Transformers Meet Directed Graphs

Simon Geisler [* 1] **Yujia Li** [2] **Daniel Mankowitz** [2] **Ali Taylan Cemgil** [2] **Stephan Günnemann** [1] **Cosmin Paduraru** [2]

Abstract

Transformers were originally proposed as a sequence-to-sequence model for text but have become vital for a wide range of modalities, including images, audio, video, and *undirected* graphs. However, transformers for *directed* graphs are a surprisingly underexplored topic, despite their applicability to ubiquitous domains, including source code and logic circuits. In this work, we propose two direction- and structure-aware positional encodings for *directed* graphs: (1) the eigenvectors of the Magnetic Laplacian – a direction-aware generalization of the combinatorial Laplacian; (2) directional random walk encodings. Empirically, we show that the extra directionality information is useful in various downstream tasks, including correctness testing of sorting networks and source code understanding. Together with a data-flow-centric graph construction, our model outperforms the prior state of the art on the Open Graph Benchmark Code2 relatively by 14.7%.[3]

1. Introduction

Transformers have become a central component in many state-of-the-art machine learning models spanning a wide range of modalities. For example, transformers are used to generate solutions for competitive programming tasks from textual descriptions (Li et al., 2022), for conversational question answering with the popular ChatGPT (OpenAI, 2022), or to find approximate solutions to combinatorial optimizations problems like the Traveling Salesman Problem (Kool et al., 2019). Transformers have also had success in graph learning tasks, e.g., for predicting the properties of molecules (Min et al., 2022). While virtually all prior works focus on undirected graphs, we advocate the use of di-

---

* Work performed while at Google DeepMind [1]Dept. of Computer Science & Munich Data Science Institute, Technical University of Munich [2]Google DeepMind. [3]Code and configuration: `www.cs.cit.tum.de/daml/digraph-transformer` Correspondence to: Simon Geisler <s.geisler@tum.de>.

rected graphs as they are omnipresent, and directedness can rule semantics. Transformers that handle both undirected and directed graphs could become an important building block for many applications. For this, the attention mechanism needs to become aware of the graph structure. For example, prior work modified the attention mechanism to incorporate structural information (Ying et al., 2021) or proposed hybrid architectures that also contain Graph Neural Networks (GNNs) (Mialon et al., 2021; Chen et al., 2022). Another (complementary) option are positional encodings that are used by many, if not most, structure-aware transformers (Min et al., 2022; Müller et al., 2023).

Directional positional encodings. Most of the literature for structure-aware positional encodings either uses basic measures like pair-wise shortest path distances (Guo et al., 2021; Ying et al., 2021) or symmetrizes the graph for principled positional encodings, e.g., based on the combinatorial Laplacian (Dwivedi & Bresson, 2021; Kreuzer et al., 2021). Importantly, symmetrization might discard essential information that determines the semantics of the input. For these reasons, we propose two direction-aware positional encodings: (1) the eigenvectors of the Magnetic Laplacian (§ 3), which naturally generalizes the well-known combinatorial Laplacian to directed graphs (see Fig. 1); and (2) directional random walk encodings (§ 4) that generalize basic measures like the shortest path distances. We show that our positional encodings are predictive for various distances on graphs (§ 5) and are useful in downstream tasks. Moreover, our positional encodings can also improve GNNs (see Fig. 10).

Motivation for directed graphs. We make the impact of appropriately modeling inputs via directed graphs explicit for our applications. One example is the correctness prediction of sorting networks (§ 6). Sorting networks (Knuth, 1973) are a certain sorting procedures that can be represented by

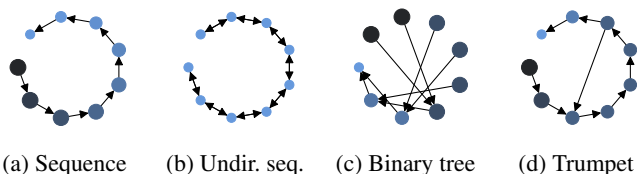(a) Sequence (b) Undir. seq. (c) Binary tree (d) Trumpet

Figure 1: First eigenvector of Magnetic Laplacian. Node size encodes the real value and colors the imaginary value.

a fixed sequence of operations. The goal is then to predict whether the sequence is a correct sorting network. Based on the sequence of operations, we can construct a (directed acyclic) data-flow graph that models the dependencies between the operations. Conversely, the topological sorts of this graph correspond to different but semantically equivalent sequences of operations. Considering the potentially large number of topological sorts, directed graphs can drastically reduce the effective input dimensionality (e.g., see Fig. 7). Moreover, we show that ignoring the edge directions maps both correct and incorrect sorting networks to the same *undirected* graph, losing critical information.

Interestingly, representing source code as a sequence is the de facto standard (Li et al., 2022; Feng et al., 2020; Chen et al., 2021; OpenAI, 2022). Even graph-based representations of code (Allamanis et al., 2018; Hu et al., 2020; Cummins et al., 2020; Guo et al., 2021; Bieber et al., 2022) only enrich sequential source code, e.g., with an Abstract Syntax Tree (AST). However, similar to sorting networks, we can often reorder certain statements without affecting the code's functionality. This motivates us to rethink the graph construction for source code, which not only boosts performance but makes the model invariant w.r.t. certain meaningless reorderings of statements (see § 7 for details).

Contributions: [I] We make the connection between sinusoidal positional encodings and the eigenvectors of the Laplacian explicit (§ 2). **[II]** We propose *spectral positional encodings* that also generalize to directed graphs (§ 3). **[III]** We extend random walk positional encodings to directed graphs (§ 4). **[IV]** As a plausibility check, we assess the predictiveness of structure-aware positional encodings for a set of graph distances (§ 5). **[V]** We introduce the task of predicting the correctness of *sorting networks*, a canonical ambiguity-free application where directionality is essential (§ 6). **[VI]** We quantify the benefits of modeling a sequence of program statements as a directed graph and rethink the graph construction for source code to boost predictive performance and robustness (§ 6 & 7). **[VII]** We set a new *state of the art* on the OGB Code2 dataset (2.85% higher F1 score, 14.7% relatively) for function name prediction (§ 7).

2. Sinusoidal and Laplacian Encodings

Due to the permutation equivariant attention, one typically introduces domain-specific inductive biases with Positional Encodings (PEs). For example, Vaswani et al. (2017) proposed sinusoidal positional encodings for sequences along with the transformer architecture. It is commonly argued (Bronstein et al., 2021; Dwivedi & Bresson, 2021) that the eigenvectors of the (combinatorial) Laplacian can be understood as a generalization of the sinusoidal positional encodings (see Fig. 2) to graphs, due to their relationship via the Graph Fourier Transformation (GFT) and Discrete
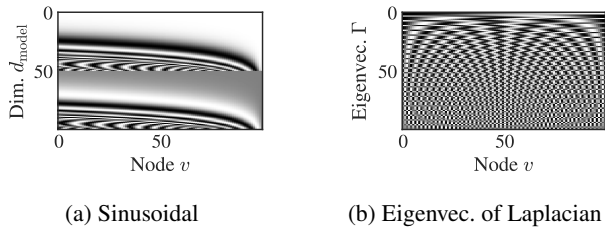

(a) Sinusoidal   (b) Eigenvec. of Laplacian

Figure 2: (a) Sinusoidal encodings (*sin* components top and *cos* below) with denominator $1,000^{2j/d_{\text{model}}}$ and $d_{\text{model}} = 100$. (b) Lap. eigenvec. of sequence Fig. 1b of len. $n = 100$.

Fourier Transformation (DFT) (Smith, 1999). Even though sinusoidal positional encodings capture the direction, eigenvectors of the Laplacian do not. But why is this the case? To understand differences and commonalities, we next contrast sinusoidal encodings, DFT, and Laplacian eigenvectors for a sequence (Fig. 1a,1b).

Sequence encodings. Sinusoidal encodings (Vaswani et al., 2017) form a d_{model}-dimensional embedding of token u's integer position in the sequence using cosine $\text{PE}_{u,2j}^{(\sin)} := \cos\left(u/10{,}000^{2j/d_{\text{model}}}\right)$ and sinus $\text{PE}_{u,2j+1}^{(\sin)} := \sin\left(u/10{,}000^{2j/d_{\text{model}}}\right)$ waves of varying frequencies with $j \in \{0, 1, ..., d_{\text{model}}/2 - 1\}$. Analogously, the **DFT** could be used to define positional encodings:

$$X_j := \sum_{u=0}^{n-1} x_u \Big[\underbrace{\cos\left(\frac{2\pi}{n}ju\right)}_{\text{PE}_{u,2j}^{(\text{DFT})}} - i \cdot \underbrace{\sin\left(\frac{2\pi}{n}ju\right)}_{\text{PE}_{u,2j+1}^{(\text{DFT})}}\Big] \quad (1)$$

Here X corresponds to signal x in the frequency domain. In contrast to the DFT, sinusoidal encodings (a) sweep the frequencies using a geometric series instead of linear; (b) also contain frequencies below $1/n$ (longest wavelength is $2\pi 10,000$); and (c) have d_{model} components instead of $2n$ (i.e., $0 \leq j < n$ in Eq. 1).

Graphs generalize sequences to sets of tokens/nodes with arbitrary connections. In a graph $\mathcal{G} = (V, E)$, the m edges E represent connections between the n nodes V. We use $\boldsymbol{X}^{(n)}$ for node features and $\boldsymbol{X}^{(m)}$ for edge features. We denote the in-degree of node u with $|\{v|(v, u) \in E\}|$ and out-degree with $|\{v|(u, v) \in E\}|$. Alternatively to E, we denote the adjacency matrix $\boldsymbol{A} \in \{0, 1\}^{n \times n}$ and refer with $\boldsymbol{D} \in \mathbb{R}^{n \times n}$ to the diagonalized degree matrix. Analogously, we describe the symmetrized adjacency matrix $\boldsymbol{A}_s = \boldsymbol{A} \vee \boldsymbol{A}^\top$ with set of edges E_s and degree matrix $\boldsymbol{D}_s$. In the main part of the paper, we only discuss unweighted graphs; however, our methods naturally generalize to weighted graphs (see § E).

Eigenvectors of Laplacian. The Graph Fourier Transformation (GFT) for undirected graphs $\boldsymbol{X} = \boldsymbol{\Gamma}^\top \boldsymbol{x}$) can be defined based on the eigendecomposition of the combina-

torial Laplacian $\boldsymbol{L} = \boldsymbol{\Gamma}\boldsymbol{\Lambda}\boldsymbol{\Gamma}^{-1}$, with diagonal matrix $\boldsymbol{\Lambda}$ of eigenvalues and orthogonal matrix $\boldsymbol{\Gamma} \in \mathbb{R}^{n \times n}$ of eigenvectors (see § B for details). Similarly to the DFT, $\boldsymbol{\Gamma}$ are suitable positional encodings. The unnormalized Laplacian $\boldsymbol{L}_U$ as well as degree-normalized Laplacian $\boldsymbol{L}_N$ are defined as:

$$\boldsymbol{L}_U := \boldsymbol{D}_s - \boldsymbol{A}_s \quad (2) \quad \boldsymbol{L}_N := \boldsymbol{I} - (\boldsymbol{D}_s^{-1/2}\boldsymbol{A}_s\boldsymbol{D}_s^{-1/2}) \quad (3)$$

The symmetrization $\boldsymbol{A}_s$ discards directionality but is required s.t. $\boldsymbol{L}$ is guaranteed to be symmetric and positive semi-definite. This ensures that $\boldsymbol{\Gamma}$ form an orthogonal basis, which entails important properties of the GFT and for PEs (see § C for a discussion). In the following, we index eigenvalues and eigenvectors by order: $0 \le \lambda_0 \le \lambda_1 \le \cdots \le \lambda_{n-1}$. We call λ_0 or $\Lambda_{0,0}$ the first eigenvalue and $\boldsymbol{\gamma}_0$ or $\boldsymbol{\Gamma}_{:,0}$ the first eigenvector that reflect the lowest frequency.

Laplacian vs. DFT. Two notable differences to the DFT in Eq. 1 are (1) that the eigenvectors of the Laplacian are real-valued; (2) the eigenvectors are not unique, e.g., due to the *sign ambiguity*. That is, if $\boldsymbol{\gamma}$ is an eigenvector, so is $-\boldsymbol{\gamma}$.

Cosine Transformation. A possible set of eigenvectors of the combinatorial Laplacian for a sequence (Fig. 1b) is given by the Cosine Transformation Type II (Strang, 1999): $\Gamma_{v,j} = \pm\cos((v + 1/2)j\pi/n)$, where we must choose the same sign per j. The $\pm$ is due to the *sign ambiguity* (2) of the eigenvector and, thus, we cannot distinguish the embedding of, e.g., the first and last node. Note that in traditional applications of the Cosine Transformation, it is possible to identify the first token and fix the sign. However, for general graphs, it is not that easy to resolve the sign ambiguity (e.g., multiple sink and source nodes). Thus, in positional encodings, we typically use an arbitrary sign for each $\boldsymbol{\gamma}$ (Dwivedi & Bresson, 2021), and are not able to distinguish direction.

3. Directional Spectral Encodings

The Magnetic Laplacian is a generalization of the combinatorial Laplacian that encodes the direction with complex numbers. We then use its eigenvectors for a structure-aware positional encoding that acknowledges the directedness.

We define the **Magnetic Laplacian** (Furutani et al., 2020)

$$\boldsymbol{L}_U^{(q)} := \boldsymbol{D}_s - \boldsymbol{A}_s \odot \exp\left(i\boldsymbol{\Theta}^{(q)}\right) \quad (4)$$

with Hadamard product $\odot$, element-wise exp, $i = \sqrt{-1}$, $\Theta_{u,v}^{(q)} := 2\pi q(A_{u,v} - A_{v,u})$, and potential $q \ge 0$. Recall, $\boldsymbol{D}_s$ is the symmetrized degree matrix and $\boldsymbol{A}_s$ the symmetrized adjacency matrix. The Magnetic Laplacian is a Hermitian matrix since it is equal to its conjugate transpose $\boldsymbol{L}^{(q)} = (\bar{\boldsymbol{L}}^{(q)})^\top$ and, thus, comes with complex eigenvectors $\boldsymbol{\Gamma} \in \mathbb{C}^{n \times n}$. Eq. 4 is equivalent to the combinatorial Laplacian for $q = 0$. Moreover, if the graph is undirected, we recover the combinatorial Laplacian for any finite $q \in \mathbb{R}$.
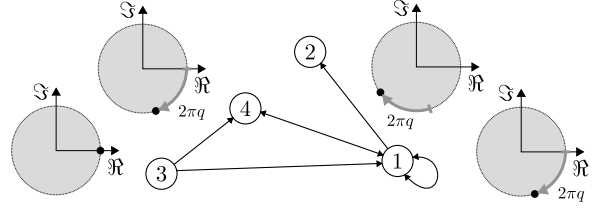

Figure 3: First eigenvec. $\boldsymbol{\gamma}_0$ of Magnetic Laplacian (Eq. 4).

The $\exp\left(i\boldsymbol{\Theta}^{(q)}\right)$ term in Eq. 4 encodes the edge direction. It resolves to 1 if $A_{u,v} = A_{v,u}$ and, otherwise, to $\exp(\pm i2\pi q)$, with the sign encoding the edge direction. The *potential* q determines the ratio of real and imaginary parts. Recall that $\exp(i\alpha) = \cos(\alpha) + i\sin(\alpha)$. Conversely, $\angle(\Gamma_{u,0}) = \text{arctan2}(\Im(\Gamma_{u,0}), \Re(\Gamma_{u,0}))$ with real / imag. value $\Re$ / $\Im$. For illustration, we next give the Magnetic Laplacian for a sequence with $q = 0$ and $q = 1/4$ (Eq. 5 & 6), as well as their first eigenvector in Fig. 1b and 1a.

$$\boldsymbol{L}_U^{(0)} = \begin{bmatrix} 1 & \text{-}1 & \cdots & 0 & 0 \\ \text{-}1 & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 2 & \text{-}1 \\ 0 & 0 & \cdots & \text{-}1 & 1 \end{bmatrix} \quad (5) \quad \boldsymbol{L}_U^{(1/4)} = \begin{bmatrix} 1 & \text{-}i & \cdots & 0 & 0 \\ i & 2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 2 & \text{-}i \\ 0 & 0 & \cdots & i & 1 \end{bmatrix} \quad (6)$$

In our experiments, we use the degree-normalized counterpart $\boldsymbol{L}_N^{(q)} := \boldsymbol{I} - \left(\boldsymbol{D}_s^{-1/2}\boldsymbol{A}_s\boldsymbol{D}_s^{-1/2}\right) \odot \exp\left(i\boldsymbol{\Theta}^{(q)}\right)$ that we find to result in slightly superior performance.

Directedness. We next illustrate how the *eigenvectors* of the Magnetic Laplacian $\boldsymbol{L}_U^{(q)}$ encode direction. For the special case of a sequence (Fig. 1a), the eigenvectors are given by $\Gamma_{v,j}^{(q)} = c\exp(\text{-}i2\pi qv)\Gamma_{v,j}^{(0)} = c\exp(\text{-}i2\pi qv)\cos((v + 1/2)j\pi/n)$ with $c \in \mathbb{C} \setminus \{0\}$. This corresponds to the Cosine Transformation Type II (see § 2) with additional factor $\exp(\text{-}i2\pi qv)$ that encodes the node position v. Importantly, the eigenvectors of the Magnetic Laplacian also encode the directionality in arbitrary (directed) graph topologies, where each *directed* edge (u, v) encourages a phase difference in the (otherwise constant) first eigenvector $\boldsymbol{\gamma}_0$, i.e., between $\Gamma_{u,0}$ and $\Gamma_{v,0}$. For simple cases with $\boldsymbol{L}_U^{(q)}$, as in Fig. 3, each directed edge induces a rotation of $2\pi q$ while each undirected edge synchronizes the rotation of the adjacent nodes. Note that self-loops are assumed to be undirected and only affect the symmetrically degree-normalized $\boldsymbol{L}_N^{(q)}$. In this example, the one-hop neighbors of node 3, namely nodes 1 and 4, have a relative rotation of $2\pi q$, while the two-hop neighbor 2 has a relative rotation of $4\pi q$. In general, the first normalized eigenvector $\boldsymbol{\gamma}$ minimizes the Rayleigh quotient

$$\min_{\boldsymbol{x} \in \mathbb{C}} \frac{\bar{\boldsymbol{x}}^\top \boldsymbol{L}_U^{(q)}\boldsymbol{x}}{\bar{\boldsymbol{x}}^\top\boldsymbol{x}} = \frac{1}{2}\sum_{(u,v) \in E_s} |\Gamma_{u,0} - \Gamma_{v,0}\exp(i\Theta_{u,v}^{(q)})|^2,$$

$$(7)$$

Thus, the eigenvectors trade off conflicting edges, e.g., if multiple (directed) routes of different lengths exist between nodes u and v. For more details, see § D.

