X, node, term`$_{\min}$`, visit)`. It is important to note that in the application of the `write-row` function, we assign the variable D the variable $\texttt{term}_{\min}$. This assignment is crucial because it ensures that the visit function is activated only when the minimum function's termination flag is also active.

C.2.10. TERMINATE: STEP (17)

The termination function follows the implementation of Appendix C.1.6 by setting `all-one(X, visit, term)`.

C.3. Proof of Theorem 6.2

In this section, we introduce the constructions used to simulate the Breadth-First Search (BFS) algorithm (Moore, 1959). The objective of this algorithm is to systematically explore nodes in a graph, starting from a root node and moving outward to adjacent nodes and beyond. First, nodes directly adjacent to the root are visited, followed by their neighbors, and so on. We will begin with an overview of our implementation strategy, including its limitations as detailed in Theorem 6.2 Next, we present the specific constructions of each step of the algorithm.

Implementation overview: In BFS, node traversal is typically managed using a queue (Cormen et al., 2022). In this way, nodes nearer to the source node are discovered and queued for visitation first. In this process, we also record their immediate predecessor, i.e., the node through which it was discovered. This continues until all nodes connected to the source node have been visited. Our implementation of BFS simulates a queue using a priority list combined with a minimum function, similar to the approach in Appendix C.2. Each time a node is selected by the minimum function, the priority value increases, thereby assigning lower priority to subsequent nodes discovered through it. This strategy effectively emulates the first-in-first-out nature of a traditional queue.

Specifications of the architecture: We outline the algorithm's execution steps in the Algorithm 4. Each step correlates with a transformer layer, as represented in Equation (2), executing the respective routine. In the case of the Breadth-First Search, there is a total of 17 steps, and therefore our implementation requires 17 layers. Furthermore, throughout the implementation details of each step, no configuration uses parameters whose count scales with the size of the graph, ensuring constant network width.

Limitations: In addition to the enumeration limitation discussed in Section 6, simulation is also constrained by the largest priority value that can be utilized. A node is visited only once, and the priority value increments with each selection. Hence, for a graph of size n, the priority can grow to a maximum of n as well As detailed in Section 6.2, the conditional selection functions are limited by Ω, the highest absolute value in a conditional clause. This limitation restricts the maximum usable value and consequently the largest graph size that can be simulated. Specifically, based on conditional selection limits, the largest feasible graph is bounded by $O(\Omega)$

In the remaining parts of the proof, we start by presenting the revised version of the algorithm, followed by the construction of each step that has not been covered in previous algorithms.

Algorithm 4 Breadth-first search

    Input: integer start
    Input: matrix A, size $n \times n$

  1: order = 0
  2: prev, visit, orders, orders$_{\text{masked}}$, changes, disc = arrays of size n
  3: **for** $i = 1$ **to** n **do**
  4:    visit[i], disc[i], orders[i], prev[i] = **false**, **false**, $\hat{\Omega}$, i
  5: **end for**
  6: visit[start] = order
  7: term = **false**
  8: $\cdots$ *Initialization of min-variables*

  9: **while** term is **false do**
10:   **for** $i = 1$ **to** n **do**
11:     **if** visit[i] is **true then**
12:       orders$_{\text{masked}}$[i] = Ω *(1) Mask visited nodes* [C.2.1]
13:     **else**
14:       orders$_{\text{masked}}$[i] = orders[i]
15:     **end if**
16:   **end for**

17:   get_minimum(orders$_{\text{masked}}$)   *(2-8) Find minimum value* [C.1]

18:   **if** term$_{\text{min}}$ is **true then**
19:     node = idx$_{\text{best}}$ *(9) Get minimum value* [C.2.2]
20:   **end if**
21:   A$_{\text{row}}$ = A[node,:]   *(10) Get row of A* [C.2.3]
22:   order = order + term$_{\text{min}}$ *(11) Update priority factor* [C.3.1]
23:   **for** $i = 1$ **to** n **do**
24:     change = term$_{\text{min}}$ is **true and** disc[i] is **false**
25:     changes[i] = change is **true and** A$_{\text{row}}$[i] is 1   *(12) Identify updates* [C.3.2]
26:   **end for**
27:   **for** $i = 1$ **to** n **do**
28:     **if** changes[i] is **true then**
29:       prev[i] = node   *(13) Update variables* [C.2.8]
30:       orders[i] = order
31:     **end if**
32:   **end for**
33:   visit[node] = visit[node] + term$_{\text{min}}$
34:   disc[node] = disc[node] + term$_{\text{min}}$
35:   **for** $i = 1$ **to** n **do**
36:     disc[i] = disc[i] or changes[i]   *(14) Visit and discover nodes* [C.3.3]
37:   **end for**
38:   interrupt = disc == visit   *(15) Compare visited and discovered nodes* [C.3.4]
39:   term = **not** (**false** in visit)   *(16) Trigger termination/interruption* [C.3.5]
40:   term = (term or interrupt) and term$_{\text{min}}$ *(17) Trigger termination by interruption* [C.3.6]
41: **end while**
    return prev

C.3.1. UPDATE PRIORITY FACTOR

In this stage, as outlined in Algorithm 4, the priority factor, denoted by `order`, needs to be increased. This adjustment ensures that updated nodes are assigned a lower priority in the list. It is important to note that `order` is only modified upon the activation of the minimum function's termination flag, which guarantees its update only when necessary.

Furthermore, the priority factor is utilized in the update step (13), so its value must be made available to all nodes. This means that not only this value is increased, but also distributed for the local variables, in a variable we denote as `order`$_{\text{n}}$.

This implementation can be seen as a direct application of the `repeat-n` operation, detailed in Appendix C.2.5, being expressed as `repeat-n(X, [term`$_{\text{min}}$`, order], [order`$_{\text{n}}$`, order`$_{\text{n}}$`])`. Furthermore, to ensure that the updated value is kept for the next iterations we set $(W_V^{(2)})_{(\text{term}_{\text{min}}, \text{order})} = 1$ in the definition of (14) Furthermore, as in the application

of Appendix C.2.5 to Dijkstra's algorithm, the termination flag for the minimum function as well as the integer representation of the current node are also repeated along the last n rows.

C.3.2. IDENTIFY UPDATES: STEP (12)

In this step, we want to prevent any changes for non-neighbor nodes (i.e. $\mathtt{A_{row}[i]}$=0), discovered nodes (i.e. $\mathtt{disc[i]}$=1), or if the minimum function's termination flag ($\mathrm{term_{min}}$) is deactivated. This step closely follows the implementation of the masking function described in Appendix C.2.7.

However, in this case, we implement the following logical expression: $\mathtt{changes[i]}$ = $\mathtt{A_{row}[i]}$ and $\mathrm{term_{min}}$ and $\mathtt{(not\ disc[i])}$. We define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i,j \in \{(\mathrm{A_{row}}, \text{changes}), (\mathrm{term_{min\_n}}, \text{changes}), (\text{changes}, S_{\text{changes}})\} \\ -1 & \text{if } i \in \{B_{\text{global}}, B_{\text{local}}, \text{disc}\},\ j = \text{changes} \\ 0 & \text{otherwise}, \end{cases}$$

$$(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{changes}, S_{\text{changes}}\}, j = i \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i,j = \text{changes} \\ -1 & \text{if } i = S_{\text{changes}}, j = \text{changes} \\ 0 & \text{otherwise}. \end{cases}$$

In this implementation, we write the logical condition as $\phi(\mathtt{A_{row}[i]} + \mathrm{term_{min\_n}}\mathtt{[i]} - \mathtt{disc[i]} - 1)$. This ensures the result equals 1 only when all conditions are satisfied. Additionally, in this implementation, the variable $\mathrm{term_{min\_n}}$ is determined in step (11) by repeating the status of the flag $\mathrm{term_{min}}$ along the last n rows of X, as described in Appendix C.3.1.