The potential q determines the strength of the induced phase shift by each edge. Thus, q plays a similar role as the lowest frequency in sinusoidal positional encodings (typically $1/2\pi 10{,}000$). Following the convention of sinusoidal encodings, one could fix q to an appropriate value for the largest expected graphs. However, scaling potential q with the number of nodes n and the amount of directed edges leads to slightly superior performance in our experiments. Specifically, we choose $q = q'/d_{\mathcal{G}}$ with *relative potential* q' and graph-specific normalizer $d_{\mathcal{G}}$. This normalizer is an upper bound on the number of directed edges in a simple path $d_{\mathcal{G}} = \max(\min(\vec{m}, n), 1)$ with the number of *purely directed edges* $\vec{m} = |\{(u, v) \in E \,|\, (v, u) \notin E\}|$ and is motivated by Eq. 7 (see § D.1). We typically choose $q' \in \{0.1, 0.25\}$ and empirically verify this in Fig. 9 where it is among the best. For high values of q' the performance drops severely (corresponds to absolute $q > 0.05$).

Scale and rotation. Eigenvectors are not unique and we normalize them by a convention. We visualize the first eigenvector *after applying our normalization* for different graphs in Fig. 1 & A.1. One source of ambiguity is its scale and rotation. If γ is an eigenvector of L then so is $c\gamma$, even if $c \in \mathbb{C} \setminus \{0\}$ (proof: $cL\gamma = c\lambda\gamma \implies L(c\gamma) = \lambda(c\gamma)$). For real symmetric matrices, there is the convention to choose $c \in \mathbb{R}$ s.t. Γ is orthonormal ($\Gamma^\top\Gamma = I$). Similarly, (1) we choose $|c|$ s.t. Γ is unitary ($\bar{\Gamma}^\top\Gamma = I$). Moreover, if not using a sign-invariant architecture, as described below, (2) we determine the sign of each eigenvector such that the maximum real magnitude is positive. This resolves the sign-ambiguity up to ties in the maximum real magnitude and numerical errors. (3) we fix the rotation. If possible for the task at hand, we use the graph's distinct "root" node u. For example, in function name prediction, the root node marks the start of the function definition. Alternatively, we use the foremost (source) node u as root, i.e., the node with maximum phase shift in the first eigenvector $u = \arg\max_v \angle(\Gamma_{v,0})$. In both cases, we then rotate all eigenvectors, such that the phase shift $\angle(\Gamma_{u,j})$ is 0 for all $j \in \{0, 1, \ldots, n-1\}$. Due to the rotation in (3), our normalization is best suited for graphs with root/source node(s). For details, see § D.2.

MagLapNet. Inspired by prior approaches (Lim et al., 2022; Kreuzer et al., 2021), we also preprocess eigenvectors before using them as positional encodings (Fig. 4b) to obtain a structure-aware transformer (Fig. 4a). We consider the eigenvectors associated with the k lowest eigenvalues $\Gamma_{:,:k-1}$ and treat k as hyperparameter. We study two architecture variants for processing the eigenvectors after stacking real and imaginary components: (a) a model that ignores the sign-invariance $f_{\text{elem}}(\gamma)$ and (b) the sign-invariant Sign-Net (Lim et al., 2022) that processes each eigenvector as $f_{\text{elem}}(-\gamma_j) + f_{\text{elem}}(\gamma_j)$, where f_{elem} is permutation equivariant over the nodes, like a point-wise Multi-Layer Perceptron (MLP) or GNN. However, when utilizing an MLP, we

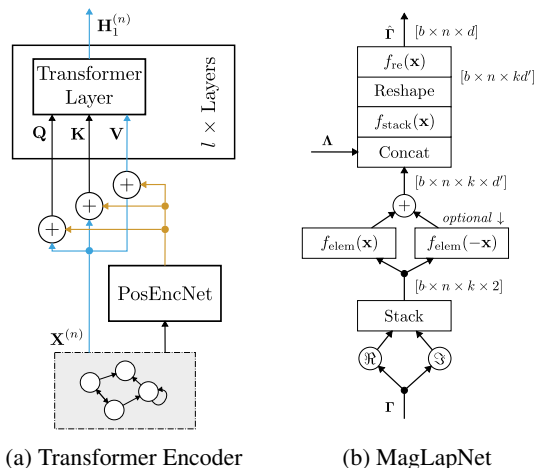

(a) Transformer Encoder   (b) MagLapNet

Figure 4: (a) shows a transformer encoder operating on a graph with omitted residual connection. (b) is one specific instantiation of the (optional) "PosEncNet" using the eigenvectors of the Magnetic Lap (see "MagLapNet" paragraph) with batch size b. See § 4 for random walk encodings.

observe that choice (a) yields superior performance. This outcome can be attributed to several factors, including our aforementioned selection of the sign (see above) and the characteristic of a point-wise MLP to disregard relative differences in γ. Note that we always process the first eigenvector as $f_{\text{elem}}(\gamma_j)$ since we fully resolve its sign ambiguity.

Thereafter, we apply LayerNorm, Self-Attention, and Dropout. Similar to Kreuzer et al. (2021), we apply self-attention independently for each node u over its k eigenvector embeddings. In other words, for each node, we apply self-attention over a set of k tokens. This models the node-wise interactions between the eigenvectors, i.e., $(\Gamma_{u,:k-1})$ for node u. The last reshape stacks each node's encoding, and the MLP f_{re} matches the transformer dimensions.

4. Directional Random Walks

An alternative principled approach for encoding node positions in a graph is through random walks. Li et al. (2020) have shown that such positional encodings can provably improve the expressiveness of GNNs, and such random walk encodings have been applied to transformers as well (Mialon et al., 2021). Interestingly, random walks generalize, e.g., shortest path distances via the number of steps required for a non-zero landing probability. However, naïvely applying random walks to directed graphs comes with caveats.

Random walks on graphs. A k-step random walk on a graph is naturally expressed via the powers T^k of the transition matrix $T = AD_{\text{out}}^{-1}$. In each step, the random walk proceeds along one of the outgoing edges with equal probability or probability proportional to the edge weight. We then

obtain the landing probability $(T^k)_{u,v}$ at node u if starting from node v. Note that even in connected graphs, we might have node pairs v, u that have zero transition probability regardless of k. Thus, the naïve application of random walks for positional encodings on directed graphs is not ideal.

Directedness. To overcome the issue of only walking in *forward* direction and in contrast to Li et al. (2020), we additionally consider the *reverse* direction $R = A^\top D_{\mathrm{in}}^{-1}$. Additionally, we add self-loops to sink nodes (nodes with zero out or in degree for T or R, respectively). This avoids that A might be nilpotent and ensures that the landing probabilities sum up to one. We then define the positional encoding for node v as $\zeta(v|\mathcal{G}) = f_{\mathrm{rw}}^{(1)}(\mathrm{AGG}(\{\zeta(v|u) \mid u \in V\}))$, where $\zeta(v|u) = f_{\mathrm{rw}}^{(2)}[(R^k)_{v,u}, \ldots, (R^2)_{v,u}, R_{v,u}, T_{v,u}, (T^2)_{v,u}, \ldots, (T^k)_{v,u}]$ and AGG performs summation. $f_{\mathrm{rw}}^{(1)}$ and $f_{\mathrm{rw}}^{(2)}$ is an MLP.

Large distances. A large amount of random walk steps k is expensive and for a sufficiently large k the probability mass concentrates in sink nodes. Thus, the random walk positional encodings are best suited for capturing short distances. For the global relations, we extend $\zeta(v|u)$ with a forward and reverse infinite step random walk, namely Personalized Page Rank (PPR) (Page et al., 1999). Importantly, PPR includes the restart probability p_r to jump back to the starting node u and has closed form solution $p_r(I - (1 - p_r)T)^{-1}$.

We provide an overview of our positional encodings in § F and discuss computational cost/complexity in § H.

5. Positional Encodings Playground

We next assess the efficacy of our two directional structure-aware positional encodings. As there is no (established) way of assessing positional encodings standalone, we rely on downstream tasks. In our first task, we verify if the encodings are predictive for distances on graphs.

Tasks. We hypothesize that a good positional encoding should be able to distinguish between ancestors/successors and should have a notion of distance on the graph. To cope with general graphs, instead of ancestor/successor nodes, we predict if a node is reachable, acknowledging the edge directions. As distance measures, we study the prediction of adjacent nodes as well as the *directed* and *undirected* shortest path distance. With *undirected* shortest path distance, we refer to the path length on the symmetrized graph, and in both cases we ignore node pairs for which no path exists. In summary, we study pair-wise binary classification of *(1) reachability* and *(2) adjacency* as well as pair-wise regression of *(3) undirected distance* and *(4) directed distance*.

Models. We use a vanilla transformer encoder (Vaswani et al., 2017) with positional encodings (see Fig. 4a). We compare our Magnetic Laplacian (ML) positional encod-

	F1		RMSE		F1		RMSE		
Lap. (basln)	0.63	0.63	0.23	0.51	0.75	0.62	0.27	1.96	val.
	0.53	0.53	0.26	0.54	0.73	0.49	0.31	2.08	test
SVD (basln)	1.00	1.00	0.83	0.38	1.00	1.00	1.02	1.64	val.
	1.00	1.00	0.97	0.45	0.97	1.00	1.12	1.86	test
ML (ours)	1.00	1.00	0.22	0.33	1.00	1.00	0.27	0.93	val.
	1.00	1.00	0.25	0.38	1.00	1.00	0.31	1.06	test
RW (ours)	0.97	0.97	1.22	0.65	1.00	0.97	1.00	1.24	val.
	0.95	0.95	1.33	0.68	0.99	0.94	1.06	1.36	test
	reach. (1)	adj. (2)	u. dist (3)	d. dist (4)	reach. (1)	adj. (2)	u. dist (3)	d. dist (4)	
	(a) Directed Acyclic Graph				(b) Regular Directed Graph				

Figure 5: Positional encodings playground results. Dark green encodes the best scores and bright green bad ones. For F1 score high values are better and for RMSE low values.

ings w/o SignNet (§ 3) with our direction-aware random walk (RW) of § 4 and eigenvectors of the combinatorial Laplacian (Lap.) from § 2. The eigenvectors of the combinatorial Laplacian are preprocessed like the ones of the Magnetic Laplacian (Fig. 4b), except that the "Stack" step is superfluous due to the real eigenvectors. Additionally, we compare to the SVD encodings of Hussain et al. (2022) that perform a low-rank decomposition of the (directed) adjacency matrix. Moreover, with the goal of obtaining general positional encodings, we do not study any heuristics that can be considered "directional". For example, if solely considering trees, it might be sufficient to add features for source and sink nodes next to undirected positional encodings.

All studied tasks are instances of link prediction or link regression where the predictions are of shape $n \times n$ (ignoring the disconnected pairs of nodes in distance regression), modeling the relative interactions between nodes. For this, we broadcast the resulting node encodings $H_l^{(n)}$ (see Fig. 4a) of the sender and receiver nodes and stack a global readout. Thereafter, we use a shallow MLP with 3-layers in total and task-dependent output activation (softmax or softplus).

Setup. We use cross-entropy for classification and L^2 loss for regression. We assess classification with the F1 score and regression with the Root Mean Squared Error (RMSE). We sample Erdős-Rènyi graphs with equally probable average degree $\{1, 1.5, 2\}$ and, additionally, Directed Acyclic Graphs (DAGs), where we draw the average degree from $\{1, 1.5, 2, 2.5, 3\}$ to account for the greater sparsity. Then, we extract the largest (weakly) connected component. For the regression tasks, we sample graphs with 16 to 63, 64 to 71, and 72 to 83 nodes for train, validation, and test, respectively. To counteract a too-severe class imbalance, we choose 16 to 17, 18 to 19, and 20 to 27 nodes for the classification tasks, respectively. We report the average over three random reruns. We sample 400,000 training instances and for test/validation 2,500 for each number of nodes n.

Results. In Fig. 5, we show the performance of the positional encodings for the four curated tasks. We see that the eigenvectors of the Magnetic Laplacian outperform the eigenvectors of the combinatorial Laplacian for all measures that rely on directedness. For *(3) undirected distance*, they perform similarly well. On the classification tasks *(1)* & *(2)*, the random walk encodings perform comparably to the Magnetic Laplacian. However, random walk encodings are less predictive for regressing distances. In general, random walks seem to show their strength for tasks that are well-aligned with their design. For example, a random walk with $k = 1$ resembles the adjacency matrix that we predict in task *(2)*. However, the random walk encodings only achieve mediocre scores for *(3) undirected distance* prediction.

The Magnetic Laplacian encodings consistently outperform the SVD encodings and achieve an up to 4 times lower RMSE. Nevertheless, the SVD encodings are a surprisingly strong baseline on the DAG (Fig. 5a), where they even outperform the random walk encodings. However, on general directed graphs (Fig. 5b), the random walk encodings outperform the SVD encodings on *(4) directed distance* with a roughly 30% lower RMSE. Moreover, we did not achieve similarly strong performance with SVD in the other studied tasks. For example, in the sorting network task (see § 6), we were not able to achieve meaningful performance after a basic hyperparameter search. This might be due to the undesirable properties low-rank SVD positional encodings have for certain graph structures (see § D.4).

In § I, we provide additional comparisons. These include (a) a comparison to GNNs and (b) the study of relative random walk encodings. For (b), we use the pair-wise encodings $\zeta(v|u)$ before the node-level aggregation and add the $n \times n \times d$ encodings to the attention matrix before applying the softmax. The relative random walk encodings can be understood as a generalization of the pair-wise shortest distances used by Ying et al. (2021). Moreover, in § J, we give a hyperparameter study of the random walk encodings and compare them to the (forward) random walk encodings of Li et al. (2020) that are designed for undirected graphs.

6. Application: Sorting Networks

Sorting networks (Knuth, 1973) are a certain class of comparison-based algorithms that have the goal of sorting any input sequence of fixed size with a static sequence of comparators. Sorting networks are a particularly interesting application since they mark the middle ground between logical statements and source code. Specifically, the sequence of comparators reflects a sequence of program instructions while asserting their correctness is related to satisfiability (Knuth, 1968). We use this task to make the implications of symmetrization (Laplacian encodings) and sequentialization (sinusoidal encodings) explicit.

We consider sorting networks that consist of a sequence of conditional exchange instructions. In Fig. 6a, the p horizontal lines represent the variables that are to be sorted, and the vertical lines are comparators that sort the connected variables. Thus, a sorting network can also be expressed by n `v_i, v_j = sorted((v_i, v_j))` statements, where `v_i` and `v_j` are two of the p variables, i.e. $i, j \in \{0, 1, \ldots, p-1\}$. Our graph construction (Fig. 6b) treats every instruction as a node with i and j as features (sinusoidal encoding). If a node operates on indices i and j, we add an edge from the last occurrences of i and j (if there are any). Thus, in this data-flow graph, the in- and outdegree equal two for all nodes except source and sink nodes.

Directed graph vs. sequence. An important implication is that each topological sort of the directed graph is an equivalent "program", i.e., a different ordering of statements that yields the same result. In Fig. 7, we show the number of topological sorts over the sequence lengths p for a type of compact and deterministically constructed sorting network. For such networks and a sequence length of just 8, the number of equivalent sequentializations already exceeds 1 million (see also § L). Note that in the worst case, a directed graph has $n!$ topological sorts. Therefore, representing directed graphs as sequences can introduce a huge amount of arbitrary orderedness. In contrast to sequential modeling, a graph-based representation can significantly reduce the size of the effective input space.

Symmetrization hurts. There exist correct and incorrect sorting networks that map to the same undirected graph. Hence, a model that uses undirected graphs cannot distinguish these cases. For example, the *correct sorting network* for length three with comparators $[(0, 2), (0, 1), (1, 2)]$ and its reversed version (*incorrect*) map to the same undirected graph. In summary, symmetrizing may hurt expressiveness.

Dataset. We construct a dataset consisting of 800,000 training instances for equally probable sequence lengths $7 \leq p_{\text{train}} \leq 11$, generate the validation data with $p_{\text{val}} = 12$, and assess performance on sequence lengths $13 \leq p_{\text{test}} \leq 16$. We construct the sorting networks greedily until we have

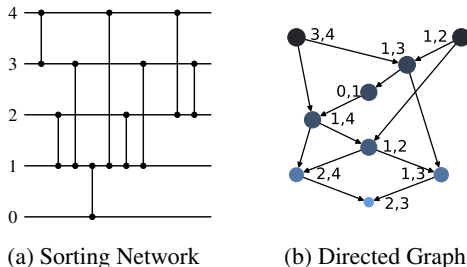(a) Sorting Network   (b) Directed Graph

Figure 6: (a) Common illustration for a sorting network with sequence length $p = 5$ and (b) as directed graph.
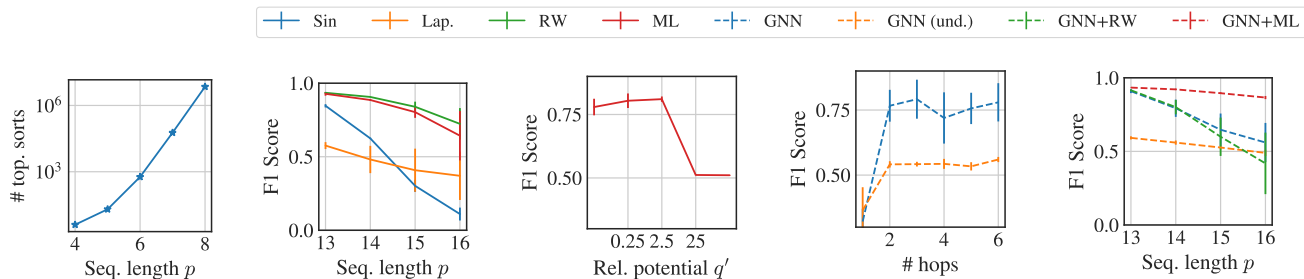
Figure 7: # topological sorts for Batcher even odd mergesort.

Figure 8: Comparing positional enc. over length p (sorting netw.).

Figure 9: Relative pot. $q = q'/\max(\min(\vec{m},n),1)$ with $k = 25$ eigenvec.

(a) # mess. pass. steps

(b) GNN only

Figure 10: (a) # message passing steps of GNN and (b) for GNN w/ and w/o pos. enc.

a correct sorting network. For this, we draw a random pair of comparators, excluding immediate duplicates and comparators between inputs that are already sorted. We then generate an incorrect example by omitting the last comparator (i.e., train is balanced). This procedure is similar to datasets for the related task of satisfiability (Selsam et al., 2019). Moreover, we add additional incorrect sorting networks by reversing the directions of the correct networks to make the test sets more challenging. Thus, the test and validation data consist of 1/3 correct sorting networks (20,000) and 2/3 of incorrect ones (40,000). Therefore, the task is to generalize the correctness prediction to longer sequences and reversed (slightly out of distribution) sorting networks. See § K for more details on the dataset construction.

Empirical Evaluation. We follow the setup of § 5 and include sinusoidal positional encodings (Sin). As shown in Fig. 8, the eigenvectors of the Magnetic Laplacian (ML) perform comparably to the random walk encodings. Both outperform the other positional encodings by a large margin. This shows that, without bells and whistles, the Magnetic Laplacian and random walk encodings provide a transformer a considerable structural awareness for directed graphs (see Fig. 4a). On the other hand, the eigenvectors of the combinatorial Laplacian barely outperform the naïve baseline that randomly chooses a class based on the prior probabilities.

Sinusoidal positional encodings perform well for sequences close to the training data but do not generalize well to longer sequences. The lacking generalization might be due to the much larger input space if measuring input space size in terms of unique inputs for the transformer.

GNN. In Fig. 10a, we study the performance of a GNN (following Battaglia et al. (2018), see § 7 & § G for specifics) with mean readout over the number of message passing steps. Since the improvements diminish for more than two message passing steps, we report the results in Fig. 10b using three message passing steps. We compare a direction-aware "GNN" and direction unaware "GNN (und.)". Ex-

pectedly, the directional information is important for generalization also in the context of GNNs. Note that the directional GNN performs on par with a transformer encoder with the Magnetic Laplacian encodings for sequence length 13. However, the GNN generalizes slightly worse to longer sequences. Motivated by Li et al. (2020), we additionally pair the GNN with positional encodings and find that the Magnetic Laplacian eigenvectors can also help a GNN's generalization. On the other hand, the random walk encodings struggle to generalize to longer sequences and harm performance. We hypothesize that the Magnetic Laplacian encodings provide complimentary information while the random walk encodings are similar to message passing.

7. Application: Function Name Prediction

We study function name prediction since it is an established task in the graph learning community (Hu et al., 2020) where the direction of edges influences the true label, and transformers seem to have an edge over GNNs. Similar to sorting networks, each program represents a specific ordering of statements, and there can be many equivalent programs via reordering statements. Thus, it is surprising that graphs for source code used for machine learning retain the sequential connections between instructions. In other words, these graphs "only" enrich sequential source code (here, add a hierarchy). For example, the Open Graph Benchmark Code2 dataset represents the 450,000 functions with its Abstract Syntax Tree (AST) and *sequential connections*. Since the input space of sequences can be much larger than the input space of directed graphs (see § 6), for some tasks, such a graph construction is an unfortunate choice.

Robustness. We trained the state-of-the-art model[1] on the Open Graph Benchmark Code2 dataset, called Structure Aware Transformer (SAT) (Chen et al., 2022). We then used OGB's code to generate multiple graph representations of

---

[1]Shortly before submission, Luo (2022) proposed in their preprint a new model with 0.4% higher F1 test score.

Prediction: `unknown`

```python
def f1_score(pred, label):
  correct = pred == label
  tp = (correct & label).sum()
  fn = (~correct & pred).sum()
  fp = (~correct & ~pred).sum()
  precision = tp / (tp + fp)
  recall = tp / (tp + fn)
  return (
    2 * (recall * precision) /
    (recall * precision)
  )
```

Prediction: `accuracy`

```python
def f1_score(pred, label):
  correct = pred == label
  tp = (correct & label).sum()
  fn = (~correct & pred).sum()
  fp = (~correct & ~pred).sum()
  precision = tp / (tp + fp)
  recall = tp / (tp + fn)
  return (
    2 * (precision * recall) /
    (precision * recall)
  )
```

Prediction: `precision`

```python
def f1_score(pred, label):
  correct = pred == label
  tp = (correct & label).sum()
  fn = (~correct & ~pred).sum()
  recall = tp / (tp + fn)
  fp = (~correct & pred).sum()
  precision = tp / (tp + fp)
  return (
    2 * (precision * recall) /
    (precision * recall)
  )
```

Figure 11: State-of-the-art model on OGB Code2 is susceptible to meaningless permutations (highlighted in yellow) due to OGB Code2's graph construction. The code was minimally modified for better layout.

functions, where we reordered some statements s.t. the functionality is preserved. In Fig. 11, we show that the state-of-the-art model using OGB's graph construction is susceptible to these semantics-preserving permutations of the source code. Moreover, the number of possible reorderings can be surprisingly high. E.g., if constructing a data-flow DAG, the F1 score function of Fig. 11 has 16 topological sorts. Further considering commutativity for ==, &, +, and *, we find 4,096 possibilities to write this seemingly simple function.

Our graph construction maps all these 4,096 possibilities to the very same directed graph. Our graph construction is greatly inspired by Bieber et al. (2022), although they also connect most instructions sequentially. While we do avoid this sequentialism, we leverage their static code analysis for a graph construction that handles the sharp bits like if-else, loops, and exceptions. The most significant differences to Bieber et al. (2022) are: (a) We construct a DAG for each "block" (e.g., body of if statement) that reflects the dependencies between instructions. We then connect the statements between blocks considering the control flow. (b) we address the commutative properties for basic Python operations via edge features; (c) we do not reference the sequence of tokenized source code; (d) we omit the (in our case) unnecessary "last read" edges; (e) we construct the graph similarly to OGB Code2 for comparability. For example, we aggregate child nodes containing only attributes into their parent's node attributes. We provide details and a side-by-side comparison to an OGB Code2 graph in § M.

Assumptions. While the right equi-/invariances are task-dependent, we argue that for high-level reasoning tasks, including function name prediction or correctness prediction, the mentioned reorderings should not affect the true label. Nevertheless, e.g., for predicting the runtime of a program, reorderings can have an impact. Moreover, we assume that non-class-member methods are side-effect-free. For example, this includes reordering print statements. Even though this will result in a different output stream, we argue that these differences are typically not vital. Moreover, since we construct the graph with lexicographical static code analysis, we do this on a best-effort basis and do not capture all dynamic runtime effects. Last, our eigenvector-based positional encodings are only permutation equivariant in the absence of repeated eigenvalues (see § D for details).

Empirical Evaluation. In Table 1, we report the results on OGB Code2. Here we also compare to the Structure Aware Transformer (SAT) of Chen et al. (2022). SAT is a hybrid transformer w/ GNN for query and key and was the prior state of the art. We illustrate the architecture in Fig. G.1b. If we omit the GNN, we recover the vanilla transformer encoder Fig. 4a (plus degree-sensitive residual). We improve the current state-of-the-art model with a number of small tricks (i.e., no new positional encoding yet). Our SAT++ (w/ GNN) improves the F1-score by 1.66% (relatively 8.6%). Besides smaller changes like replacing ReLU with GeLU activations, we most notably (1) add dropout on the sparsely populated node attributes and (2) offset the softmax score to adjust for the class imbalance of the special tokens for *unknown words* as well as *end of sequence*. We also replace the GCN with a three-layer GNN following Battaglia et al. (2018) (w/o global state). The edge and node embeddings are updated sequentially with independently aggregated forward and backward. Then, we concatenate the embeddings we obtained after each message passing step and apply an MLP with two layers. For details on the GNN see § G.

Our graph construction ("data-flow" in Table 1) consistently increases the predictive performance. We do not report results w/o GNN and solely w/ AST depth positional encodings because this approach does not make use of the enhanced graph structure. Our graph construction raises the F1 score by almost 0.58% (relatively 2.8%), if using the SAT++ architecture (w/ GNN) with AST depth encodings. Note that the gain partially stems from the improved edge features. In a dedicated experiment, we compare the effect of our data-flow edges with the sequential edges of Bieber et al. (2022) and find that our edges contribute to an $\approx 0.1\%$ greater F1 score (model uses Magnetic Laplacian encodings). Nevertheless, we want to emphasize that our graph construction yields robustness gains w.r.t. certain reorderings of statements in the source code (see Fig. 11).

Table 1: Results on the Open Graph Benchmark Code2 dataset. The first two rows correspond to prior work. All other approaches are our contribution. We report the average and error of the mean over 10 reruns. Best is bold.

	Position. Enc.	GNN	Test F1-Score	Val. F1-Score	
Sequen.	AST depth	✗	16.70±0.05	15.46±0.06	**Prev.**
		✓	19.37±0.09	17.73±0.07	
		✗	19.09±0.10	17.68±0.06	
		✓	21.03±0.07	19.38±0.07	
Data-flow	AST depth	✓	21.61±0.12	19.79±0.11	**Ours**
	Random walk	✗	19.34±0.08	**17.96±0.05**	
		✓	21.82±0.20	20.03±0.17	
	Magnetic Lap.	✗	**19.43±0.03**	17.83±0.05	
		✓	**22.22±0.10**	**20.44±0.06**	

Hybrid. The Magnetic Laplacian also helps in the hybrid transformer GNN architecture. *Our SAT++ with Magnetic Laplacian positional encodings (SignNet w/ GNN) marks the new state of the art on the Code2 dataset, outperforming SAT by 2.85% (relatively 14.7%).* The Random Walk positional encodings only slightly improve performance. For the Code2 graphs, the GNN for query and key appears to be of great importance. We hypothesize that this is due to the sparsely populated node features. Only a few nodes are attributed, and additionally, the permitted vocabulary is restrictive. The local message passing might spread the information to neighboring nodes to adjust for this sparseness. Moreover, w/o GNN, we do not make use of edge features.

Dataset challenges. The node attributes (e.g., variable names) and function names are only lightly preprocessed. For example, for perfect performance, one needs to distinguish singular and plural method names. Although singular/plural semantically makes a difference, the naming consistency is lacking for the 450k functions taken from github. We refrain from adjusting the dataset accordingly to maintain comparability to prior work.

8. Related Work

Directed graphs appear in various applications and can be crucial to appropriately model the input data, also in well-established domains for GNNs such as citation networks (Rossi et al., 2023). An important related GNN for directed graphs is MagNet (Zhang et al., 2021) since it used the Magnetic Laplacian within its message passing. We compare to MagNet in § I on the playground tasks.

Positional encodings. Prior work on positional encodings includes traditional graph metrics, like shortest path distances (Guo et al., 2021). Related to the distance from a node to the AST root node in the OGB Code2 dataset (see § 7), Luo (2022) proposes sinusoidal positional encodings for DAGs leveraging their partial order. An alternative form

of spectral encodings, based on Singular Value Decomposition (SVD), was used for positional encodings (Hussain et al., 2022). The authors argue that these encodings also include directed graphs; however, they do not verify this choice, and the SVD of the adjacency matrix has undesirable properties (see § D.4). We compare the SVD encodings in § 5 on the tasks of the positional encodings playground.

We include a discussion of Laplacians for directed graphs in § C. For an in-depth overview and a how-to for graph transformers, we refer to Min et al. (2022), Müller et al. (2023) and Rampášek et al. (2022). They also provide an overview of graph transformers that rethink attention architectures for structure-awareness like (Dwivedi & Bresson, 2021; Mialon et al., 2021; Chen et al., 2022; Kim et al., 2022; Hussain et al., 2022; Diao & Loynd, 2022).

Graph construction. There are many attempts to enrich source code in a graph-structured manner for machine learning (Allamanis et al., 2018; Cummins et al., 2020; Guo et al., 2021; Bieber et al., 2022). However, they all retain the sequentialism of the underlying source code. As we see in Fig. 11, this can lead to a fragile representation w.r.t. to semantically meaningless reorderings. Such reorderings are a novel perspective on the robustness of models for source code (e.g., see (Jha & Reddy, 2022; Yefet et al., 2020)). However, the relationship between a directed graph and its sequentializations is well-known in task scheduling. Similarly, an appropriate graph construction may improve the robustness of transformers if applied to other combinatorial optimization problems (Geisler et al., 2022; Kool et al., 2019) than correctness prediction of sorting networks.

9. Conclusion

We propose positional encodings for directed graphs based on the Magnetic Laplacian and random walks. Both positional encodings can help transformers to gain considerable structure awareness and show complementary strengths in our experiments. We argue that direction-aware positional encodings are an important step towards true multi-purpose transformers universally handling undirected and directed graphs. We show that directedness can be central for the semantics in the target domain and that directed graphs can drastically lower the effective input dimensionality (i.e., many instances map to one graph).

References

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations, ICLR*, 2018.

Bandeira, A. S., Singer, A., and Spielman, D. A. A Cheeger Inequality for the Graph Connection Laplacian. *SIAM J. Matrix Anal. Appl.*, 2013.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y., and Pascanu, R. Relational inductive biases, deep learning, and graph networks, In *arXiv*, 2018.

Belkin, M. and Niyogi, P. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 2003.

Bieber, D., Shi, K., Maniatis, P., Sutton, C., Hellendoorn, V., Johnson, D., and Tarlow, D. A Library for Representing Python Programs as Graphs for Machine Learning, In *arXiv*, 2022.

Bojchevski, A., Klicpera, J., Perozzi, B., Kapoor, A., Blais, M., Rózemberczki, B., Lukasik, M., and Günnemann, S. Scaling Graph Neural Networks with Approximate PageRank. *International Conference on Knowledge Discovery and Data Mining, KDD*, 2020.

Brock, A., De, S., Smith, S. L., and Simonyan, K. High-Performance Large-Scale Image Recognition Without Normalization. In *International Conference on Machine Learning, ICML*, 2021.

Bronstein, M. M., Bruna, J., Cohen, T., and Veličković, P. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges, In *arXiv*, 2021.

Chen, D., O'Bray, L., and Borgwardt, K. Structure-Aware Transformer for Graph Representation Learning. *International Conference on Machine Learning, ICML*, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating Large Language Models Trained on Code, In *arXiv*, 2021.

Choromanski, K. M., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J. Q., Mohiuddin, A., Kaiser, L., Belanger, D. B., Colwell, L. J., and Weller, A. Rethinking Attention with Performers. In *International Conference on Learning Representations, ICLR*, 2020.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. ProGraML: Graph-based Deep Learning for Program Optimization and Analysis, In *arXiv*, 2020.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Neural Information Processing Systems, NeurIPS*, 2017.

Diao, C. and Loynd, R. Relational Attention: Generalizing Transformers for Graph-Structured Tasks, In *arXiv*, 2022.

Dwivedi, V. P. and Bresson, X. A Generalization of Transformer Networks to Graphs. *Deep Learning on Graphs at AAAI Conference on Artificial Intelligence*, 2021.

Fanuel, M., Alaíz, C. M., and Suykens, J. A. K. Magnetic eigenmaps for community detection in directed networks, In *arXiv*, 2016.

Fanuel, M., Alaíz, C. M., Fernández, , and Suykens, J. A. K. Magnetic Eigenmaps for the visualization of directed networks. *Appl. Comput. Harmon. Anal.*, 2018.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP*, 2020.

Furutani, S., Shibahara, T., Akiyama, M., Hato, K., and Aida, M. Graph Signal Processing for Directed Graphs Based on the Hermitian Laplacian. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD*, 2020.

Geisler, S., Sommer, J., Schuchardt, J., Bojchevski, A., and Günnemann, S. Generalization of Neural Combinatorial Solvers Through the Lens of Adversarial Robustness. In *International Conference on Learning Representations, ICLR*, 2022.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. GraphCodeBERT: Pre-training Code Representations with Data Flow. *International Conference on Learning Representations, ICLR*, 2021.

He, Y., Perlmutter, M., Reinert, G., and Cucuringu, M. MSGNN: A Spectral Graph Neural Network Based on a Novel Magnetic Signed Laplacian. In *Learning on Graphs Conference*, 2022.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Neural Information Processing Systems, NeurIPS*, 2020.

Hussain, M. S., Zaki, M. J., and Subramanian, D. Global Self-Attention as a Replacement for Graph Convolution. In *International Conference on Knowledge Discovery and Data Mining, KDD*, 2022.

Jha, A. and Reddy, C. K. CodeAttack: Code-based Adversarial Attacks for Pre-Trained Programming Language Models, In *arXiv*, 2022.

Kim, J., Nguyen, T. D., Min, S., Cho, S., Lee, M., Lee, H., and Hong, S. Pure Transformers are Powerful Graph Learners. In *Neural Information Processing Systems, NeurIPS*, 2022.

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

Kitaev, N., Kaiser, , and Levskaya, A. Reformer: The Efficient Transformer. *International Conference on Learning Representations, ICLR*, 2020.

Knuth, D. E. *The art of computer programming, Volume 4, Fascicle 6*. Addison-Wesley series in computer science and information processing. Addison-Wesley, Reading, Mass., 1968.

Knuth, D. E. *The art of computer programming, Volume 3*. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co, Reading, Mass, 1973.