C.3.3. VISIT AND DISCOVER NODES: STEP (14)

In this step, both discovered and visited nodes are updated. This implementation closely follows Appendix C.1.5, where we write $\mathtt{write\text{-}row(X, node, term_{min}, visit)}$ and $\mathtt{write\text{-}row(X, node, term_{min}, disc)}$. The divergence from the implementation of Appendix C.1.5 is that, for the definition of f_{MLP} we let:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i = \text{changes}, j = \text{disc} \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(2,3,4)})_{i,j} = \begin{cases} 1 & \text{if } i,j = \text{disc} \\ 0 & \text{otherwise}, \end{cases}$$

This extra modification ensures that all nodes that were changed in the previous step are also marked as discovered. In addition, since the condition for change requires nodes not to have been discovered, this formulation does not overcount discovered nodes.

C.3.4. COMPARE VISITED AND DISCOVERED NODES: STEP (16)

The principle of utilizing a priority list to determine which nodes to visit introduces a challenge in the simulation of the algorithm: the scenario when the graph is disconnected. As it stands, without any interruption mechanisms the implementation described in Algorithm 4 would erroneously visit nodes disconnected from the chosen root node, compromising the integrity of the simulation. To resolve this, we introduce an interruption criterion to identify when all nodes connected to the root have been visited.

This is achieved by comparing the list of visited nodes with the list of discovered nodes. After the first node is selected by the minimum function, a connected sub-graph is entirely visited if its list of visited nodes is the same as the discovered nodes. In this step, we examine whether each element in the $\mathtt{visit}$ variable corresponds to its counterpart in the $\mathtt{disc}$ variable. To this end, we set the attention parameters to zero and define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \left\{ \begin{array}{l} (\text{visit}, S_1), (B_{\text{local}}, B_{\text{local}}), \\ (\text{disc}, S_2), (\text{equal}, \text{equal}) \end{array} \right\} \\ -1 & \text{if } (i,j) \in \{(\text{visit}, S_2), (\text{disc}, S_1)\} \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \{(B_{\text{local}}, S_1), (\text{equal}, \text{equal})\} \\ -1 & \text{if } i \in \{S_1, S_2\}, \ j = S_1 \\ 0 & \text{otherwise}, \end{cases}$$

$$(W^{(3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{equal}, S_1\}, \ j = i \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_1, \ j = \text{equal} \\ -1 & \text{if } i, j = \text{equal} \\ 0 & \text{otherwise}. \end{cases}$$

In this operation, we compute the subtraction of `visit` minus `disc` and also `disc` minus `visit`. Ideally, if the two lists are identical, both subtractions should result in zero, leaving only the bias term introduced. The outcome of these computations is then stored in a variable named `equal`, which we utilize in the next step of the operation.

C.3.5. COMPUTE TERMINATION AND INTERRUPTION CRITERIA

The termination operation is described in the same form as Appendix C.1.6, and expressed as: `all-one(X, visit, term)`. A key addition to this calculation is the inclusion of the condition for the `equal` variable, computed in the previous step and described in Appendix C.3.4. To streamline the process, we consolidate the comparison done in the previous step into a single variable, denoted `interrupt`. Consequently, we write this expression as `all-one(X, equal, interrupt)`.

C.3.6. COMBINE INTERRUPTION AND TERMINATION FLAGS

After calculating both the interruption and termination flags, the final step is to combine them using the following logic: `term = (term or interrupt) and term_min`. The extra condition on the termination of the minimum function ensures that the simulation does not terminate in the first iteration, since in that stage, `visit` and `disc` list are both initialized with zeros. We implement this condition by setting the parameters of f_{attn} to zero, and we define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \left\{ \begin{array}{l} (\text{term}, \text{term}), (B_{\text{global}}, B_{\text{global}}), \\ (\text{interrupt}, S_1), (\text{term}_{\text{min}}, \text{term}_{\text{min}}) \end{array} \right\} \\ -1 & \text{if } i = \text{term}, \ j = S_1 \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in \left\{ \begin{array}{l} (\text{term}, \text{term}), (S_1, S_1), \\ (\text{term}, S_1), (\text{term}_{\text{min}}, S_1) \end{array} \right\} \\ -1 & \text{if } i = B_{\text{global}}, \ j = S_1 \\ 0 & \text{otherwise}, \end{cases}$$

$$(W^{(3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{term}, S_1\}, \ j = i \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } = S_1, \ j = \text{term} \\ -1 & \text{if } i, j = \text{term} \\ 0 & \text{otherwise}. \end{cases}$$

This above construction essentially realizes the intended logical condition through the following operation: `term =` $\phi(\phi(\texttt{interrupt} - \texttt{term}) + \texttt{term} + \texttt{term}_{\texttt{min}} - 1)$. Here, the inner ReLU function, when added to `term`, calculates the logical disjunction, and the sum of this result with the other terms forms the input for the second ReLU function, thereby computing the logical conjunction.

C.4. Proof of Theorem 6.3

This section outlines the constructions used for simulating the Depth-First Search (DFS) algorithm (Moore, 1959). DFS traverses nodes from the most recently discovered node, aiming to visit all nodes by progressing from one branch to its end before moving to another branch. Initially, we provide an overview of our DFS implementation approach, addressing its constraints as specified in Theorem 6.3. Next, we present the specific constructions of each step of the algorithm.

Implementation overview: One alternative to avoid the recursive formulation of DFS is to use a stack guided by node discovery order (Cormen et al., 2022). In this way, nodes are explored in the order they are found, recording their immediate predecessor. This continues until all nodes connected to the source node have been visited. Our implementation of DFS simulates a stack using a priority list combined with a minimum function, similar to the approach in Appendix C.3. The priority decreases with each node selection via the minimum function, prioritizing newly discovered nodes, thus replicating the last-in-first-out property of a stack.

Specifications of the architecture: We outline the algorithm's execution steps in the Algorithm 5. Each step correlates with a transformer layer, as represented in Equation (2), executing the respective routine. In the case of the Depth-First Search, there is a total of 15 steps, and therefore our implementation requires 15 layers. Furthermore, throughout the implementation details of each step, no configuration uses parameters whose count scales with the size of the graph, ensuring constant network width.

Limitations: In addition to the enumeration limitation discussed in Section 6, simulation is also constrained by the largest priority value that can be utilized. A node is visited only once, and the priority value decreases with each selection. Hence, for a graph of size n, the absolute priority value can grow to a maximum of n as well As detailed in Section 6.2, the conditional selection functions are limited by Ω, the highest absolute value in a conditional clause. This limitation restricts the maximum usable value and consequently the largest graph size that can be simulated. Specifically, based on conditional selection limits, the largest feasible graph is bounded by $O(\Omega)$

In the remaining parts of the proof, we start by presenting the revised version of the algorithm, followed by the construction of each step, particularly those not covered in earlier algorithms.