Kool, W., Hoof, H. v., and Welling, M. Attention, Learn to Solve Routing Problems! In *International Conference on Learning Representations, ICLR*, 2019.

Kreuzer, D., Beaini, D., Hamilton, W. L., Létourneau, V., and Tossou, P. Rethinking Graph Transformers with Spectral Attention. In *Neural Information Processing Systems, NeurIPS*, 2021.

Li, P., Wang, Y., Wang, H., and Leskovec, J. Distance Encoding: Design Provably More Powerful Neural Networks for Graph Representation Learning. In *Neural Information Processing Systems, NeurIPS*, 2020.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F.,

Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-Level Code Generation with AlphaCode, In *arXiv*, 2022.

Lim, D., Robinson, J., Zhao, L., Smidt, T., Sra, S., Maron, H., and Jegelka, S. Sign and Basis Invariant Networks for Spectral Graph Representation Learning, In *arXiv*, 2022.

Loshchilov, I. and Hutter, F. SGDR: Stochastic gradient descent with warm restarts. *International Conference on Learning Representations, ICLR*, 2017.

Loshchilov, I. and Hutter, F. Decoupled Weight Decay Regularization. *International Conference on Learning Representations, ICLR*, 2019.

Luo, Y. DAGformer: Directed Acyclic Graph Transformer, In *arXiv*, 2022.

Marques, A. G., Segarra, S., and Mateos, G. Signal Processing on Directed Graphs: The Role of Edge Directionality When Processing and Learning From Network Data. *IEEE Signal Processing Magazine*, 2020.

Mialon, G., Chen, D., Selosse, M., and Mairal, J. GraphiT: Encoding Graph Structure in Transformers, In *arXiv*, 2021.

Min, E., Chen, R., Bian, Y., Xu, T., Zhao, K., Huang, W., Zhao, P., Huang, J., Ananiadou, S., and Rong, Y. Transformer for Graphs: An Overview from Architecture Perspective, In *arXiv*, 2022.

Müller, L., Galkin, M., Morris, C., and Rampášek, L. Attending to Graph Transformers, In *arXiv*, 2023.

OpenAI. ChatGPT: Optimizing Language Models for Dialogue. URL https://openai.com/blog/chatgpt/, 2022.

Page, L., Brin, S., Motwani, R., and Winograd, T. The PageRank Citation Ranking : Bringing Order to the Web. In *The Web Conference*, 1999.

Rampášek, L., Galkin, M., Dwivedi, V. P., Luu, A. T., Wolf, G., and Beaini, D. Recipe for a General, Powerful, Scalable Graph Transformer. In *Neural Information Processing Systems, NeurIPS*, 2022.

Rossi, E., Charpentier, B., Di Giovanni, F., Frasca, F., Günnemann, S., and Bronstein, M. Edge Directionality Improves Learning on Heterophilic Graphs, In *arXiv*, 2023.

Sandryhaila, A. and Moura, J. M. F. Discrete Signal Processing on Graphs. *IEEE Transactions on Signal Processing*, 2013.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT Solver from Single-Bit Supervision. In *International Conference on Learning Representations, ICLR*, 2019.

Sevi, H., Rilling, G., and Borgnat, P. Harmonic analysis on directed graphs and applications: from Fourier analysis to wavelets, In *arXiv*, 2021.

Singh, R., Chakraborty, A., and Manoj, B. S. Graph Fourier Transform based on Directed Laplacian, In *arXiv*, 2016.

Smith, S. W. *The scientist and engineer's guide to digital signal processing*. California Technical Pub., San Diego (Calif.), 2nd edition edition, 1999. OCLC: 493473234.

Stevanović, D. Research problems from the Aveiro Workshop on Graph Spectra. *Linear Algebra and its Applications*, 2007.

Strang, G. The Discrete Cosine Transform. *SIAM Review*, 1999.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, , and Polosukhin, I. Attention is all you need. *Neural Information Processing Systems, NeurIPS*, 2017.

von Luxburg, U. A tutorial on spectral clustering. *Statistics and Computing*, 2007.

Wang, H., Yin, H., Zhang, M., and Li, P. Equivariant and Stable Positional Encoding for More Powerful Graph Neural Networks. In *International Conference on Learning Representations, ICLR*, 2022.

Wu, Q., Zhao, W., Li, Z., Wipf, D., and Yan, J. NodeFormer: A Scalable Graph Structure Learning Transformer for Node Classification. In *Neural Information Processing Systems, NeurIPS*, 2022.

Yefet, N., Alon, U., and Yahav, E. Adversarial Examples for Models of Code. *ACM Program. Lang.*, 2020.

Ying, C., Cai, T., Luo, S., Zheng, S., Ke, G., He, D., Shen, Y., and Liu, T.-Y. Do Transformers Really Perform Bad for Graph Representation? *Neural Information Processing Systems, NeurIPS*, 2021.

Zhang, X., He, Y., Brugnone, N., Perlmutter, M., and Hirn, M. MagNet: A Neural Network for Directed Graphs. In *Neural Information Processing Systems, NeruIPS*, 2021.

A. Example Graphs

Additionally to the graphs in Fig. 1, here we give further examples (illustrated in Fig. A.1). These examples, are used for the more elaborate discussion in the appendix. The construction of most graphs should be self-explanatory, even if we increase the number of nodes. We next provide necessary details. For the two disconnected sequences Fig. A.1i, we split the sequence after the first $\lfloor n/2 \rfloor$ tokens/nodes. All "trumpet graphs" (Fig. A.1d, A.1h, and A.1l) connect the nodes (between) $\lfloor 3n/10 \rfloor$ and $\lfloor 7n/10 \rfloor$. For Fig. A.1k, we first construct a fully connected DAG plus self-loops (entries of main diagonal and above are all one). Then, we add the reversed edges for the inner 50% of nodes ($\lfloor n/4 \rfloor$ to $\lfloor 3n/4 \rfloor$).

(a) Sequence	(b) Undirected sequence	(c) Balanced binary tree	(d) Trumpet
(e) Reversed sequence	(f) Cirlce	(g) Reversed bal. bin. tree	(h) Trumpet (DAG)
(i) Disconnected seq.	(j) Fully connected DAG	(k) Mix DAG & fully con.	(l) Trumpet (fully con.)
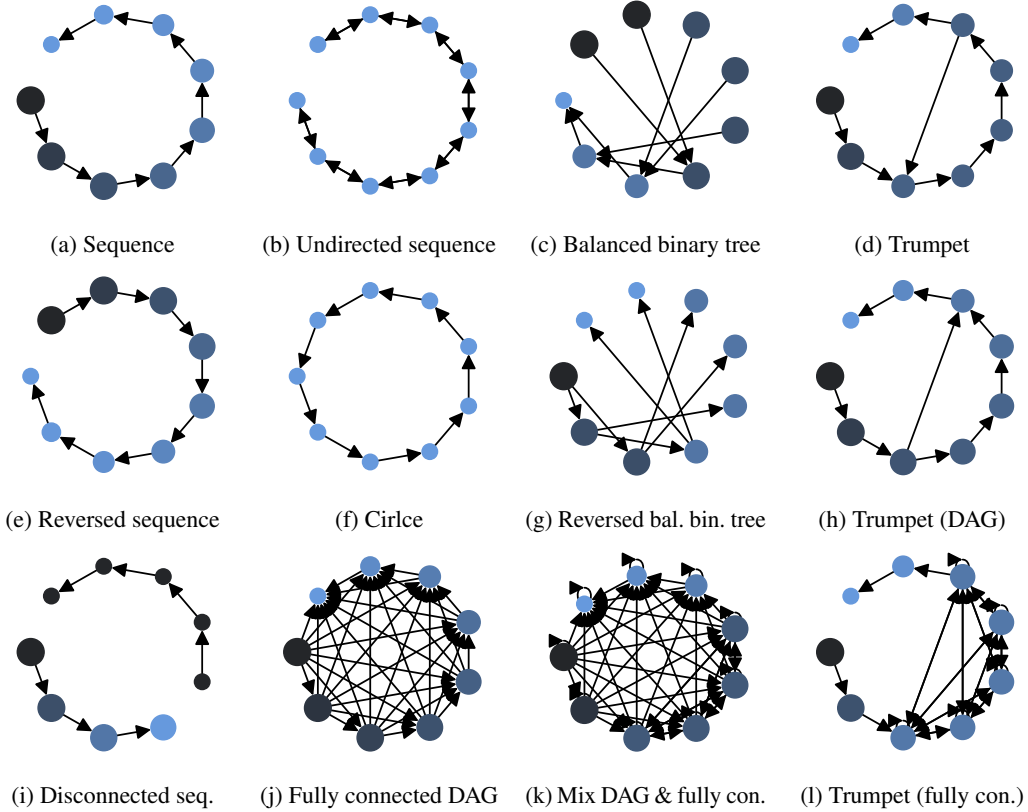
Figure A.1: First eigenvector of Magnetic Laplacian. Node size encodes the real value and color the imaginary value.

B. Graph Fourier Transformation

Similarly to the Discrete Fourier Transformation (DFT) Eq. 1, a Graph Fourier Transformation (GFT) for *undirected* graphs can be defined with the eigenvectors $\mathbf{\Gamma}$ of the combinatorial Laplacian (Eq. 2 or Eq. 3). Here the graph signal $\boldsymbol{x} \in \mathbb{R}^n$ is mapped to the frequency domain as

$$\boldsymbol{X} := \mathbf{\Gamma}^\top \boldsymbol{x} \tag{B.1}$$

or for a specific frequency/eigenvalue

$$X(\boldsymbol{\gamma}_k) = X_k = \sum_{i=1}^{n} x_i \Gamma_{i,k} \tag{B.2}$$

The inverse transform is then given as $\boldsymbol{x} = \mathbf{\Gamma} \boldsymbol{X}$. Here we assume $\mathbf{\Gamma}$ to be orthonormal ($\mathbf{\Gamma}\mathbf{\Gamma}^\top = \boldsymbol{I}$). In analogy to sequences, the lowest eigenvalues and eigenvectors reflect the low frequencies. For undirected and connected graphs, the first (also called trivial) eigenvector is constant across nodes: $\Gamma_{u,0} = \pm 1/\sqrt{n}$. This primitive is apparent from the quadratic form of the Laplacian

$$\boldsymbol{x}^\top \boldsymbol{L}_U \boldsymbol{x} = \frac{1}{2} \sum_{(u,v) \in E} (x_u - x_v)^2 = \frac{1}{2} \sum_{u=1}^{n} \sum_{v=1}^{n} A_{u,v}(x_u - x_v)^2 \tag{B.3}$$

13

that is minimized by its smallest eigenvector $\gamma_0^\top \boldsymbol{L}_U \gamma_0 = \lambda_0 = 0$. We will come back to a similar relation for the smallest eigenvector of the Magnetic Laplacian in § D.

Graph convolution. Particularly important for graph signal processing is the convolution defined on graphs

$$g_\theta \circledast \boldsymbol{x} = \boldsymbol{\Gamma} g_\theta(\boldsymbol{\Lambda}) \underbrace{\boldsymbol{\Gamma}^\top \boldsymbol{x}}_{\text{GFT}} = \boldsymbol{\Gamma} \underbrace{g_\theta(\boldsymbol{\Lambda}) \boldsymbol{X}}_{\text{filter} \times \text{signal}} = \underbrace{\boldsymbol{\Gamma} \hat{\boldsymbol{X}}}_{\text{inverse GFT}} \tag{B.4}$$

where $g_\theta(\boldsymbol{\Lambda})$ is a diagonal matrix parameterized by θ that can be understood as a function of the eigenvalues $\boldsymbol{\Lambda}$ and represents the filter in the frequency domain. For more details see Defferrard et al. (2017) or Furutani et al. (2020).

Applications in machine learning. The eigenvectors of the combinatorial Laplacian are widely used in graph machine learning. For example, they are the workhorse in spectral clustering (von Luxburg, 2007) where one can gain helpful insights if dealing with disconnected graphs. Moreover, the Laplacian eigendecomposition also stands at the core of many Graph Neural Networks. For example, ChebyNet (Defferrard et al., 2017) approximates $g_\theta(\boldsymbol{\Lambda})$ (Eq. B.4) via k-order polynomial in the spatial domain and, similarly, the popular Graph Convolutional Network (Kipf & Welling, 2017) uses an alike linear approximation. For more background, we refer to Bronstein et al. (2021).

C. Laplacians for Directed Graphs

It is well known that the combinatorial Laplacian is a discretization of the Laplace-Beltrami operator on Riemannian manifolds (Belkin & Niyogi, 2003). This insight allows for many connections, including the negligence of direction. In short, here we only encode distances on a manifold where the order of start and end point is irrelevant. In other words, the eigenvectors of the combinatorial Laplacian form an *isotropic* transformation.

Combinatorial Laplacian w/o symmetrization. For directed graphs the Laplacian, e.g., $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$ is in general not symmetric. Hence, the eigenvectors as well as eigenvalues are potentially complex, $\boldsymbol{L}$ is not necessarily diagonalizable, and the resulting eigenvectors might not form an orthonormal basis. Especially, the latter criterion is important s.t. the different signals do not interfere with each other and that we can go back and forth from the spatial to the frequency domain without complications. Specifically, the eigenvectors might neither be *unitary* $\bar{\boldsymbol{\Gamma}}^\top \boldsymbol{\Gamma} = \boldsymbol{I}$ or there might not even be a basis of eigenvectors.

We plot the first 5 eigenvectors of the combinatorial Laplacian without symmetrization in the right column of Fig. D.1 and D.2 for all the graphs in Fig. A.1. It is clear that these eigenvectors are well-suited for positional encodings of nodes for many of the graph topologies. For example, if we would use $\bar{\boldsymbol{\Gamma}}^\top$ or $\boldsymbol{\Gamma}$ in the GFT or its inverse, respectively, we would potentially ignore the signal of some nodes. Equivalently, a positional encoding would assign the the zero vector $\boldsymbol{0}$ to these nodes.

Magnetic Laplacian. Various alternatives were proposed that generalize (or substitute) the eigenvectors for the Laplacian to directed graphs. So far, we have only discussed the Magnetic Laplacian (§ 3 and D) that bridges the gap using complex values, i.e., using a Hermitian matrix for which eigenvectors again form an orthogonal basis. We can use these eigenvectors for a GFT that includes directed graphs (Furutani et al., 2020). Such a transformation, where the direction does matter, is called *anisotropic*.

Jordan Decomposition. Other approaches include the use of generalized eigendecomposition, like the Jordan decomposition (Sandryhaila & Moura, 2013). However, here we are still left with a potentially non-orthogonal set of eigenvectors, and, most importantly, in this case, the "low frequencies" do not necessarily change smoothly over the nodes in the graph (Singh et al., 2016).

Eigenfunctions of random walk operator. Sevi et al. (2021), on the other hand, use the Dirichlet energy of eigenfunctions of a random walk operator. However, this approach is only applicable to strongly connected graphs and comes with restrictions related to the orthogonality of the Fourier basis.

Optimization. Last, (non-convex) optimization problems with constraints were proposed that typically minimize the Lovász extension of the graph cut size or local variations of the graph signal. Both directions additionally impose orthogonality constraints. A good entry point for the literature can be found in (Marques et al., 2020). It is worth noting though, that these approaches typically do not preserve all information s.t. we can reconstruct the graph structure (Furutani et al., 2020). Hence, it is not clear if they are well-suited positional encodings.

D. Magnetic Laplacian

We next give more details on the Magnetic Laplacian and its eigenvectors. For this recall its definition:

$$\boldsymbol{L}_U^{(q)} := \boldsymbol{D}_s - \boldsymbol{A}_s \odot \exp\left(i\boldsymbol{\Theta}^{(q)}\right) \tag{D.1}$$

with Hadamard product $\odot$, element-wise $\exp$, $i = \sqrt{-1}$, $\Theta_{u,v}^{(q)} := 2\pi q(A_{u,v} - A_{v,u})$, and potential $q \geq 0$. The degree-normalized counterpart is defined as

$$\boldsymbol{L}_N^{(q)} := \boldsymbol{I} - \left(\boldsymbol{D}_s^{-1/2}\boldsymbol{A}_s\boldsymbol{D}_s^{-1/2}\right) \odot \exp\left(i\boldsymbol{\Theta}^{(q)}\right) \tag{D.2}$$

Here the quadratic form (see Eq. B.3) becomes

$$
\begin{aligned}
&\frac{1}{2}\sum_{(u,v)\in E_s} |x_u - x_v \exp(i\Theta_{u,v}^{(q)})|^2 \\
&= \frac{1}{2}\sum_{(u,v)\in E_s} \overline{(x_u - x_v \exp(i\Theta_{u,v}^{(q)}))}(x_u - x_v \exp(i\Theta_{u,v}^{(q)})) \\
&= \frac{1}{2}\sum_{(u,v)\in E_s} \bar{x}_u x_u - \exp(i\Theta_{u,v}^{(q)})\bar{x}_u x_v - \exp(i\Theta_{v,u}^{(q)})\bar{x}_v x_u + \underbrace{\exp(i\Theta_{u,v}^{(q)})\exp(i\Theta_{v,u}^{(q)})}_{=1} \bar{x}_v x_v \\
&= \frac{1}{2}\sum_{(u,v)\in E_s} \bar{x}_u x_u - \exp(i\Theta_{u,v}^{(q)})\bar{x}_u x_v - \exp(i\Theta_{v,u}^{(q)})\bar{x}_v x_u + \bar{x}_v x_v \\
&= \sum_{(u,v)\in E_s} \bar{x}_u x_u - \exp(i\Theta_{u,v}^{(q)})\bar{x}_u x_v \\
&= \underbrace{\sum_{(u,v)\in E_s} \bar{x}_u x_u}_{=\bar{\boldsymbol{x}}^\top \boldsymbol{D}_s \boldsymbol{x}} - \underbrace{\sum_{(u,v)\in E_s} \exp(i\Theta_{u,v}^{(q)})\bar{x}_u x_v}_{=\bar{\boldsymbol{x}}^\top (\boldsymbol{A}_s \odot \exp(i\boldsymbol{\Theta}^{(q)}))\boldsymbol{x}} \\
&= \bar{\boldsymbol{x}}^\top \left(\boldsymbol{D}_s - \boldsymbol{A}_s \odot \exp(i\boldsymbol{\Theta}^{(q)})\right)\boldsymbol{x} \\
&= \bar{\boldsymbol{x}}^\top \boldsymbol{L}_U^{(q)}\boldsymbol{x}
\end{aligned}
\tag{D.3}
$$

where E_s is the set of edges of the symmetrized graph. Note that either $\Theta_{v,u}^{(q)} = -\Theta_{u,v}^{(q)}$ or $\Theta_{v,u}^{(q)} = 0$.