Algorithm 5 Depth-first search

Input: integer start
Input: matrix A, size $n \times n$

1: order = 0
2: prev, visit, orders, orders$_{masked}$, changes = arrays of size n
3: **for** $i = 1$ **to** n **do**
4:    visit[i], orders[i], prev[i] = **false**, $\hat{\Omega}$, i
5: **end for**
6: visit[start] = order
7: term = **false**
8: $\cdots$ *Initialization of min-variables*

9: **while** term is **false do**
10:   **for** $i = 1$ **to** n **do**
11:     **if** visit[i] is **true then**
12:       orders$_{masked}$[i] = Ω *(1) Mask visited nodes* [C.2.1]
13:     **else**
14:       orders$_{masked}$[i] = orders[i]
15:     **end if**
16:   **end for**

17:   get_minimum(orders$_{masked}$)   *(2-8) Find minimum value* [C.1]

18:   **if** term$_{min}$ is **true then**
19:     node = idx$_{best}$ *(9) Get minimum value* [C.2.2]
20:   **end if**
21:   A$_{row}$ = A[node,:]   *(10) Get row of A* [C.2.3]
22:   order = order - term$_{min}$ *(11) Update priority factor* [C.4.1]
23:   visit[node] = visit[node] + term$_{min}$ *(12) Visit node* [C.2.9]
24:   **for** $i = 1$ **to** n **do**
25:     change = term$_{min}$ is **true and** visit[i] is **false**
26:     changes[i] = change is **true and** A$_{row}$[i] is 1   *(13) Identify updates* [C.4.2]
27:   **end for**
28:   **for** $i = 1$ **to** n **do**
29:     **if** changes[i] is **true then**
30:       prev[i] = node   *(14) Update variables* [C.2.8]
31:       orders[i] = order
32:     **end if**
33:   **end for**
34:   term = **not** (**false** in visit)   *(15) Trigger termination* [C.2.10]
35: **end while**
    return prev

C.4.1. UPDATE PRIORITY FACTOR: STEP (11)

In this step, the priority factor, indicated by order in Algorithm 5, is reduced. This adjustment ensures that updated nodes receive a higher priority than the remaining in the priority list, effectively replicating the behavior of a stack.

The implementation of this routine is largely based on the construction detailed in Appendix C.3.1 used in the Breadth-First Search (BFS) algorithm, with a few key modifications. Specifically, we alter the values in the construction as follows: $(W^{(1)}V)$termmin, ordern = -1 and $(W^{(2)}V)$term$_{min}$, order = -1. These modifications are important as they enable the priority factor to decrease with each iteration of the main algorithm.

C.4.2. IDENTIFY UPDATES (13)

This step follows the same implementation of Appendix C.3.2, replacing the variable disc with visit.

C.5. Proof of Theorem 6.4

In this section, we introduce the constructions utilized to simulate Kosaraju's Strongly Connected Components (SCC) algorithm (Aho et al., 1974). This algorithm aims to detect the strongly connected components of a graph. A strongly connected component in a graph is a subgraph in which every vertex is reachable from every other vertex through directed paths. To this end, we first present an overview of the implementation strategy, which accompanies the limitations that are described in Theorem 6.4. These are then followed by the specific construction of each step of the algorithm.

Implementation overview: Executing Kosaraju's SCC algorithm first consists of performing a depth-first search on the source node. To implement this, we create a priority list whose updated values get decreased at each iteration, effectively replicating the behavior of a stack. For the current node in the execution, we check whether its neighbors have been visited. If so, the corresponding value on the second priority list is updated. This procedure simulates the insertion of nodes in a stack based on their finish times, as described in Kosaraju's SCC algorithm.

This process continues until each node has its values updated in the second priority list, which can take as much as $2n$ if the graph is a path. The end of this process triggers a partial termination criterion, which activates the second phase of the algorithm. In the second part, we perform another depth-first search. However, this time, the search is determined by the priority set in the previous phase, and it is performed on the transposed version of the graph, i.e. with the edges in the reverse direction. The index to indicate a strongly connected component is the source node of the breadth-first search of each subgraph. This process continues until all nodes have been visited, for a total of $3n$ node visits.

Specifications of the architecture: In the following pseudocode description, we show the steps required to execute the algorithm. Each of the steps has an associated transformer layer in the form of (2) that implements the corresponding routine. In the case of Kosaraju's Strongly Connected Components algorithm, there is a total of 22 steps, and therefore our implementation requires 22 layers. Furthermore, throughout the implementation details of each step, no configuration uses parameters whose count scales with the size of the graph, thus accounting for constant network width.

Limitations: Beside the enumeration limitations explained in Section 6, the number of node updates also determines the size of the graph that can be simulated. This is because the update process is governed by a single variable (denoted `order` in Algorithm 6) that decreases every update. As discussed in Section 6, since the constructions are limited to a certain size in the input, one of the restrictions of simulation is that the graph size can have at most $\Omega/3$ nodes.

In the remaining of the proof, we start by presenting the revised version of the algorithm, followed by the construction of each step that has not been covered in previous algorithms.

Algorithm 6 Strongly Connected Components (SCC) Algorithm

Input: integer start
Input: matrix A, size $n \times n$

1: term, $\text{term}_{\text{part}}$, order, node_{ref} = **false**, **false**, 0, 0
2: $\text{orders}_{\text{masked}}$, changes_1, changes_2, changes_3, nvisit = arrays of size n
3: sccs = array of size n, init. $[1, \dots, n]$
4: visit_1, visit_2, visit_3 = arrays of size n, init. **false**
5: orders_1, orders_2 = arrays of size n, init. $\hat{\Omega}$
6: $\text{orders}_1[\text{start}]$, $\text{orders}_2[\text{start}]$ = order, order
7: $\cdots$ *Initialization of min-variables*

8: **while** term is **false do**
9:     orders = orders_2 **if** $\text{term}_{\text{part}}$ is **true else** orders_1
10:     visit = visit_3 **if** $\text{term}_{\text{part}}$ is **true else** visit_2 *(1) Select list to find minimum* [C.5.1]
11:     **for** $i = 1$ **to** n **do**
12:        $\text{orders}_{\text{masked}}[i] = \Omega$ **if** visit[i] is **true else** orders[i]   *(2) Mask visited nodes* [C.2.1]
13:     **end for**

14:     get_minimum($\text{orders}_{\text{masked}}$, sccs)   *(3-9) Find minimum value* [C.1]

15:     $\text{write}_1 = \text{term}_{\text{min}}$ **and not** $\text{term}_{\text{part}}$
16:     $\text{write}_2 = \text{term}_{\text{min}}$ **and** $\text{term}_{\text{part}}$ *(10) Update write flags* [C.5.2]
17:     **if** term_{min} is **true then**
18:        node, scc = idx_{best}, scc_{best} *(11) Get minimum values* [C.2.2]
19:     **end if**
20:     A_{row}, A_{col} = A[node,:], A[:, node]   *(12) Get row/column of A* [C.5.3]
21:     $\text{visit}_1[\text{node}]$ = visit[node] + write_1
22:     $\text{visit}_3[\text{node}]$ = visit[node] + write_2 *(13) Visit nodes* [C.5.4]
23:     **for** $i = 1$ **to** n **do**
24:        nvisit = ($\text{visit}_1[i]$ **and** $A_{\text{row}}[i]$ is 1) **or** $A_{\text{row}}[i]$ is 0
25:        nvisit[i] = nvisit **and** write_1 is **true**   *(14) Check visited neighbors* [C.5.5]
26:     **end for**
27:     $\text{nvisit}_{\text{all}}$ = **not** (**false** in nvisit)   *(15) Check if all neighbors are visited* [C.5.6]
28:     $\text{visit}_{\text{scc}}$ = $\text{visit}_3[\text{scc}]$ *(16) Check if SCC is visited* [C.5.7]
29:     node_{ref} = scc **if** $\text{visit}_{\text{scc}}$ **else** node_{ref} *(17) Update reference node* [C.1.1]
30:     **for** $i = 1$ **to** n **do**
31:        $\text{changes}_1[i]$ = write_1 **and** ($A_{\text{row}}[i]$ is 1) **and not** $\text{visit}_1[i]$
32:        $\text{changes}_3[i]$ = write_2 **and** ($A_{\text{col}}[i]$ is 1) **and not** $\text{visit}_3[i]$
33:     **end for**
34:     $\text{changes}_2[\text{node}]$ = $\text{nvisit}_{\text{all}}$ **and** write_1
35:     order = order - term_{min} *(18) Identify updates and update priority* [C.5.8]
36:     **for** $i = 1$ **to** n **do**
37:        $\text{orders}_1[i]$ = order **if** $\text{changes}_1[i]$ **else** $\text{orders}_1[i]$ *(19) Update variables I* [C.2.8]
38:        $\text{orders}_2[i]$ = order **if** $\text{changes}_2[i]$ **else** $\text{orders}_2[i]$
39:        $\text{visit}_2[i]$ = $\text{visit}_1[i]$ **if** $\text{changes}_2[i]$ **else** $\text{visit}_2[i]$ *(20) Update variables II* [C.2.8]
40:        $\text{orders}_2[i]$ = order **if** $\text{changes}_3[i]$ **else** $\text{orders}_2[i]$
41:        sccs[i] = node_{ref} **if** $\text{changes}_3[i]$ **else** sccs[i]   *(21) Update variables III* [C.2.8]
42:     **end for**
43:     term = **not** (**false** in visit_3)
44:     $\text{term}_{\text{part}}$ = **not** (**false** in visit_2)   *(22) Trigger (partial) termination* [C.5.9]
45: **end while**
    return sccs