Recall that the first eigenvector minimizes the Rayleigh quotient

$$\min_{\boldsymbol{x}\in\mathbb{C}^n} \frac{\bar{\boldsymbol{x}}^\top \boldsymbol{L}_U^{(q)}\boldsymbol{x}}{\bar{\boldsymbol{x}}^\top \boldsymbol{x}} = \frac{\bar{\boldsymbol{\gamma}}_0^\top \boldsymbol{L}_U^{(q)}\boldsymbol{\gamma}_0}{\bar{\boldsymbol{\gamma}}_0^\top \boldsymbol{\gamma}_0} = \lambda_0 \tag{D.4}$$

From this, we can obtain the behavior of the first eigenvector of the Magnetic Laplacian (as illustrated in Fig. 3). The first eigenvector $\boldsymbol{\gamma}_0$ is related to the so-called "angular synchronization problem" (Eq. D.5). In angular synchronization, we seek the L^2-optimal estimate of n angles $\boldsymbol{\alpha}$ given m (noisy) measurements of phase offsets $\alpha_u - \alpha_v \mod 2\pi$ where $u, v \in \{0, 1, \ldots, n-1\}$. Formally, the angular synchronization problem (Bandeira et al., 2013) is defined as (we drop the normalizing constants as they do not influence the minimum)

$$\angle(\boldsymbol{\gamma}_0) \in \arg\min_{\boldsymbol{\alpha}\in[0,2\pi)^n} \eta(\boldsymbol{\alpha}) \quad \text{with } \eta(\boldsymbol{\alpha}) = \sum_{u,v\in E} |\exp(i\alpha_v) - \exp(i\alpha_u + i\Theta_{u,v})|^2 \tag{D.5}$$

(a) Sequence

(b) Eigenvectors of Magnetic Laplacian

(c) Eigenvec. of comb. Lap. w/o symmetrization

(d) Reversed sequence

(e) Eigenvectors of Magnetic Laplacian

(f) Eigenvec. of comb. Lap. w/o symmetrization

(g) Undirected sequence

(h) Eigenvectors of Magnetic Laplacian

(i) Eigenvec. of comb. Lap. w/o symmetrization

(j) Circle

(k) Eigenvectors of Magnetic Laplacian

(l) Eigenvec. of comb. Lap. w/o symmetrization

(m) Disconnected seq. sequence

(n) Eigenvectors of Magnetic Laplacian

(o) Eigenvec. of comb. Lap. w/o symmetrization

(p) Binary tree

(q) Eigenvectors of Magnetic Laplacian

(r) Eigenvec. of comb. Lap. w/o symmetrization

Figure D.1: First eigenvector(s) for sample graphs (part 1). In the left column (a, d, g, j, m, p), we show the first eigenvector of the Magnetic Laplacian for $q = 0.25$. The node size encodes the real value and colors the imaginary value. In the middle column (b, e, h, k, n, q), we show the first 5 eigenvectors on a graph with $n = 100$ nodes. In the right column (c, f, i, l, o, r), we show instead the eigenvectors of the Laplacian (Eq. 2) omitting the symmetrization.

16

(a) Reversed bin. tree

(b) Eigenvectors of Magnetic Laplacian

(c) Eigenvec. of comb. Lap. w/o symmetrization

(d) Trumpet

(e) Eigenvectors of Magnetic Laplacian

(f) Eigenvec. of comb. Lap. w/o symmetrization

(g) Trumpet (forward)

(h) Eigenvectors of Magnetic Laplacian

(i) Eigenvec. of comb. Lap. w/o symmetrization

(j) Trumpet (DAG)

(k) Eigenvectors of Magnetic Laplacian

(l) Eigenvec. of comb. Lap. w/o symmetrization

(m) Fully con. DAG

(n) Eigenvectors of Magnetic Laplacian

(o) Eigenvec. of comb. Lap. w/o symmetrization

(p) Mix DAG & f. con.

(q) Eigenvectors of Magnetic Laplacian
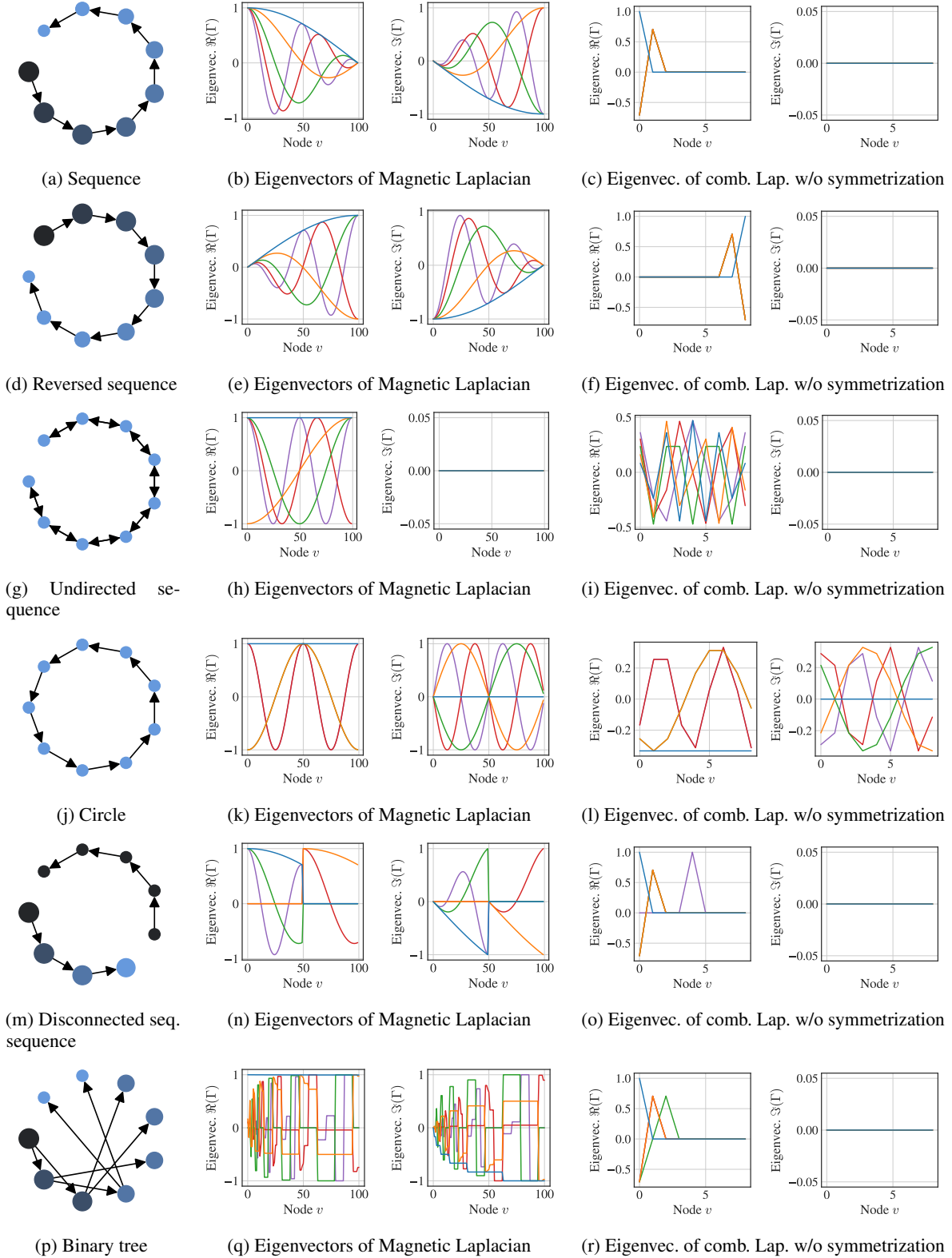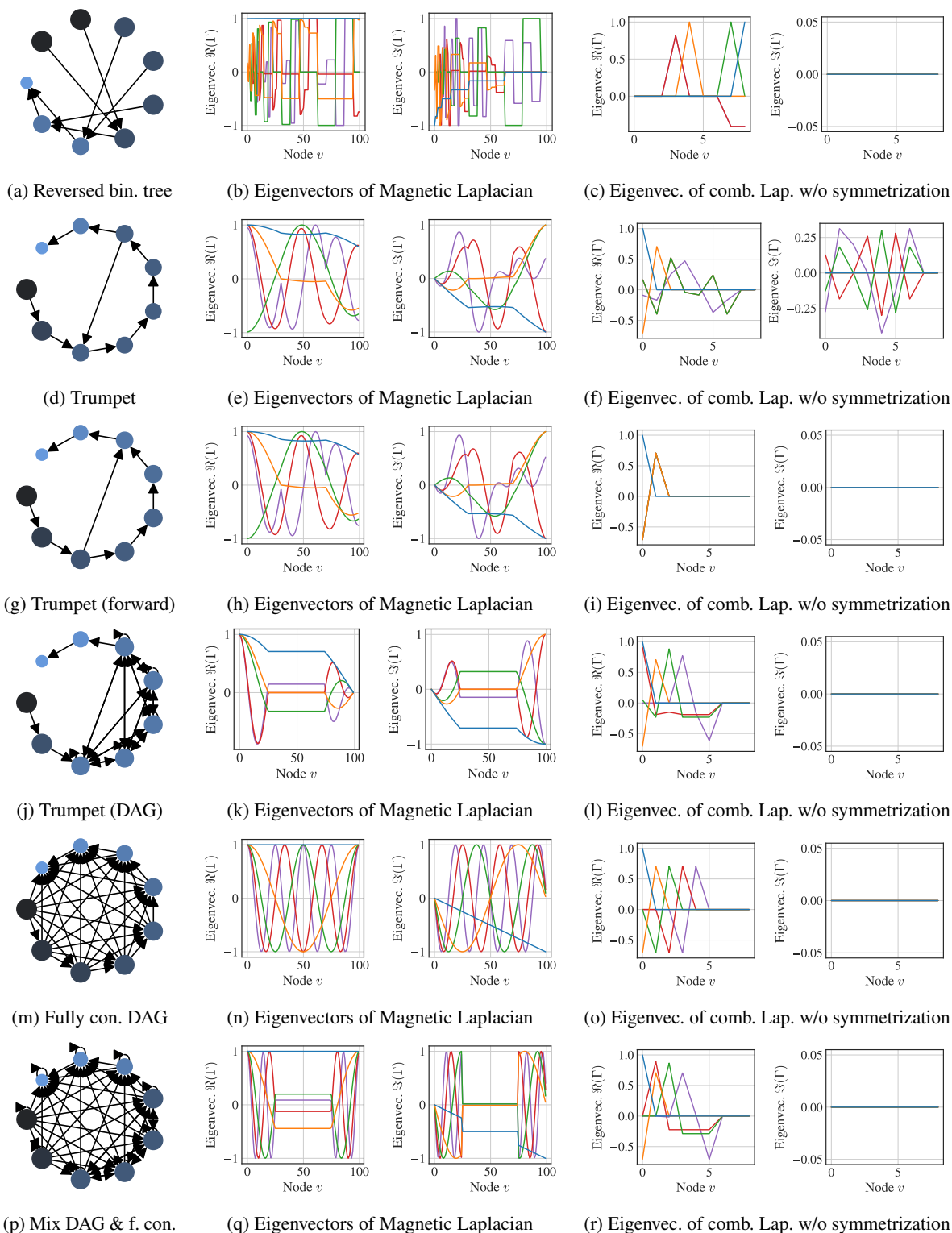
(r) Eigenvec. of comb. Lap. w/o symmetrization

Figure D.2: First eigenvector(s) for sample graphs (part 1). In the left column (a, d, g, j, m, p), we show the first eigenvector of the Magnetic Laplacian for $q = 0.25$. The node size encodes the real value and colors the imaginary value. In the middle column (b, e, h, k, n, q), we show the first 5 eigenvectors on a graph with $n = 100$ nodes. In the right column (c, f, i, l, o, r), we show instead the eigenvectors of the Laplacian (Eq. 2) omitting the symmetrization.

In this analogy, each *directed* edge $A_{u,v} \neq A_{v,u}$ encourages a difference in the angle / phase in the first eigenvector $\Gamma_{u,0}$ and $\Gamma_{v,0}$, while an *undirected* edge $A_{u,v} = A_{v,u} = 1$ supports them to be equal. We give an example in Fig. 3. Here each directed edge induces a phase shift in $\boldsymbol{\gamma}_0$ of $2\pi q h \mod 2\pi$ and the undirected edges connect to nodes of equal phase.

In contrast to the combinatorial Laplacian, the first eigenvalue can only be equal to zero if the angles of the first eigenvector $\boldsymbol{\gamma}_0$ (also see Eq. D.5) are "conflict-free", i.e., $|\Gamma_{u,0} - \Gamma_{v,0} \exp(\Theta_{u,v}^{(q)})|^2 = 0$ for all $(u,v) \in \mathbb{E}$ (this term also appears in § D.1). We plot the first eigenvector(s) of the Magnetic Laplacian in the first two columns of Fig. D.1 and D.2. For an eased comparability, here we normalize the eigenvectors s.t. the maximum absolute value is equal to one ($\max_u |\boldsymbol{\gamma}_u| = 1$).

Relationship between combinatorial and Magnetic Laplacian. For certain graphs we can relate the eigenvectors of the Magnetic Laplacian to the eigenvectors of the combinatorial Laplacian: $\Gamma_{v,j}^{(q)} = c s_v \Gamma_{v,j}^{(0)}$ for node v, the j-th eigenvector, normalizer $c \in \mathbb{C} \setminus \{0\}$, and vector $\boldsymbol{s} \in \mathbb{C}^n$. Moreover, we define $\boldsymbol{S}$ as the diagonal matrix with $\boldsymbol{s}$ on its diagonal. Then, if we choose $\boldsymbol{S}$ s.t. $\boldsymbol{L}^{(q)}\boldsymbol{S} = \boldsymbol{S}\boldsymbol{L}^{(0)}$ it follows that $\boldsymbol{L}^{(q)}\boldsymbol{S}\boldsymbol{\gamma}_j^{(0)} = \boldsymbol{S}\boldsymbol{L}^{(0)}\boldsymbol{\gamma}_j^{(0)} = \boldsymbol{S}(\boldsymbol{L}^{(0)}\boldsymbol{\gamma}_j^{(0)})$. Since, $\boldsymbol{L}^{(0)}\boldsymbol{\gamma}_j^{(0)} = \lambda_j^{(0)}\boldsymbol{\gamma}_j^{(0)}$ we conclude $\boldsymbol{L}^{(q)}\boldsymbol{S}\boldsymbol{\gamma}_j^{(0)} = \boldsymbol{S}(\lambda_j^{(0)}\boldsymbol{\gamma}_j^{(0)}) = \lambda_j^{(0)}\boldsymbol{S}\boldsymbol{\gamma}_j^{(0)}$. Thus, the eigenvectors of the Magnetic Laplacian can be calculated form the eigenvectors of the combinatorial Laplacian $\boldsymbol{\gamma}_j^{(q)} = \boldsymbol{S}\boldsymbol{\gamma}_j^{(0)}$ if $\boldsymbol{S}$ exists and is known. For example, for trees and sequences it is trivial to construct $\boldsymbol{S}$. Here the elements on the diagonal can be chosen to $S_{v,v} = \exp(-i2\pi d_v)$, where d_v is the distance from the root node to node v.

Repeated eigenvalues. A source of ambiguity in the eigenvectors $\boldsymbol{\Gamma}$ emerges in the case of l repeated eigenvalues (also called multiple eigenvalues) of a connected component. Then, the respective eigenvectors can be chosen from an l-dimensional subspace as long as they are orthogonal. We refer to (Lim et al., 2022) and Wang et al. (2022) on how to address it.

Permutation equivariance. Notably, in the presence of repeated eigenvalues, the eigenvectors calculated by the eigensolvers are generally not equivariant to node-permutations. Following, also our eigenvector encodings are not permutation equivariant for graphs with repeated eigenvalues. Similarly, permutation equivariance is affected by the arbitrary factor $c \in \mathbb{C} \setminus \{0\}$ we can apply to eigenvectors $c\boldsymbol{L}\boldsymbol{\gamma} = c\lambda\boldsymbol{\gamma} \implies \boldsymbol{L}(c\boldsymbol{\gamma}) = \lambda(c\boldsymbol{\gamma})$. Thus, on needs also to be careful about modelling the scale, sign and rotation of c to retain permutation equivariance. If using a sign-net like encoder with our convention of normalizing the eigenvectors, we obtain equivariance w.r.t. scale, sign and rotation.

Disconnected components. In the case of disconnected components, the eigenvectors and eigenvalues resolve as if we would decompose each component independently, where the components of the eigenvectors for the other components are set to zero. For example, for two disconnected sequences Fig. A.1i with an even number of nodes, we obtain two equal disconnected components. Here, we will obtain each eigenvalue λ twice. Also, $\boldsymbol{\Gamma}$ contains two full sets of equivalent eigenvectors, except that they are zero for the other component and vice versa.

Co-spectrality (Stevanović, 2007) describes the phenomenon that there exist (potentially non-isomorphic) graphs with identical eigenvalues. Co-spectrality is of lesser concern for our architecture since we also process the eigenvectors. Similarly to co-spectrality, many well-known facts for the combinatorial Laplacian (e.g., the Cheeger inequality) extend to the Magnetic Laplacian. For the interested reader, we refer to (Bandeira et al., 2013).