C.5.1. SELECT LIST TO FIND MINIMUM: STEP (1)

As mentioned in Appendix C.5, the implementation of the Strongly Connected Components Algorithm utilizes a two-staged approach consisting of two breadth-first searches. As demonstrated in Appendix C.4, the depth-first search is implemented using the minimum function and a priority list. To enable the change between stages, we utilize a conditional selection function, as formalized in Appendix C.1.1 to select between the two priority lists and their corresponding lists of visited nodes for masking. We express this function as cond-select(X, [orders_2, visit_3], [orders, visit_2], $\text{term}_{\text{part}}$, [orders, visit]).

C.5.2. UPDATE WRITE FLAGS: STEP (10)

In this step, we utilize the partial termination criterion $\texttt{term}_{\text{part}}$ to indicate which part of the function should be activated in the next steps of the execution. To this end, we implement the logical conditions expressed in lines [15-16] of Algorithm 6 as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i,j \in \{(\text{write}_1, \text{write}_1), (\text{write}_2, \text{write}_2), (\text{term}_{\min}, S_1), (\text{term}_{\min}, S_2), (\text{term}_{\text{part}}, S_2) \\ -1 & \text{if } i,j \in \{(\text{term}_{\text{part}}, S_1), (B_{\text{global}}, S_2) \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{write}_1, \text{write}_2, S_1, S_2\}, j = i \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(S_1, \text{write}_1), (S_2, \text{write}_2)\} \\ -1 & \text{if } i \in \{\text{write}_1, \text{write}_2\}, j = i \\ 0 & \text{otherwise,} \end{cases}$$

We erase the original content from the $\texttt{write}_1$ and $\texttt{write}_2$ columns and insert $\text{write}_1 = \phi(\text{term}_{\min} - \text{term}_{\text{part}})$ and $\text{write}_2 = \phi(\text{term}_{\min} + \text{term}_{\text{part}} - 1)$, respectively, which simulate the corresponding logical conditions.

C.5.3. READ ROW AND COLUMN OF A: STEP (12)

Once the node has been selected by the minimum function, we perform the extraction of both a row and a column of A. The process of reading a row of A is described in Section 5.3, and we can express it as $\texttt{read-A(X, Ã, node, A}_{\text{row}}\texttt{)}$ For the extraction of the column of A, we adopt the same procedure, utilizing an additional attention head that interacts with A. The implementation details for this operation are similar to the configurations described in Section 5.3, which enables us to express it as $\texttt{read-A(X, Ã.T, node, A}_{\text{col}}\texttt{)}$.

C.5.4. VISIT NODES: STEP (13)

In addition to updating the list of visited nodes in the graph, the following step also serves to update an auxiliary variable, denoted $\texttt{cur\_i}$, which aims to store the position of the current node in a one-hot encoding. This can be easily accomplished by applying the implementation presented in Appendix C.1.5 as follows: $\texttt{write-row(X, node, write}_1\texttt{, cur\_i)}$.

C.5.5. CHECK VISITED NEIGHBORS: STEP (14)

Throughout the execution of the first depth-first search, we must check if the neighbors of the current node have been visited. If this condition is met, the value of this node is updated in the second priority list, replicating the intended behavior of building a stack. This is because nodes that meet this condition later in the algorithm receive higher priority in the second list. Additionally, this condition is also used to mask nodes from subsequent steps in the first DFS.

To this end, this process is implemented in two stages: steps (14) and (15). In this step, we verify the condition expressed in lines [24-25] of Algorithm 6. The parameters of f_{attn} are set according to the implementation of Appendix C.2.5. More specifically, we implement the routines for $\texttt{repeat-n(X, write}_1\texttt{, write}_{1_n}\texttt{)}$ and $\texttt{repeat-n(X, write}_2\texttt{, write}_{2_n}\texttt{)}$. We then define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i,j \in \{(B_{\text{global}}, B_{\text{global}}), (\text{write}_{1_n}, \text{write}_{1_n}), \\ & \quad (\text{nvisit}, \text{nvisit}), (A_{\text{row}}, S_1)\} \\ -1 & \text{if } i = \text{visit}_1, \ j = S_1 \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{nvisit}, S_1\}, \ j = i \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } i,j \in \{(\text{write}_{1_n}, S_1), (\text{nvisit}, \text{nvisit})\} \\ -1 & \text{if } i,j = S_1 \\ -\Omega & \text{if } i = B_{\text{global}}, \ j = S_1 \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_1, \ j = \text{nvisit} \\ -1 & \text{if } i,j = \text{nvisit} \\ 0 & \text{otherwise.} \end{cases}$$

This implementation not only clears the target field $\texttt{nvisit}$ but also simulates the logical conditions of lines [24-25] with the following expression $\phi(\text{write}_1[i] - \phi(A_{\text{row}}[i] - \text{visit}_1[i]))$. The extra negative Ω used in $W^{(2)}$ is to remove any unwanted entries in the top row.

C.5.6. CHECK NEIGHBORING CONDITION: STEP (15)

In this step, we check if all neighbors of the current node have been visited. This information is then used to determine if the current node should be updated in the second priority list. This implementation closely follows the one utilized for the termination criterion, since it also collects all values along the last n rows of X. Therefore, we can express it as `all-one(X, nvisit, nvisit_all)`.

C.5.7. CHECK IF CURRENT SCC IS VISITED: STEP (16)

In this stage, we must verify the condition of whether the Strongly Connected Component index that is assigned to the current node is visited. This condition becomes important in the subsequent step. Initially, the SCC index of each node is itself. So, throughout the execution of the code, if we encounter an unvisited SCC index, it means that a new Strongly Connected Component has been found. In this case, the $\text{visit}_{\text{scc}}$ flag must be triggered so that in the next step, the reference node (node_{ref}) is updated. To this end, we utilize the same formulation as the read function in Appendix C.1.3 and write `read-X(X, scc, visit_3, visit_scc)`.

C.5.8. IDENTIFY UPDATES AND UPDATE PRIORITY: STEP (18)

This step of execution focuses on increasing the priority value and building the decision lists for the subsequent update steps. Despite being represented as a global variable, in our implementation of Algorithm 6, we set `order` to be a local variable, since it is solely used for updating the values of the priority lists. First, in the definition of f_{attn}, we set all parameters to zero. In the implementation of f_{MLP}, since we build three decision lists as well as the priority update, the parameters are presented separately for each of these purposes.

For the update of the priority values, we set:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{write}_{1_n}, \text{write}_{2_n}\}, j = S_{\text{order}} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i = S_{\text{order}}, \ j = i \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(4)})_{i,j} = \begin{cases} -1 & \text{if } i = S_{\text{order}}, \ j = \text{order} \\ 0 & \text{otherwise.} \end{cases}$$

In this part of the implementation, we simply subtract the writing flags that are distributed along the last n rows from the order variable, thereby increasing the priority of newly updated nodes.

For building the first list of decision variables, we write:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(\text{A}_{\text{row}}, S_1), (\text{write}_{1_n}, S_1), (\text{changes}_1, \text{changes}_1)\} \\ -1 & \text{if } i \in \{\text{visit}_1, B_{\text{local}}\}, \ j = S_1 \\ -\Omega & \text{if } i = B_{\text{global}}, \ j = S_1 \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{S_1, \text{changes}_1\}, \ j = i \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_1, \ j = \text{changes}_1 \\ -1 & \text{if } i, j = \text{changes}_1 \\ 0 & \text{otherwise,} \end{cases}$$

which simulates the condition in line 31 of Algorithm 6 with the expression: $\phi(\text{A}_{\text{row}}[i] + \text{write}_{1_n}[i] - \text{visit}_1[i] - 1)$. The additional entry of $-\Omega$ serves to clear any undesired values in the top row. For the second decision list we write:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(\text{cur\_i}, S_2), (\text{write}_{1_n}, S_2), (\text{nvisit}, S_2), (\text{changes}_2, \text{changes}_2)\} \\ -2 & \text{if } i \in \{B_{\text{global}}, B_{\text{local}}, \}, \ j = S_1 \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{S_2, \text{changes}_2\}, \ j = i \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_2, \ j = \text{changes}_2 \\ -1 & \text{if } i, j = \text{changes}_2 \\ 0 & \text{otherwise,} \end{cases}$$

where `cur_i` is the variable obtained in Appendix C.5.4 which represents the one-hot encoding of the current node. This operation simulates the condition in line 34 by using the expression: $\phi(\text{write}_{1_n}[i] + A_{\text{row}}[i] + \text{cur\_i}[i] - 2)$.

For the third decision list, we write:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(A_{\text{col}}, S_3), (\text{write}_{2_n}, S_3), (\text{changes}_3, \text{changes}_3)\} \\ -1 & \text{if } i \in \{\text{visit}_1, B_{\text{local}}\}, \ j = S_3 \\ -\Omega & \text{if } i = B_{\text{global}}, \ j = S_3 \\ 0 & \text{otherwise,} \end{cases}$$

$$(W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{S_3, \text{changes}_3\}, \ j = i \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_3, \ j = \text{changes}_3 \\ -1 & \text{if } i, j = \text{changes}_3 \\ 0 & \text{otherwise,} \end{cases}$$

This implementation simulates the condition in line 32 using the expression: $\phi(A_{\text{col}}[i] + \text{write}_{2_n}[i] - \text{visit}_3[i] - 1)$.

C.5.9. TRIGGER TERMINATION AND PARTIAL TERMINATION: STEP (22)

The last step of the loop is to verify the termination criteria for the first and second phases of the algorithm. Using the implementation described in Appendix C.1.6, we simply write these expressions as: `all-one(X, visit₃, term)` and `all-one(X, visit₂, term_part)`. Since these implementations utilize the same attention heads, their implementations are compatible and can be simply merged.

C.6. Proof of Remark 6.5

In this section, we present the methodology adopted for constructing a unified model capable of executing multiple graph algorithms, as stated in Remark 6.5. More specifically, we construct a multitask model that executes three distinct graph algorithms: Breadth-first search, Depth-first search, and Dijkstra's shortest path algorithm. We limit our scope to these three algorithms, considering the complexity and minimal additional insights gained from incorporating more algorithms. However, this same design principle can be applied to a broader range of algorithms, including Strongly Connected Components (referenced as Appendix C.5). Incorporating a new set of algorithms could potentially incur the introduction of new functions and a distinct set of variables to be integrated into the matrix X.

Some initial considerations are necessary to describe the construction strategy. Each algorithm requires a distinct set of variables and functions. In our constructions, this is reflected as specific columns in the input matrix and the different implemented layers, respectively. Therefore, a model that unifies different algorithms must accommodate these individual components without compromising the execution of any particular algorithm. Nevertheless, some algorithms can have common structures, sharing similar functions or variables utilized in analogous ways. The challenge of multitasking extends beyond just encapsulating different executions within the same model. It also involves efficiently reusing shared variables and functions to avoid redundancy, which also significantly reduces overhead in terms of memory (number of columns of X) or runtime (number of layers).

Our goal is to provide a single implementation capable of executing one of these three algorithms given the appropriate input configuration. The structure of the input is the same for all three algorithms. However, its configurations slightly change for the execution of each algorithm. While Breadth & Depth-first search operate on unweighted graphs, the execution of Depth-first search is distinguished by the activation of a specific flag in X, denoted by γ_s. For Breadth-first search and Dijkstra's algorithm, the configuration of X is the same. What sets Dijkstra's algorithm apart from BFS is its operation on a weighted graph, as opposed to the unweighted graphs used by the other two algorithms. This distinction also highlights the fact that Breadth-first search can be considered a special case of Dijkstra's algorithm when applied to unweighted graphs. Consequently, we can leverage a single execution for both breadth-first search and Dijkstra's algorithm.

Furthermore, all three algorithms share a large portion of similar functions. For example, they all employ the minimum function during the initial phase of iteration and utilize a similar termination criterion. Our implementation strategy consists of leveraging this shared structure while individually accommodating the unique functions of each algorithm. The selection of specific elements necessary for executing a particular algorithm is managed by a selector function. This function determines the variables that need to be updated, thus ensuring the execution reflects the intended algorithmic behavior.

The comprehensive structure of our implementation is illustrated in Algorithm 7. Non-highlighted lines indicate the shared structural components common to all three algorithms. In contrast, colored lines denote algorithm-specific adaptations. Specifically, lines highlighted in blue (18, 22, 29, 38, and 39) are modifications for Depth-first search. Lines in red (24, 25, 30, 36, and 37) indicate the adaptations for both Dijkstra's and Breadth-first search. Lastly, the lines highlighted in orange (42-45) represent the conditional selection mechanism. This mechanism is crucial for dynamically selecting the algorithm-specific elements and the boolean variables that trigger these adaptations.

Since the implementations directly follow the specifications outlined in previous sections, the guarantees for each algorithm are established according to their respective designs (refer to Appendix C.3, Appendix C.4, and Appendix C.2). Furthermore, except for the selector function process, the details of each algorithmic step have been thoroughly discussed earlier. In the following, we present the implementation of the selector function, along with a detailed description of the algorithm.

Additionally, we also conduct empirical validation, as detailed in Appendix B.3. This validation confirms the robustness of our unified implementation, described in Algorithm 7, which demonstrates a 100% accuracy across all tested instances of the three algorithms.

C.6.1. Update priority factor: Step (12)

As previously discussed in Appendix C.4.1, for the execution of Depth-first search, the priority variable `order` must be decreased at each iteration. However, for the multitask model, this process should not be carried out if the model is executing a different algorithm. To this end, we introduce a condition for updating the priority factor.

In our construction, we substitute the conditional form for an equivalent expression: $\text{order} = \text{order} - \phi(\text{term}_{\min} + \gamma_s - 1)$. This ensures that the variable `order` is only updated if `term`$_{\min}$ and γ_s are activated. We implement this condition by

setting the parameters of f_{attn} to zero, and we define the parameters of f_{MLP} as follows:

$$
(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{term}_{\min}, \gamma_s\}, j = \text{order} \\ -1 & \text{if } i = B_{\text{global}}, j = \text{order} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i, j = \text{order} \\ 0 & \text{otherwise.} \end{cases}
$$

$$
(W^{(4)})_{i,j} = \begin{cases} -1 & \text{if } i, j = \text{order} \\ 0 & \text{otherwise.} \end{cases}
$$

The output of the last layer of f_{MLP} is directly added to the residual connection X, effectively replicating the expression above.