D.1. Influence of Potential q

The potential q seems to be a crucial choice for successfully encoding direction. For example, in Fig. 9, we see that for too large potentials q, the performance degrades on the sorting network tasks. Moreover, Furutani et al. (2020) show that for too large values of q, the eigenvectors of the Magnetic Laplacian do not order from low to high frequencies. In other words, then the eigenvalue order is not predictive for the variation of the graph signal.

In applications of the Magnetic Laplacian stemming from physics, q is typically given since, e.g., it represents the electric charge in a magnetic field. However, in general, it is not clear how to derive appropriate q values from the domain. Although the Magnetic Laplacian has been used for a Spectral GNN (Zhang et al., 2021), Community Detection (Fanuel et al., 2016) or the visualization of directed graphs (Fanuel et al., 2018), there is not much guidance on how to choose q in these works. These works treat q simply as a (hyper-) parameter.

Fanuel et al. (2018) open a perspective on q that is perhaps not well-suited for positional encodings but conveys an interesting intuition. They argue to choose q as a rational number. For example, $q = 1/3$ is particularly well-suited for visualizing graphs that consist of directed triangles. In such a directed triangle, each edge can induce a shift of $2/3\pi$. Consequently, a triangle would induce a cumulative shift of a full 360 degrees.
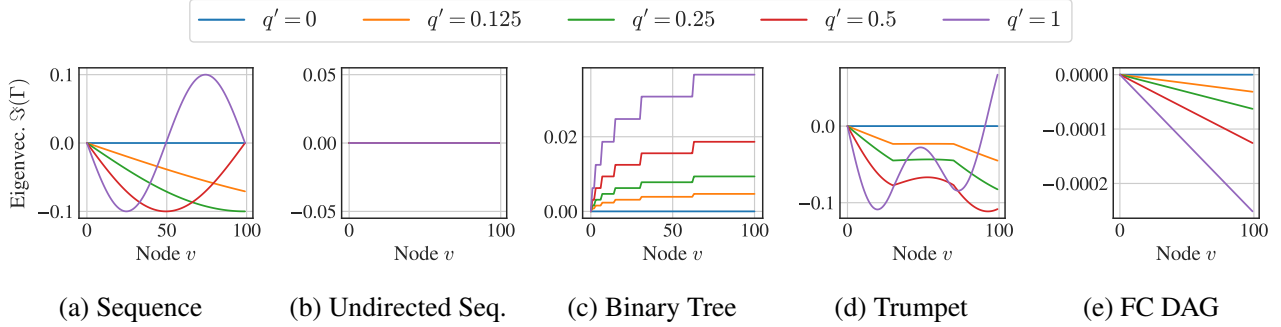
Figure D.3: The influence of q on the imaginary part of the eigenvectors for the graphs in Fig. 1.

In graph signal processing, Furutani et al. (2020) propose to choose potential q using insights from eigenvector perturbation theory. However, they choose potential q based on the average node degree which is not related to the directedness of the graph. Additionally, it does not scale with n and, hence, the maximum phase shift between a source and target node is not bounded for larger n.

Conflict-free graphs. We argue that for positional encodings it is particularly helpful to bound the total phase shift (here for Eq. 4) to avoid degenerate cases. For this we first discuss graphs we call *conflict-free*, i.e., if its first eigenvalue is $\bar{\gamma}_0^\top \boldsymbol{L}^{(q)} \gamma_0 = \lambda_0 = 0$ for all $0 < q \le 1/4$. It is easy to see that for graphs without conflicting edges the phase shift between at least one source and sink nodes is $2\pi q l$, where l is the maximum number of *purely directed edges* ($\{(u,v) \in E \mid (v,u) \notin E\}$) in a path accounting for their direction (Eq. D.4). That is, we increment f l for every *purely directed edge* that we traverse in its direction and decrement l if we traverse such an edge against its direction. Bidirectional edges do not affect l. This can be understood as a weighted longest simple path problem. In the following, we call l simply the longest simple path.

(a) Absolute potential $q = 0$ (relative $q' = 0$)

(b) Absolute potential $q = 2.5 \times 10^{-3}$ (relative $q' = 2.5e - 1$)

(c) Absolute potential $q = 2.5 \times 10^{-2}$ (relative $q' = 2.5$)

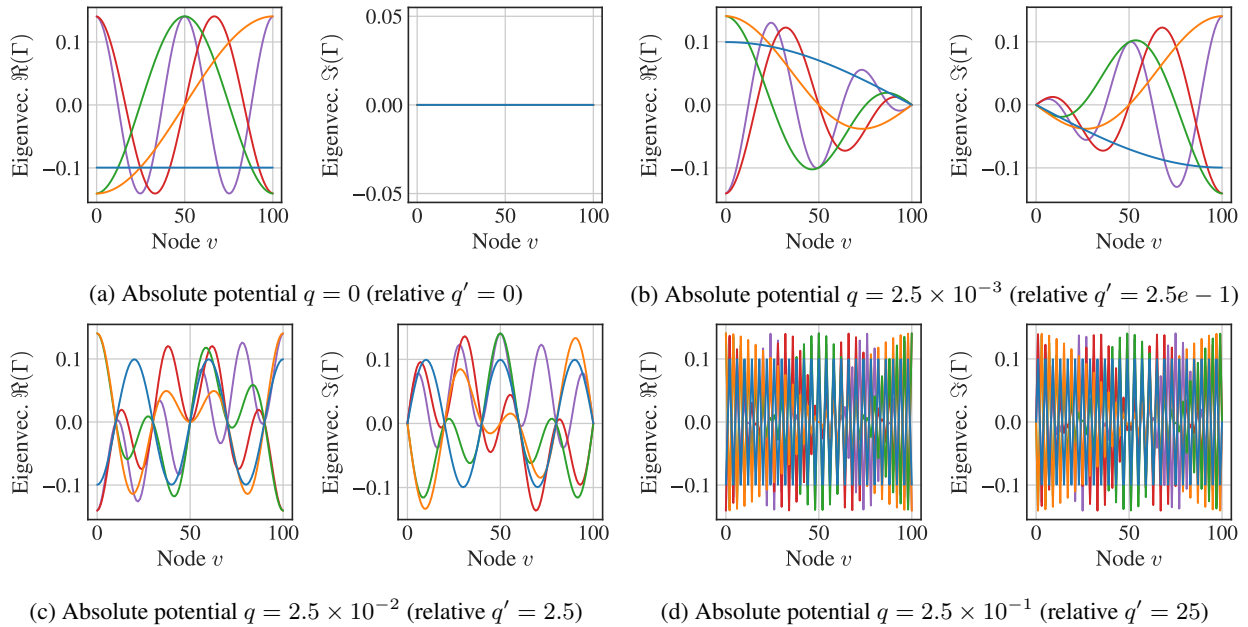(d) Absolute potential $q = 2.5 \times 10^{-1}$ (relative $q' = 25$)

Figure D.4: Eigenvectors of Magnetic Laplacian (Eq. D.1) for a sequence Fig. 1a with $n = 101$ nodes where we also include particularly large q values (see subcaptions).

All directed trees are conflict-free, but there also exist conflict-free graphs with cycles. For example, we can construct a conflict-free graph with cycles, if we add self-loops. Alternatively, a graph remains conflict-free, if we add bidirectional edges between (some) pairs of nodes u and v that have the same phase, i.e., if $\angle(\Gamma_{u,0}) = \angle(\Gamma_{v,0})$.

Moreover, except for numerical issues of the eigendecomposition, also for general graphs ($\lambda_0 > 0$) the phase shift is bounded by $2\pi q l$. The argument is as the following: Choose an arbitrary pair of nodes u and v for which the longest simple path distance is l (see above). Although there might exist other paths between u and v, they have at most a simple path length of l. If there exist alternative paths of smaller length $o < l$, then $\bar{\gamma}_0^\top \boldsymbol{L}^{(q)} \gamma_0$ is optimal for a maximum shift in the open interval $(2\pi q o, 2\pi q l)$ (here with accounting for overflows in the value range).

Visualizations. We next validate our choice for $q = q'/d_\mathcal{G}$ with *relative potential* q' (see § 3) and graph specific normalizer $d_\mathcal{G} = \max(\min(\vec{m}, n), 1)$. In Fig. D.3, we show the imaginary value of the first eigenvector for different exemplary graphs. We see that for the conflict-free sequence with longest path distance $\vec{m}$, $2\pi/q'$ is the total induced phase shift. For all shown graphs, with $q' \leq 1/4$ we could reorder the graph nodes using a simple sort operation (up to ties and cycles). We argue that this is a desirable property for directional positional encodings. We demonstrate this ability in § D.3. Moreover, in Fig. D.4, we contrast the eigenvectors of the combinatorial Laplacian (a) as well as Magnetic Laplacian with reasonable $q' = 1/4$ (b) to very high values for potential q (c-d). For the large potentials q, the resulting oscillations in the direction can be of high frequency (relatively to n). Consequently, it seems neither helpful nor necessary to choose high potential values q. Our graph-specific normalization avoids such degenerate cases.

D.2. Sign, Scale and Rotation

As discussed in § 3, if γ is an eigenvector of $\boldsymbol{L}$ then so is $c\gamma$, even if $c \in \mathbb{C}$ with $|c| > 0$ (proof: $c\boldsymbol{L}\gamma = c\lambda\gamma \implies \boldsymbol{L}(c\gamma) = \lambda(c\gamma)$). One way to cope with this is to fix c s.t. the neural network does not need to be invariant w.r.t. to arbitrary factors c. We note that this is much more challenging for complex eigenvectors as than it is for real-valued eigenvectors. We give an algorithmic description in Fig. D.1.

For the subsequent procedure it is important that the maximum relative phase shift is small enough. Moreover, the used eigensolver chooses a zero imaginary component for the first node (in terms of the adjacency matrix). With out choice of potential q, the rotation to the first node is within $[-\pi/2, \pi/2]$.

Algorithm D.1 Normalize Eigenvectors

1: **Input:** Eigenvectors $\boldsymbol{\Gamma} \in \mathbb{C}^{n \times k}$
2: $\boldsymbol{j} \leftarrow \operatorname{argmax}(|\Re(\boldsymbol{\Gamma})|, \text{axis} = 0)$ // shape k
3: $\boldsymbol{s} \leftarrow \operatorname{sign}(\Re(\boldsymbol{\Gamma})[0 : n - 1, \boldsymbol{j}])$ // shape k
4: $\boldsymbol{\Gamma} \leftarrow \boldsymbol{s} \odot \boldsymbol{\Gamma}$
5: $j \leftarrow \max(\Im(\boldsymbol{\Gamma}[:, 0]))$ // scalar
6: $\alpha \leftarrow \angle(\boldsymbol{\Gamma}[j, :])$
7: $\boldsymbol{\Gamma} \leftarrow \exp(-i\alpha)^\top \odot \boldsymbol{\Gamma}$
8: Return $\boldsymbol{\Gamma}$

Sign and scale. For the scale we choose c for all eigenvectors s.t. $\boldsymbol{\Gamma}$ is unitary. However, this still allows for choosing the sign / direction as well as rotation. For the sign, we choose maximum real component for each γ to be positive.

Rotation. If there is an application specific origin, e.g., as in function name prediction, we also use this to choose the rotation relatively to it (i.e., replace line 5 or abort if $j = 0$). Otherwise, as in the playground and sorting network tasks, we choose the foremost source node as origin. That is, we search first for the source node with greatest phase $\arg\max_v \angle(\Gamma_{v,0}) = u$. Then we use this node as origin. That is, we rotate *all* eigenvectors: $\angle(\Gamma_{u,j}) = 0, \ \forall j \in \{0, 1, \dots, n - 1\}$.

D.3. Reorder Permuted Graphs

We can also use the Magnetic Laplacian for reordering permuted graphs, up to ties. For example, we have ties in a balanced binary tree (Fig. A.1c or A.1g) stemming from the equal distance to the root node. For the three exemplary graphs in Fig. D.5, D.6, and D.7 we show that the eigenvectors of the Magnetic Laplacian can be used to reorder directed graphs. First, we permute the nodes arbitrarily, perform the eigendecomposition, and, lastly reorder the graphs after applying our normalization scheme for the eigenvectors (§ D.2). We naïvely recover the order by $\text{idx} = \operatorname{argsort}(\Im(\gamma_0))$.
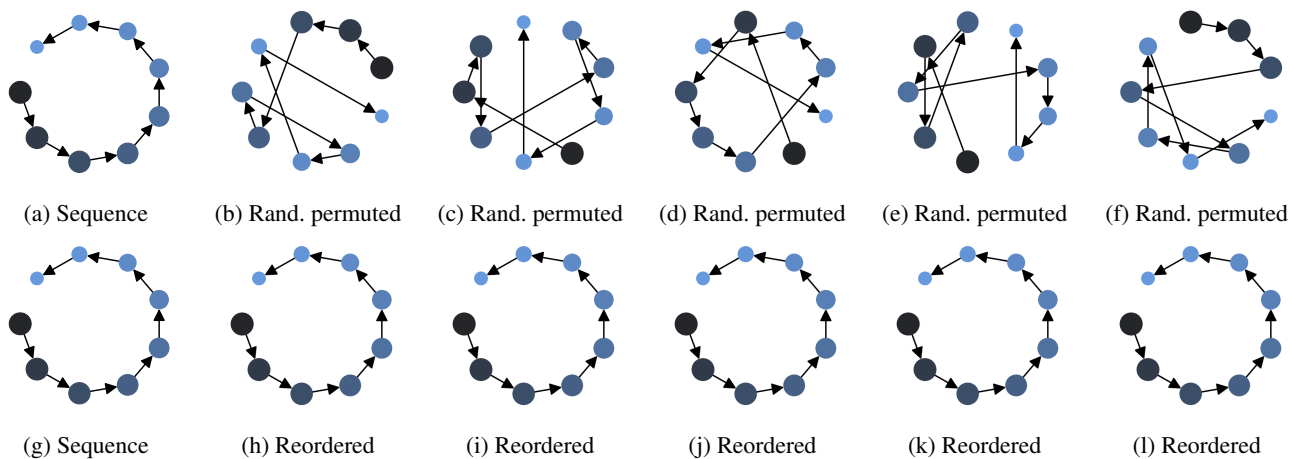
(a) Sequence   (b) Rand. permuted   (c) Rand. permuted   (d) Rand. permuted   (e) Rand. permuted   (f) Rand. permuted

(g) Sequence   (h) Reordered   (i) Reordered   (j) Reordered   (k) Reordered   (l) Reordered

Figure D.5: Reordering of randomly permuted sequence Fig. A.1a with $\text{idx} = \text{argsort}(\Im(\boldsymbol{\gamma}_0))$.

(a) Binary tree   (b) Rand. permuted   (c) Rand. permuted   (d) Rand. permuted   (e) Rand. permuted   (f) Rand. permuted

(g) Binary tree   (h) Reordered   (i) Reordered   (j) Reordered   (k) Reordered   (l) Reordered
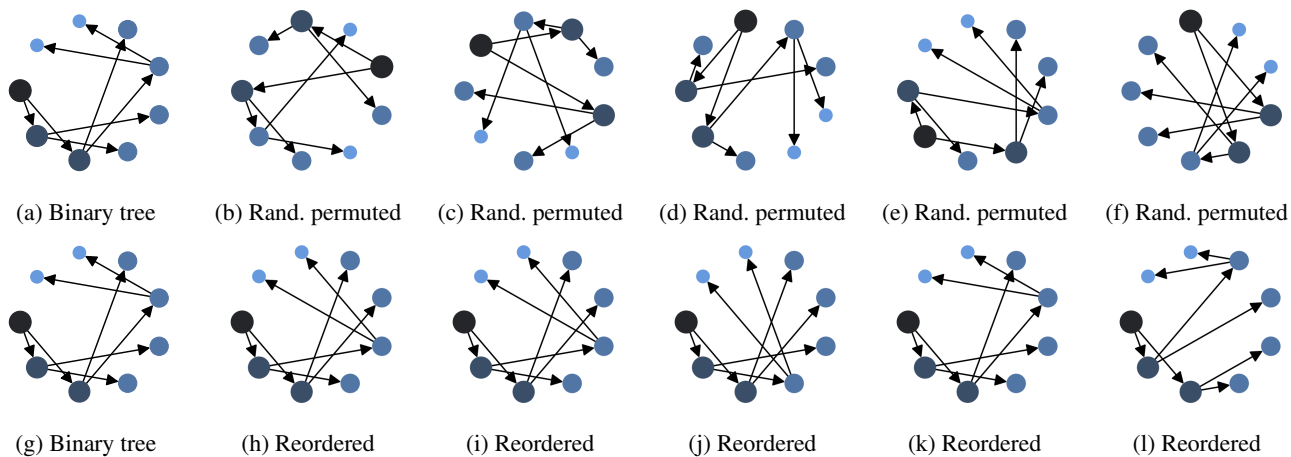
Figure D.6: Reordering of randomly permuted binary tree Fig. A.1c with $\text{idx} = \text{argsort}(\Im(\boldsymbol{\gamma}_0))$.