C.6.2. SELECT CANDIDATES AND CHANGES VARIABLES: STEP (17)

The variable `candidates` represents the values used for updating the current distances or priorities, essential for the Dijkstra/BFS and DFS algorithms. Specifically, candidates_1 refers to the candidate values for Dijkstra/BFS, while candidates_2 indicates those for DFS. Similarly, the `changes` variable is a boolean-flag array containing flags that indicate which values require updating. The variables changes_1 and changes_2 correspond to the update flags for Dijkstra/BFS and DFS, respectively.

Finally, during the algorithm's execution, we must determine which variables are going to be chosen: changes_1 and candidates_1 or changes_2 and candidates_2. This decision is guided by the boolean flag γ_s, which, when activated, indicates that the DFS routine should be executed, thereby selecting the second set of variables; otherwise, the first set is chosen. This expression is implemented as `cond-select(X, [change₂, candidates₂], [change₁, candidates₁], γₛ, [change, candidates]`, utilizing the conditional selection function described in Appendix C.1.1. Here, the variable γ_s is also repeated along the last n rows during step (14) of the algorithm, whose objective is to replicate the top row value along these rows.

Algorithm 7 General algorithm for DFS/BFS/Dijkstra

Input: integer start
Input: bool γ_s, switch flag
Input: matrix A, size $n \times n$

1: visit[start], order, term = 0, 0, **false**
2: prev, visit, dists, dists$_{masked}$, changes, is_zero, candidates = arrays of size n
3: **for** $i = 1$ **to** n **do**
4:    visit[i], dists[i], prev[i] = **false**, $\hat{\Omega}$, i
5: **end for**
6: $\cdots$ *Initialization of min-variables*

7: **while** term is **false** **do**
8:    **for** $i = 1$ **to** n **do**
9:       **if** visit[i] is **true** **then**
10:          dists$_{masked}$[i] = Ω *(1) Mask visited nodes* [C.2.1]
11:       **else**
12:          dists$_{masked}$[i] = dists[i]
13:       **end if**
14:    **end for**

15:    get_minimum(dists$_{masked}$)   *(2-8) Find minimum value* [C.1]

16:    **if** term$_{min}$ is **true** **then**
17:       node = idx$_{best}$
18:       dist = val$_{best}$ *(9) Get minimum values* [C.2.2]
19:    **end if**
20:    A$_{row}$ = A[node, :]   *(10) Get row of A* [C.2.3]
21:    **for** $i = 1$ **to** n **do**
22:       is_zero[i] = (A$_{row}$[i] ≤ 0)   *(11) Mark non-neighbors* [C.2.4]
23:    **end for**
24:    **if** γ_s is **true** **then**
25:       order = order - term$_{min}$ *(12) Update priority factor* [C.6.1]
26:    **end if**
27:    visit[node] = visit[node] + term$_{min}$ *(13) Visit node* [C.2.9]
28:    **for** $i = 1$ **to** n **do**
29:       candidates$_1$[i] = A$_{row}$[i] + dist   *(14) Build candidates* [C.2.5]
30:       candidates$_2$[i] = order
31:    **end for**
32:    **for** $i = 1$ **to** n **do**
33:       changes$_1$[i] = candidates$_1$[i] $<$ dists[i]   *(15) Identify updates* [C.2.6]
34:    **end for**
35:    **for** $i = 1$ **to** n **do**
36:       change$_2$ = term$_{min}$ is **true** and visit[i] is **false**   *(16) Build flags* [C.2.7/C.4.2]
37:       changes$_2$[i] = change$_2$ is **true** and A$_{row}$[i] is 1
38:       **if** term$_{min}$ is **false and** is_zero[i] is **true** **then**
39:          changes$_1$[i] = 0
40:       **end if**
41:    **end for**
42:    **if** γ_s is **true** **then**
43:       candidates, changes = candidates$_2$, changes$_2$ *(17) Select candidates/changes* [C.6.2]
44:    **else**
45:       candidates, changes = candidates$_1$, changes$_1$
46:    **end if**
47:    **for** $i = 1$ **to** n **do**
48:       **if** changes[i] is **true** **then**
49:          prev[i], dists[i] = node, candidates[i]   *(18) Update variables* [C.2.8]
50:       **end if**
51:    **end for**
52:    term = **not** (**false** in visit)   *(19) Trigger termination* [C.2.10]
53: **end while**
   return prev, dists

C.7. Proof of Remark 6.6

In this section, we present the results on the Turing Completeness of the architecture described in (2). Previous work by Pérez et al. (2021) and Giannou et al. (2023) has established universality results for looped transformers with standard attention. Nevertheless, it is important to assess how the expressiveness of the architecture is affected when graph connectivity is stored as a separate object and graph convolution is integrated into the attention mechanism.

To establish Turing Completeness, we adapt the method employed in Giannou et al. (2023). In particular, we illustrate how our proposed architecture can efficiently emulate SUBLEQ, a single-instruction language recognized for its Turing Completeness (Mavaddat & Parhami, 1988).

Our discussion begins with an overview of SUBLEQ, followed by the introduction of a modified version termed SUBLEQ$^-$ (read as *SUBLEQ minus*), which is also Turing Complete. Subsequently, we demonstrate that SUBLEQ$^-$ can be simulated using a SUBLEQ-like instruction that uses a specialized memory object for the adjacency matrix, which we name Graph-SUBLEQ. Lastly, we establish that the architecture depicted in (2) is capable of simulating Graph-SUBLEQ.

SUBLEQ: Named for its operation "subtract and branch if less than or equal to zero", SUBLEQ is a one-instruction set computer. As detailed in Algorithm 8, it consists of subtracting the content at address a from that at address b, and storing the result back at b. All these values are stored in a one-dimensional memory array. If the result is non-negative, the computer executes the next instruction; otherwise, it jumps to the instruction at address c. Despite this operational simplicity, SUBLEQ is Turing Complete (Mavaddat & Parhami, 1988).

Algorithm 8 SUBLEQ (a, b, c)

Input: memory object M, addresses a, b, c
1: $M[b] = M[b] - M[a]$
2: **if** $M[b] \leq 0$ **then**
3:     go to c
4: **else**
5:     go to next instruction
6: **end if**

SUBLEQ$^-$: In the remainder of the proof, we focus on a specialized variant of SUBLEQ, which we refer to as SUBLEQ$^-$. This modified version operates similarly to the standard SUBLEQ, with a key distinction in handling the adjacency matrix in memory. In SUBLEQ, a common approach to represent graph adjacency data involves vectorizing the adjacency matrix and placing it at the beginning of the memory. In this case, specifically in SUBLEQ$^-$, the first n^2 memory entries, representing the row-major order vectorization of an adjacency matrix for a graph with n nodes, are set to be read-only. It is important to note that this read-only constraint on the first n^2 entries does not diminish the expressive power of SUBLEQ$^-$. The data in these entries can always be accessed and then copied into writable memory locations using standard SUBLEQ instructions. Furthermore, these n^2 entries may not exist for routines that do not utilize the graph in the proposed way. Consequently, even with this restriction, SUBLEQ$^-$ remains Turing Complete.

Graph-SUBLEQ: Building upon the concept of SUBLEQ$^-$, we introduce Graph-SUBLEQ, a new formulation with a distinct approach to memory management. Unlike SUBLEQ$^-$, which uses a single memory object with varying writing permissions, Graph-SUBLEQ separates its memory into two distinct entities: a writable one-dimensional memory object, as in standard SUBLEQ, and a separate read-only memory dedicated to storing graph connectivity data. Crucially, in this second memory, data is arranged in a matrix format, with the same structure as an adjacency matrix.

Algorithm 9 Graph-SUBLEQ (a, b, c, γ_a)

Input: memory objects M, M_G, addresses $a := (a_1, a_2)$, b and c, flag γ_a

1: **if** γ_a is **true then**
2:     $m_a = M_G[a_1, a_2]$
3: **else**
4:     $m_a = M[a_1]$
5: **end if**
6: $m_b = M[b]$
7: $M[b] = m_b - m_a$
8: **if** $M[b] \leq 0$ **then**
9:     go to c
10: **else**
11:     go to next instruction
12: **end if**

The implementation of Graph-SUBLEQ, as shown in Algorithm 9, closely resembles that of SUBLEQ in Algorithm 8. The main difference lies in the structure of the instructions. While SUBLEQ operates with the triplet a, b, c, a Graph-SUBLEQ instruction is defined by a, b, c, γ_a. In this context, a represents a pair of addresses, a_1 and a_2. These addresses serve two functions: together, they can access an element in the graph memory object M_G, or, using only a_1, access an element in the main memory M. The decision of which memory is accessed by a is determined by the boolean flag γ_a: if γ_a is True, the instruction accesses the graph memory; otherwise, it accesses the main memory M. All other aspects of Graph-SUBLEQ's implementation are akin to those of SUBLEQ.

With the frameworks of SUBLEQ$^-$ and Graph-SUBLEQ established, we now present a lemma important for our concluding remark.

Lemma C.1. *Graph-SUBLEQ is Turing Complete.*

Proof. To establish the Turing Completeness of Graph-SUBLEQ, we must demonstrate that every instruction in SUBLEQ$^-$ has an equivalent instruction in Graph-SUBLEQ. This equivalence is essential to ensure that Graph-SUBLEQ can perform all operations that SUBLEQ$^-$ can. The equivalence is examined in two scenarios based on the type of instruction in SUBLEQ$^-$: instructions that read from the read-only graph adjacency block, and those that read from other memory areas. To maintain clarity in our notation, we use a, b, c to represent instructions in SUBLEQ$^-$, and $\tilde{a}_1, \tilde{a}_2, \tilde{b}, \tilde{c}$ for instructions in Graph-SUBLEQ.

- **No access to graph data:** To establish equivalence for instructions not accessing graph data, we define $\gamma_{\tilde{a}} = 0$, and set $\tilde{a}_1 = a - n^2$, $\tilde{a}_2 = 0$, $\tilde{b} = b - n^2$, and $\tilde{c} = c$. Here, n represents the size of the graph.

- **Access to graph data:** For instructions that access graph data, the equivalence is achieved by defining $\gamma_{\tilde{a}} = 1$, and setting $\tilde{a}_1 = \lfloor a/n \rfloor$ and $\tilde{a}_2 = a - n\tilde{a}_1$. We also set $\tilde{b} = b - n^2$, and $\tilde{c} = c$, where n is the size of the graph.

When SUBLEQ$^-$ does not access graph data, the conversion to Graph-SUBLEQ requires a few adjustments to the memory addresses. In Graph-SUBLEQ, the address $\tilde{a}_2$ is not utilized in this scenario. The addresses $\tilde{a}$ and $\tilde{b}$ in Graph-SUBLEQ are simply the corresponding SUBLEQ$^-$ addresses offset by $-n^2$, to accommodate the different indexing scheme in Graph-SUBLEQ's memory objects. For the addresses c and $\tilde{c}$, no modification is needed, as they are already aligned and refer to elements outside the memory reserved for the graph. In situations where a graph is not used in the memory, n is equal to 0, making the addresses in SUBLEQ$^-$ and Graph-SUBLEQ naturally align.

In cases where SUBLEQ$^-$ accesses graph data through the address a, we derive the corresponding elements $\tilde{a}_1$ and $\tilde{a}_2$ in the matrix format of the graph memory. Given that a in SUBLEQ$^-$ represents an entry in the vectorized adjacency matrix arranged in row-major order, $\tilde{a}_1$ is set to the integer part of a/n, and $\tilde{a}_2$ is the remainder of the division a/n. This approach ensures proper alignment of indexes between SUBLEQ$^-$ and Graph-SUBLEQ in both scenarios, thereby showing that Graph-SUBLEQ is also Turing Complete. $\square$

Having properly defined Graph-SUBLEQ and showing it that is Turing Complete, we now write the Remark 6.6 in its more precise form:

Remark C.2. There exists a looped-transformer h_T in the form of (2), which utilizes the modified attention head in (1), with 11 layers, 3 attention heads, and layer width $O(1)$ that simulates Graph-SUBLEQ.

Proof. In this proof, we employ the architecture outlined in Equation (2) to simulate Algorithm 9. This constructive approach is used to demonstrate that the architecture is Turing Complete. As with our other proofs, we start by presenting an adaptation of Algorithm 9 more aligned with our architecture. Within this adapted algorithm, the commands executed inside the while loop correspond to the instructions of Graph-SUBLEQ. Following the adaptation, we provide a comprehensive description of the input structure, detailing how the data and commands are organized and processed.

Finally, we delve into the systematic construction of the algorithm. Each step has an associated transformer layer in the form of (2) that implements the corresponding routine. In the simulation of Graph-SUBLEQ shown in Algorithm 10 there is a total of 11 steps, and therefore our implementation requires 11 layers, which use a total of 3 distinct attention heads. Furthermore, throughout the implementation details of each step, no configuration uses parameters whose count scales with the size of the graph, thus resulting in constant network width.

Algorithm 10 Graph-SUBLEQ (a, b, c, γ_a) with external memory block

      Input: memory objects M, M_G, instruction list I, instruction index k

1: **while true do**
2:     $((a_1, a_2), b, c, \gamma_a) = I[k]$ *(1) Read instructions* [C.7.2]
3:     $m_G = M_G[a_1, :]$ *(2) Read row a_1 from M_G* [C.7.3]
4:     $m_{a,G} = m_G[a_2]$ *(3) Read address a from M_G* [C.7.2]
5:     $m_{a,X} = M[a_1]$ *(4) Read address a from M* [C.7.2]
6:     $m_a = m_{a,G}$ **if** γ_a is **true else** $m_{a,X}$ *(5) Select the memory value of a* [C.7.4]
7:     $m_b = M[b]$ *(6) Read address b from M* [C.7.2]
8:     diff $= m_b - m_a$ *(7) Compute difference between values* [C.7.5]
9:     $M[b] = $ diff   *(8) Write difference in memory* [C.7.6]
10:    $k_{\text{next}} = k + 1$ *(9) Compute next instruction* [C.7.7]
11:    cond$_k = $ diff≤ 0 *(10) Compute condition for k* [C.7.8]
12:    $k = c$ **if** cond$_k$ is **true else** k_{next} *(11) Select next instruction* [C.7.4]
13: **end while**

C.7.1. INPUT INITIALIZATION

We utilize the same format consistently used in this work, having an input matrix X and an external adjacency matrix A. The structure of the input matrix follows the convention outlined in Section 5.1, incorporating elements such as global and local variables, positional encodings, and biases. We maintain the same naming convention for consistency, with some adaptations specific to this context. For instance, we also refer to the list of all positional encodings by P, while the positional encoding of the current instruction in this implementation is represented by k, as detailed in Algorithm 10.