(a) Trumpet (fully c.)   (b) Rand. permuted   (c) Rand. permuted   (d) Rand. permuted   (e) Rand. permuted   (f) Rand. permuted

(g) Trumpet (ful. c.)   (h) Reordered   (i) Reordered   (j) Reordered   (k) Reordered   (l) Reordered
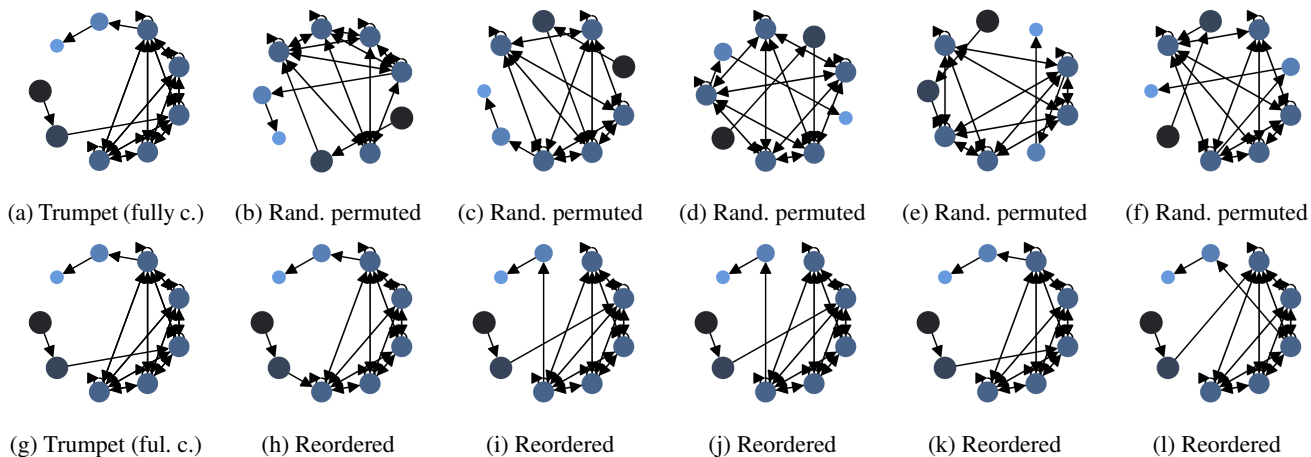
Figure D.7: Reordering of randomly permuted trumpet with fully connected middle part Fig. A.1l with $\text{idx} = \text{argsort}(\Im(\boldsymbol{\gamma}_0))$.

D.4. Comparison to Singular Value Decomposition

One could also use the Singular Value Decomposition (SVD) $U\Sigma V$ to obtain structure-aware positional encodings. Specifically, Hussain et al. (2022) argue that a low-rank approximation (via SVD) of the adjacency matrix yields general positional encodings that are also suitable for directed graphs (see § I for an empirical comparison). In contrast to the eigendecomposition, the singular values and singular vectors are real also if decomposing asymmetric matrices. However, it is questionable if the SVD of the adjacency matrix has desirable properties.

For example, a low-rank approximation of the adjacency matrix of a directed sequence Fig. A.1a simply filters out some of the edges. For example, a 2-rank approximation for a directed sequence with $n = 5$ nodes is

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \approx \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{D.6}$$

One could obtain appropriate results if using well-suited matrices like the combinatorial Laplacian for undirected graphs or the Magnetic Laplacian for directed graphs. However, then the alleged advantage of the SVD for directed graphs vanishes. That is, the singular vectors of a Hermitian Matrix are complex $\boldsymbol{U} \in \mathbb{C}^{n \times n}$ and $\boldsymbol{V} \in \mathbb{C}^{n \times n}$ and not real. Moreover, one needs to account then for the (slightly different) properties of the SVD, like its sign ambiguity $\boldsymbol{U}\Sigma\boldsymbol{V} = (\text{-}\boldsymbol{U})\Sigma(\text{-}\boldsymbol{V})$.

E. Weighted Graphs

Our positional encodings based on random walks as well as the Magnetic Laplacian straight-forwardly generalize to weighted graphs. Here the adjacency matrix $\boldsymbol{A} \in \mathbb{R}_{\geq 0}^{n \times n}$ contains positive real-valued weights. For the extension of the Magnetic Laplacian to signed graphs, i.e., also allowing negative edge weights, we refer to He et al. (2022).

For the random walk encodings, all equations in § 4 are directly applicable, however, for the (Magnetic) Laplacian one needs to adjust the strategy for symmetrizing the graph and potentially how to choose potential q. Then, for weighted graphs, the smallest eigenvalue of the Magnetic Laplacian solves

$$\min_{\boldsymbol{x} \in \mathbb{C}} \frac{\bar{\boldsymbol{x}}^{\top} \boldsymbol{L}^{(q)} \boldsymbol{x}}{\bar{\boldsymbol{x}}^{\top} \boldsymbol{x}} = \frac{1}{2} \sum_{(u,v) \in E_s} w_{u,v} |\Gamma_{u,0} - \Gamma_{v,0} \exp(\Theta_{u,v}^{(q)})|^2 \tag{E.1}$$

where $w_{u,v}$ is the weight for edge (u, v) in the then weighted symmetrized adjacency matrix $\boldsymbol{A}_s$.

F. Overview of Our Positional Encodings

We next provide a concise overview over the positional encodings. In Algorithm F.1, we provide an overview for the Magnetic Laplacian (§ 3). Thereafter, we list the full equation for the random walk encodings (§ 4). In both cases, we add the obtained positional encodings to the node features. For the relative random walk encodings see § I.

Algorithm F.1 Magnetic Laplacian Positional Encodings

1: **Input:** Adjacency matrix $\boldsymbol{A} \in \{0,1\}^{n \times n}$, potential q, number of eigenvectors k
2: Calculate $\boldsymbol{L}^{(q)} \leftarrow \text{Laplacian}(\boldsymbol{A}, q)$ // e.g. Eq. 4
3: Decompose $\boldsymbol{\Lambda}, \boldsymbol{\Gamma} \leftarrow \text{eigh}(\boldsymbol{L}^{(q)}, k)$
4: Normalize $\boldsymbol{\Gamma}, \boldsymbol{\Lambda} \leftarrow \text{norm}(\boldsymbol{\Gamma}, \boldsymbol{\Lambda})$ according to Algorithm D.1
5: Obtain preprocessed $\hat{\boldsymbol{\Gamma}} \leftarrow \text{MagLapNet}(\boldsymbol{\Gamma}, \boldsymbol{\Lambda})$ // see Fig. 4b
6: Return $\hat{\boldsymbol{\Gamma}}$

The finite-horizon random walk and Personalized Page Rank landing probabilities are processed together as

$$\zeta(v|u) = f_{\text{rw}}^{(2)}[\Pi(\boldsymbol{R})_{v,u}, (\boldsymbol{R}^k)_{v,u}, \ldots, (\boldsymbol{R}^2)_{v,u}, R_{v,u}, T_{v,u}, (\boldsymbol{T}^2)_{v,u}, \ldots, (\boldsymbol{T}^k)_{v,u}, \Pi(\boldsymbol{T})_{v,u}] \tag{F.1}$$

where $\Pi(\boldsymbol{T}) = p_r(\boldsymbol{I} - (1 - p_r)\boldsymbol{T})^{-1}$. Then, the node encodings (here for node v) are given as

$$\zeta(v|\mathcal{G}) = f_{\text{rw}}^{(1)}(\text{AGG}(\{\zeta(v|u) \mid u \in V\})) \tag{F.2}$$

Table G.1: Most important hyperparameters for tasks and models

	Model	Hyperparameter
Playground and Sorting Network	Base	15 epochs sorting network / 30 epochs playground, learning rate $\eta \approx 8.3 \times 10^{-6} \times$ batch size, weight decay $\alpha = 6 \times 10^{-5}$, Adam $\beta_1 = 0.7$, $\beta_2 = 0.9$, AGC clipping 7.5×10^{-2}
	+ Combinatorial Laplacian	Top $k = 25$ eigenvectors (degree normalized Eq. 3), dropout $p_{\text{pos}} = 0.15$, w/o SignNet
	+ Magnetic Laplacian	Top $k = 25$ eigenvectors (deg. norm.), rel. potential $p' = 1/4$, dropout $p_{\text{pos}} = 0.15$, w/o SignNet
	+ Random Walk	$k = 3$ random walk steps, personalized page rank restart probability $p_r = 0.05$
OGB Code2	Sequential data	15 epochs, learning rate $\eta / approx 5.4 \times 10^{-5} \times$ batch size, weight decay $\alpha = 7.5 \times 10^{-5}$, Adam $\beta_1 = 0.75$, $\beta_2 = 0.935$, AGC clipping 0.1, 3 mess. passing steps
	Our dataset constr.	32 epochs, learning rate $\eta = 5.4 \times 10^{-5} \times$ batch size, weight decay $\alpha = 6 \times 10^{-5}$, Adam $\beta_1 = 0.9$, $\beta_2 = 0.95$, AGC clipping 5×10^{-2}, 3 mess. passing steps
	+ Magnetic Laplacian	Top $k = 25$ eigenvectors (deg. norm.), rel. potential $p' = 1/4$, dropout $p_{\text{pos}} = 0.15$, SignNet w/ GNN
	+ Random Walk	$k = 3$ random walk steps, personalized page rank restart probability $p_r = 0.05$

G. Experimental Setup

We use the AdamW optimizer (Loshchilov & Hutter, 2019) and employ early stopping, using the validation data. Moreover, we apply adaptive gradient clipping (AGC) as proposed by Brock et al. (2021) and decay the learning rate with cosine annealing (Loshchilov & Hutter, 2017). We report peak learning rate in Table G.1. For the playground classification tasks § 5, we train on one Nvidia GeForce GTX 1080TI with 11 GB RAM. Regression as well as sorting network results are obtained with a V100 with 40 GB RAM. For training the models on function name prediction dataset, we used four Google Cloud TPUv4 (behaves like 8 distributed devices). For eased reproduction of results, we also provide configuration for a single V100. In the single device setup, training with precomputed eigenvectors and eigenvalues of the Magnetic Laplacian requires less than 4 hours.

Models. We study two architectures: Fig. G.1a a standard transformer encoder that relies on the positional encodings for structure awareness and Fig. G.1b a hybrid GNN transformer architecture, also called Structure Aware Transformer (Chen et al., 2022). The latter is motivated via the connection of the self-attention to kernels and the GNN resembles here a learnable graph kernel. Additionally, the Structure Aware Transformer uses the node degree for weighting the residual connection around self-attention in each transformer layer.

GNN architecture We follow the generic GNN "framework" of Battaglia et al. (2018) w/o a global state. Their framework subsumes most major spatial message-passing schemes. We tested various variants but only report the best-performing model. The used GNN alternatingly updates edge embeddings $\mathbf{e}_p^{(l)}$ and node embeddings $\mathbf{v}_j^{(l)}$, with layer l, node index $v \in V$, and edge index p from the set of edges $(u, v) \in E$. Specifically, $\mathbf{e}_p^{(l)} = f_e(\mathbf{e}_p^{(l-1)} || \sum_{k \in \mathcal{V}_\leftarrow(p)} \mathbf{v}_k^{(l-1)} || \sum_{k \in \mathcal{V}_\rightarrow(p)} \mathbf{v}_k^{(l-1)})$ and $\mathbf{v}_j^{(l)} = f_v(\mathbf{v}_j^{(l-1)} || \sum_{k \in \mathcal{E}_\leftarrow(j)} \mathbf{e}_k^{(l)} || \sum_{k \in \mathcal{E}_\rightarrow(j)} \mathbf{e}_k^{(l)})$ with concate-
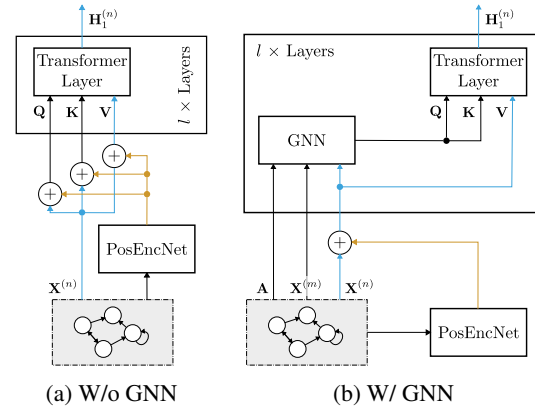

(a) W/o GNN   (b) W/ GNN

Figure G.1: Architectures of structure-aware transformers that we study in this work. (a) solely relies on positional encodings for structure awareness and (b) resembled the Structure Aware Transformer (SAT) (Chen et al., 2022). Analogously, to their architecture for the OGB Code2 dataset, here we omit the subgraph sampling. We only show the first of l layers. Subsequent layers $\boldsymbol{H}^{(j)}$ for $1 \leq j \leq l$ operate on the input data besides the node embeddings. $\boldsymbol{H}^{(l)}$ is used for the downstream task.

nation || and the sets of incoming nodes $\mathcal{V}_\leftarrow(p)$ as well as outgoing nodes $\mathcal{V}_\rightarrow(p)$ of edge p. Respectively, $\mathcal{E}_\leftarrow(j)$ and $\mathcal{E}_\rightarrow(j)$ are the sets of incoming and outgoing edges of node j. f_e and f_v are MLPs. For the undirected GNN we sum up forward and backward messages instead of concatenating them.

Batching. The maximum batch size per device is 48, except for the tasks in § 5, where we use a batch size of 96. Since we use JAX for our experiments, we have constraints on the tensor shape variations. Thus, in each batch, we consider graphs of similar size to avoid excessive padding. Moreover, we increase the batch size $4\times$ and $2\times$ if $n < 256$ and $n < 512$, respectively.

Hyperparameters. We choose the hyperparameters for each model based on a random search over the important learning parameters like learning rate, weight decay, and the parameters of AdamW (30 random draws for the sorting network and 50 for OGB). Due to the small and mostly insignificant differences, we consolidated the parameters for both architectures (Fig. G.1) and all positional encodings. That is, hyperparameters unspecific to the respective positional encoding are shared. Moreover, for the results in § 5, we use the parameters for the sorting network task (§ 6) without additional tuning. We list the important hyperparameters in Table G.1.

H. Scalability

Both positional encodings, namely random walks (§ 4) and eigenvectors of Magnetic Laplacian (§ 3) can be calculated efficiently. Although, for the random walk encodings one has to be cautious since even for a small number of steps k the transition matrix becomes practically dense (complexity $\mathcal{O}(n^2)$) if not using some sparsification technique. For the scalability of personalized page rank encodings within a neural netowork, we refer to (Bojchevski et al., 2020).

Moreover, we only study the standard self-attention that scales with $\mathcal{O}(n^2)$. Scalable alternatives were extensively studied, e.g., (Choromanski et al., 2020; Kitaev et al., 2020), also covering the graph domain (Dwivedi & Bresson, 2021; Rampášek et al., 2022; Wu et al., 2022). It is straightforward to apply our positional encodings to most of these approaches.
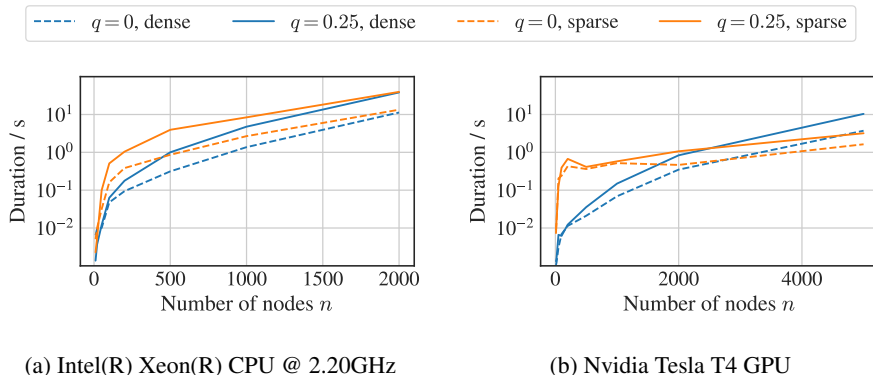

(a) Intel(R) Xeon(R) CPU @ 2.20GHz   (b) Nvidia Tesla T4 GPU

Figure H.1: Runtime required for eigendecomposition on random Erdős-Rènyi graphs.

In Fig. H.1, we study the duration of the eigendecompositon on a CPU (`scipy`) and GPU (`cupy`) over graphs of different size. We also contrast the overhead of having a Hermitian matrix ($q = 0.25$) in contrast to the decomposition of a real matrix ($q = 0$) and in which cases a sparse calculation (with $k = 25$) is beneficial over its dense equivalent. For this benchmark, we draw random Erdős-Rènyi graphs with an average degree of 5 (similar to the positional encodings playground § 5) and report the average over 10 trials via `timeit` or `cupyx.profiler.benchmark`.

In any case, once q and other hyperparameters have been chosen it is beneficial to precompute the eigenvectors for training. With precomputed eigenvectors, the training on OGB Code2 (32 epochs) finishes within 4 hours using a single V100.

I. Positional Encodings Playground

GNNs. Additionally to the baselines of Laplacian and SVD encodings, we also compare to our default GNN architecture (see § G) and MagNet (Zhang et al., 2021). MagNet is a GNN for directed graphs that uses the Magnetic Laplacian in its message passing. Specifically, they formulate each message passing step as a polynomial of the Magnetic Laplacian. We follow the default parameters of the authors and choose (absolute) potential $q = 0.1$, which is performs best out of $q \in \{0, 0.05, 0.1, 0.15, 0.2, 0.25\}$. We find that both GNNs are predictive for classifying reachability and adjacency, however, fall behind in the distance regression tasks (see Fig. I.1).

Relative random walk encodings can be constructed following the recipe in § 4. However, instead of aggregating the encodings to a node-level, we keep the $n \times n$ encodings $\zeta(v|u)$ using the a different linear transformation $f_{rw}^{(2)}$ for each transformer layer and head. The attention mechanism for each head becomes $\mathrm{softmax}(\boldsymbol{QK}^\top/\sqrt{d}+\zeta)\boldsymbol{V}$, where $\zeta \in \mathbb{R}^{n \times n}$ is the matrix containing the one-dimensional $\zeta(v|u)$. Note tha these positional encodings can be understood as a generalization of the pair-wise shortest path distances used by Ying et al. (2021) and Guo et al. (2021), if using a sufficiently large number of random walk steps.

In Fig. I.1, we show that these relative positional random walk encodings (RW rel.) consistently outperform node-level random walk encodings (RW). However, the relative positional encodings seem to be more prone to overfitting and training becomes less stable. This can, e.g., be seen in the comparably large differences between validation and test scores. Recall that the validation set is more similar to the training set than the test set due to the different number of nodes. Due to the brittleness and the additional tuning required, we do not study the relative positional encodings in the other tasks.