Given that in Graph-SUBLEQ, the size of the main memory and the graph might differ, directly translating these to the formats of X and A could result in a dimension mismatch, thereby hindering the multiplication process described in (1). To address this, we define $K = \max(|M|, n) + 1$, where K is the larger of the two dimensions, either the size of the main memory or the number of nodes in the graph. This ensures compatibility in dimensions for encoding in X and A. In this setup, both the instructions and memory are padded with zeros in X and A to align with K.

In terms of specific variable representation, the main memory block, denoted as M, is represented as a single-column local variable within this framework. The instruction list, labeled as I, comprises a set of local variables, each representing an address in the instruction set. These are individually identified by their specific assignments. For example, I_{a_1} refers to the column in the instruction list where addresses a_1 are stored.

C.7.2. READ FROM INPUT: STEPS (1), (3), (4), AND (6)

In the implementation of the read function within our architecture, we follow the construction outlined in Appendix C.1.3. For step (1), the operation is expressed as read-X (X, k, I_{a_1}, Z_{a_1}). Here, k specifies the columns that store the positional encodings of the current instruction, and Z_{a_1} is the variable holding the current value for a_1. The same formulation is then extended to all other values contained in the instruction, that is, a_2, b, c and γ_a.

Moving to step (3), after the row of the graph memory is written into the variable m_G, this value is retrieved using the

operation $\texttt{read-X}\,(X, Z_{a_2}, m_G, m_{a,G})$. In this context, $m_{a,G}$ is the target field where the retrieved value will be stored.

In step (4), the read operation is defined as $\texttt{read-X}\,(X, Z_{a_1}, M, m_{a,X})$. The variable M represents the memory column, and $m_{a,X}$ is designated to hold the entry of a retrieved from M.

Finally, for step (6), the operation is set as $\texttt{read-X}\,(X, Z_b, M, m_b)$. Here, Z_b indicates the columns corresponding to the current address b, and m_b is the variable designated to store the retrieved entry of b from M.

C.7.3. READ FROM $\tilde{A}$: STEP (2)

The implementation of step (2), which involves reading from the adjacency matrix $\tilde{A}$, is based on the procedure detailed in Section 5.3. In this specific application, the read operation from $\tilde{A}$ is described as $\texttt{read-A}\left(X, \tilde{A}, , Z_{a_1}, m_G\right)$. Here, Z_{a_1} represents the columns that contain the current value of a_1, which is used to locate the specific row in the adjacency matrix $\tilde{A}$ that needs to be read. The variable m_G is designated to store the retrieved row from $\tilde{A}$.

C.7.4. CONDITIONAL SELECTION: STEPS (5) AND (11)

The conditional selection process in steps (5) and (11) of our Graph-SUBLEQ implementation follows the approach outlined in previous algorithms, as detailed in Appendix C.1.1.

In step (5), the conditional selection operation is expressed as $\texttt{cond-select}\,(X, m_{a,G}, m_{a,X}, Z_{\gamma_a}, m_a)$. Here, m_a is the target field where the selected value will be stored. Z_{γ_a} is the variable holding the current value of γ_a, which determines whether to select $m_{a,G}$ (the value from the graph memory) or $m_{a,X}$ (the value from the main memory).

For step (11), the operation is defined as $\texttt{cond-select}\,(X, Z_c, k_{\text{next}}, \text{cond}_k, k)$. In this context, Z_c refers to the columns that hold the current value of c, and $k$next represents the columns storing the index of the next instruction's positional encoding. The conditional logic applied here determines which instruction's positional encoding, either the current or next, is to be used based on the condition cond_k.

C.7.5. COMPUTE SUBTRACTION: STEP (7)

In step (7) of our implementation, we focus on calculating the difference between the values stored in m_b and m_a, and recording the result in a designated column, which we refer to as $\texttt{diff}$. To carry out this operation, we set the parameters of f_{attn} to zero and define the parameters of f_{MLP} as follows:

$$
(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{(m_b, S_1), (\text{diff}, \text{diff})\} \\ -1 & \text{if } i \in \{(m_b, S_1), (\text{diff}, S_2)\} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{S_1, S_2, \text{diff}\}, \ j = i \\ 0 & \text{otherwise,} \end{cases}
$$

$$
(W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{S_1, S_2\}, \ j = \text{diff} \\ -1 & \text{if } i, j = \text{diff} \\ 0 & \text{otherwise.} \end{cases}
$$

Here, the first layers obtain the subtraction of the desired values, while the last layer places the result in the desired field. Additionally, the original value in $\texttt{diff}$ is also replicated for both its positive and negative versions. These are used in the final layer to clear the previous entry.

C.7.6. WRITE IN MEMORY: STEP (8)

For the implementation of step (8), where we write the result back into memory, we adopt the methodology outlined in Appendix C.1.5. We express this operation by $\texttt{write-row}\,(X, Z_b, \texttt{diff}, M)$. In this context, X represents our input matrix, and Z_b indicates the columns corresponding to the address b in the memory, where the subtraction result ($\texttt{diff}$) needs to be written. The $\texttt{diff}$ value holds the result of the subtraction computed in the previous step. The target for this write operation is the memory column M, which will be updated with the new value.

C.7.7. INCREMENT POSITIONAL ENCODING: STEP (9)

The implementation of step (9), which involves incrementing the positional encoding to move to the next instruction, follows the procedure described in Appendix C.1.2. The operation is defined as increment (X, k, k_{next}). Here, X is the input matrix, k represents the columns storing the current positional encoding, and k_{next} denotes the columns that will store the positional encoding of the next instruction.

C.7.8. COMPUTE CONDITION FOR NEXT INSTRUCTION: STEP (10)

In step (10), we need to check if the result of the subtraction (stored in diff) is less than or equal to zero. This is similar to the comparison function in Equation Equation (3), but with a modification to include equality. Essentially, this function will return true if diff is zero or negative, allowing the algorithm to decide the next step based on this condition. In this case, we approximate the less-or-equal function as follows:

$$X[:, C] \le X[:, D] \approx \varepsilon^{-1} \left(\phi \left(X[:, D] - X[:, C] + \varepsilon \right) - \phi \left(X[:, D] - X[:, C] \right) \right) \tag{15}$$

where C and D are the columns used for comparison. Notice that in this implementation, different from (3), the argument of the first ReLU includes an additional positive ε value Furthermore, for the application in the context of the simulation of Graph-SUBLEQ, the utilization of two columns C and D is not strictly necessary, since the difference between the values is already stored in the variable diff and the comparison is essentially made against zero.

To implement this function, we begin by setting the parameters f_{attn} to zero. Then, we define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i, j = \text{cond}_k \\ -1 & \text{if } i = \text{diff}, \ j \in \{S_1, S_2\} \\ \varepsilon & \text{if } i = B_{\text{global}}, \ j = S_2 \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(2)})_{i,j} = \begin{cases} 1 & \text{if } i, j = \text{cond}_k \\ \varepsilon^{-1} & \text{if } i, j = S_1 \\ -\varepsilon^{-1} & \text{if } i = S_2, \ j = S_1 \\ 0 & \text{otherwise}, \end{cases}$$

$$(W^{(3)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{cond}_k, S_1\}, \ j = i \\ 0 & \text{otherwise}, \end{cases} \qquad (W^{(4)})_{i,j} = \begin{cases} 1 & \text{if } i = S_1, \ j = \text{cond}_k \\ -1 & \text{if } i, j = \text{cond}_k \\ 0 & \text{otherwise}. \end{cases}$$

In the first layer of the process, we construct the arguments for the ReLU functions as defined in (15). Additionally, we maintain the value in cond_k to clear the target column. Moving to the second layer, the ReLU terms are then processed by subtracting one from the other, and the resulting value is divided by ε. In the final layer, this computed value is recorded in the target column, simultaneously erasing its original value.

$\square$