	F1	F1	RMSE	RMSE		F1	F1	RMSE	RMSE	
Lap. (basln)	0.63	0.63	0.23	0.51	val.	0.75	0.62	0.27	1.96	val.
	0.53	0.53	0.26	0.54	test	0.73	0.49	0.31	2.08	test
SVD (basln)	1.00	1.00	0.83	0.38	val.	1.00	1.00	1.02	1.64	val.
	1.00	1.00	0.97	0.45	test	0.97	1.00	1.12	1.86	test
GNN (basln)	0.95	0.95	1.94	0.62	val.	0.92	0.96	1.41	2.13	val.
	0.94	0.94	2.12	0.67	test	0.91	0.95	1.51	2.34	test
MagNet (basln)	0.97	0.97	1.99	0.82	val.	0.89	0.95	1.62	2.34	val.
	0.97	0.96	2.17	0.86	test	0.87	0.94	1.72	2.56	test
RW (ours)	0.97	0.97	1.22	0.65	val.	1.00	0.97	1.00	1.24	val.
	0.95	0.95	1.33	0.68	test	0.99	0.94	1.06	1.36	test
RW rel. (ours)	0.99	0.98	0.81	0.35	val.	1.00	0.96	0.76	0.97	val.
	0.96	0.95	0.98	0.44	test	0.98	0.79	0.91	1.54	test
ML w/o SignNet (ours)	1.00	1.00	0.22	0.33	val.	1.00	1.00	0.27	0.93	val.
	1.00	1.00	0.25	0.38	test	1.00	1.00	0.31	1.06	test
ML w/ SignNet (ours)	1.00	0.99	0.54	0.81	val.	1.00	1.00	0.64	1.71	val.
	1.00	0.99	0.58	0.83	test	1.00	1.00	0.68	1.85	test
	reach. (1)	adj. (2)	u. dist (3)	d. dist (4)		reach. (1)	adj. (2)	u. dist (3)	d. dist (4)	

Figure I.1: Complimentary results to Fig. 5 for *(1) reachability*, *(2) adjacency*, *(3) undirected distance*, and *(4) directed distance*.

SignNet with MLP. Additionally to the Magnetic Laplacian encodings w/o SignNet (as presented in Fig. 5), we report results w/ SignNet $f_{elem}(-\gamma_j) + f_{elem}(\gamma_j)$ using using an MLP for f_{elem}. As we can see in Fig. I.1, the encodings w/ SignNet perform considerably worse than the encodings w/o SignNet. As hypothesised in § 3, one reason could be that our convention for choosing the eigenvectors sign resolves the sign ambiguity to a sufficient extend. Moreover, SignNet with an element-wise MLP behaves similarly to processing the absolute value of the eigenvectors. If using solely the absolute values, we loose the information about relative differences between different nodes that include sign changes. Note that this finding crucially rely on the usage of a point-wise MLP and if using a GNN (e.g., as we do for function name prediction in § 7) SignNet appears to help achieving better performance.

J. Random Walk Hyperparameter Study

We next study our most important design choices along the impact of the number of random walk steps k. We find that these decision are rudimental for random walk positional encodings on directed graphs and that two or three random walk steps k are sufficient for this task.

In Fig. J.1a, we show the random walk encodings alike Li et al. (2020). That is, solely relying on the transition matrix $\boldsymbol{T} = \boldsymbol{A}\boldsymbol{D}_{\text{out}}^{-1}$, without backward direction and without Personalizd Page Rank (PPR). In Fig. J.1b, we show the results with reversed random walk (transition matrix $\boldsymbol{R} = \boldsymbol{A}^\top \boldsymbol{D}_{\text{in}}^{-1}$). In Fig. J.1c, we study the random walk encodings as presented in § 4, including backward random walk and PPR. Both, choices substantially boost performance, although, for PPR the gains diminish for $k > 3$ (on the admittedly small graphs in this task).

	(a) W/o rev., w/o PPR				(b) W/ reversal, w/o PPR				(c) W/ reversal, w/ PPR				
	F1		RMSE		F1		RMSE		F1		RMSE		
$k=1$	0.80	0.29	1.64	2.46	0.89	0.58	1.43	2.28	0.99	0.84	1.17	1.69	val.
	0.76	0.23	1.76	2.59	0.86	0.50	1.47	2.42	0.98	0.78	1.23	1.81	test
$k=2$	0.89	0.49	1.50	2.28	0.98	0.86	1.20	1.88	1.00	0.94	1.14	1.52	val.
	0.87	0.39	1.53	2.41	0.96	0.79	1.25	2.02	0.99	0.91	1.20	1.65	test
$k=3$	0.92	0.63	1.44	2.16	0.99	0.94	1.07	1.51	1.00	0.97	1.09	1.34	val.
	0.91	0.56	1.47	2.29	0.99	0.91	1.13	1.64	0.99	0.94	1.15	1.46	test
$k=4$	0.94	0.72	1.40	2.06	1.00	0.97	1.00	1.29	1.00	0.97	1.13	1.15	val.
	0.92	0.66	1.44	2.19	0.99	0.95	1.06	1.42	0.99	0.95	1.18	1.27	test
$k=5$	0.95	0.76	1.37	1.89	1.00	0.98	0.95	1.20	1.00	0.98	1.04	1.06	val.
	0.93	0.71	1.41	2.01	1.00	0.96	1.01	1.32	1.00	0.97	1.10	1.18	test
	reach. (1)	adj. (2)	u. dist (3)	d. dist (4)	reach. (1)	adj. (2)	u. dist (3)	d. dist (4)	reach. (1)	adj. (2)	u. dist (3)	d. dist (4)	

(a) W/o rev., w/o PPR (Li et al., 2020)   (b) W/ reversal, w/o PPR   (c) W/ reversal, w/ PPR

Figure J.1: Hyperparemter study for random walk encodings on the playground tasks: *(1) reachability*, *(2) adjacency*, *(3) undirected distance*, and *(4) directed distance* Dark green encodes the best scores and bright green bad ones. For F1 score high values are better and for RMSE low values.

K. Sorting Networks Dataset Construction

We give the data generation process for a single sorting network in Algorithm K.1. Additionally, we only consider sorting networks with less than 512 comparators and abort early if this bound is exceeded. Since adding a comparator cannot diminish any progress of the sorting network, we only need to generate all possible test sequences once in the beginning and sort ti successively. Moreover, we make use of the so-called "0-1-principle" (Knuth, 1973). That is, if a sorting network sorts all possible sequences in $\{0, 1\}^p$, then it sorts all possible sequences consisting of comparable elements. Once a valid sorting network was constructed, we drop the last comparator $C[:-1]$ for an instance of an incorrect sorting network. Moreover, for test and validation, we include another (typically incorrect) sorting network via swapping the order of comparators $C[::-1]$.

L. Sorting Networks Are Near-Sequential

All the comparators perform a data-dependent in-place swap operation. In other words, there are no buffers to store, e.g., a value at the beginning of the program and retrieve it at the end. Consequently, edges in the resulting directed graph (e.g. Fig. 6) are typically connected to close ancestor nodes and align well with the main diagonal of the adjacency matrix. We give an example in Fig. K.1. For this reason, we call the graphs in the sorting network task *near-sequential*.

Due to the near-sequentiality, sinusoidal positional encodings are intuitively an appropriate choice (see § 2). However, even these near-sequential graphs can have a huge amount of topological sorts. Enumerating all topological orders for the training instances rarely terminates within a few minutes (using python). We estimate, that the number of topological sorts typically

---

Algorithm K.1 Generate Sorting Network

1: **Input:** Number of nodes set N
2: Sample uniformly $n \sim U(N)$
3: Empty list of comparators $C \leftarrow []$
4: Unsorted sequences $S \leftarrow$ all possible sequences($\{0, 1\}, n$)
5: Unsorted locations $L \leftarrow \{0, 1, \ldots, n - 1\}$
6: **while** $|S| \geq n + 1$ **do**
7:     Sample uniformly $(u, v) \sim U(L)$ without replacement
8:     **if** $(u, v) \neq C[-1]$ **then**
9:         $C \leftarrow C + [(u, v)]$
10:        $S \leftarrow$ sort positions(S, u, v)
11:        $L \leftarrow$ determine unsorted(S)
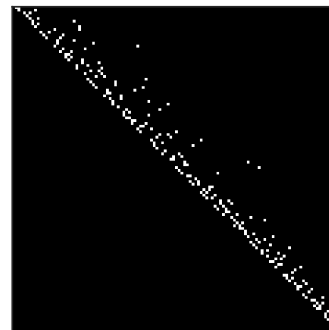12:   **end if**
13: **end while**
14: Return C

Figure K.1: Sorting networks are *near-sequential* since the edges align well with main diagonal.

exceeds 1×10^6. This, surprisingly high number of topological sorts could be the reason why the positional encodings for directed graphs are superior to the sinusoidal encodings (see § 6) and shows the significance of this relationship.

M. Application: Function Name Prediction

We next describe how we construct the directed graph in the function name prediction task. Recall, that we construct a data-flow-centric directed graph that is also able to handle the sharp bits like if-else, loops, and exceptions. In our graph, we avoid sequential connections that originate from the sequence of statements in the source. We aim for a similar reduction of the input space size to the sorting network task § 6. To explain how we construct this graph, we will first give a high-level description of the design narratives. Then, we include the Abstract Syntax Tree (AST) in the discussion.

Design principles. In Fig. M.1, we give an example function and give a data-flow-centric dependency graph for the actual computations. The function can be clustered into three blocks (excluding function definition and return statement): (1) variable d is calculated, (2) if-else-statement for refining the value of variable a, and (3) variable b is changed if negative. These three blocks are each represented by a grey box. Further, we highlight (sub-) blocks white that correspond to the bodies of the if-else statement.


```python
def func(a, b, c):
    d = min(a, b)
    if a > 0:
        e = a ** 2
        f = b ** 2
        a = sqrt(e + f)
    else:
        a = -a
    if b < 0:
        b = b + d
    return a + b
```
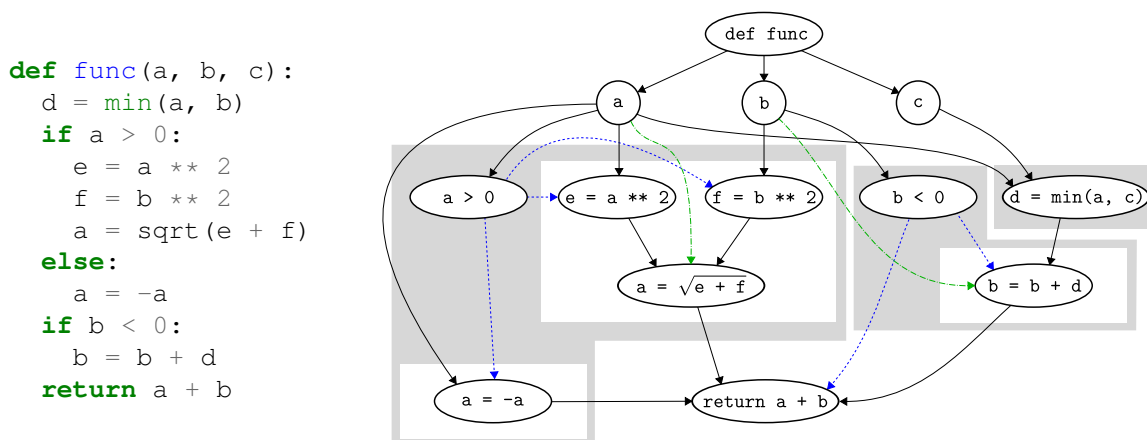
Figure M.1: Exemplary data-flow-centric graph construction

We connect nodes based on the required inputs for the computation (aka data flow). Moreover, we add edges to describe the control flow of the if-statements (dashed blue lines). Last, we add edges if values are being overwritten (dash-dotted green

line). Conceptually, we generate a Directed Acyclic Graph (DAG) for each block and then recursively traverse the hierarchy and connect the contained instructions. Hence, the resulting graph is not necessarily acyclic.

Order of computation. In this example, each computation can only take place after all its immediate ancestors have been calculated (and if these values are kept in memory). Vice versa, all computations could take in arbitrary order as long as the parent nodes have been executed. For general functions, the picture is a bit more complicated (but similar) due to, for example, loops, exceptions, or memory constraints.

Hierarchy. To generate the connections we rely on the fact that (Python) code can be decomposed into a hierarchy of blocks. Except for ancestor and successor nodes, these blocks build a mutually exclusive separation of instructions. This decomposition is what an AST provides. While also prior work uses ASTs for their graph construction, they retain the sequential structure of the source code in the graph construction. For example in Fig. M.2a, we show the graph constructed by the OGB Code2 dataset (Hu et al., 2020) for the code in Fig. M.3. From this, it should be clear without additional explanation why we argue that the AST solely enriches the sequence of code statements with a hierarchy. In stark contrast, our graph construction (Fig. M.2b) is by far not as sequential.
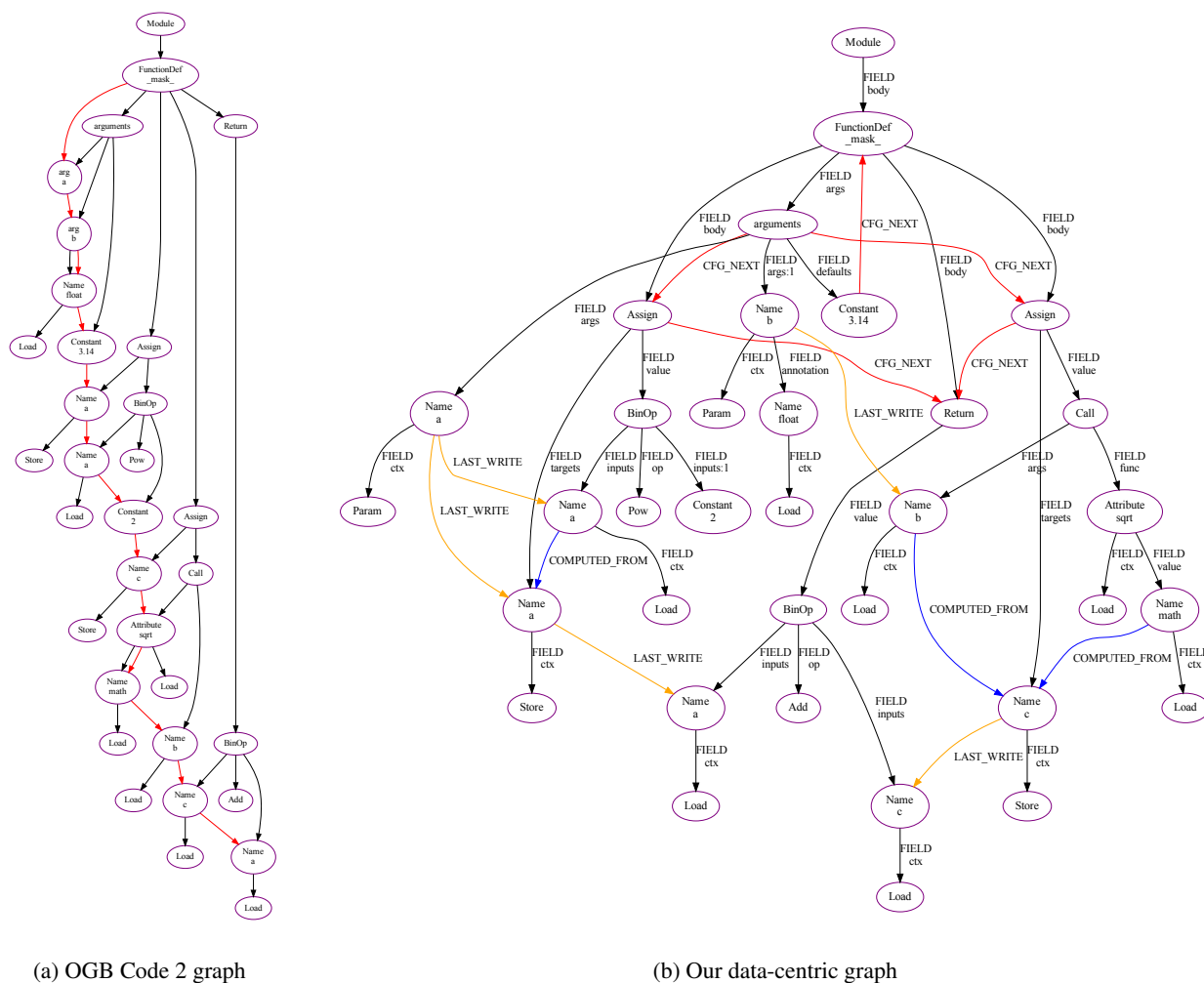

(a) OGB Code 2 graph

(b) Our data-centric graph

Figure M.2: Comparison of OGB Code2's graph construction to ours.

Sequentialization. Generating semantically equivalent sequences of program statements from such a directed graph is more challenging than determining feasible orders of computation or in the sorting network task § 6. For example, in DAG of Fig. M.1, not every possible topological sort corresponds to a possible sequentialization of the program. To determine sequentializations one needs to consider the hierarchical block structure. For example, it is possible to reorder the blocks

highlighted in grey, depending on their dependencies. However, our data and control flow does not capture all dependencies required to generate the program. For example, as already hinted above, one caveat resides in the availability of intermediate values. Although the first block (to determine d) and second block (if else construct) are independent in the shown graph, *overwriting* the value of a has not been modeled. In other words, it would make a difference to swap these blocks since a changes its value in the second block. Thus, for constructing a possible sequence of program instructions, we would also need to address changing variable values. For example, we could assign a new and unique name after changing a variable's value (as in functional programming or like a compiler does). Alternatively, adding further edges could be sufficient. Nevertheless, these dependencies are not important for the semantic meaning of a program.

OGB's graph construction first converts the source to AST and adds additional edges to simulate the sequential order of instructions. In Fig. M.2a, the black edges are the edges from the AST and the red edges for the sequential order.

Our graph construction. We also construct the AST from source (FIELD edges) and build on top of the graph construction / static code analysis of Bieber et al. (2022). In the example in Fig. M.2b, we have the same amount of nodes as Fig. M.2a except for two "Param" nodes belonging to the argument nodes.

```
def transform_add(
    a, b: float = 3.14):
  a = a**2
  c = math.sqrt(b)
  return c + a
```

Figure M.3: Code used for Fig. M.2.

Similarly to the example in Fig. M.1, we augment the AST with additional edges mimicking the data flow and program flow. Here, we have Control Flow Graph edges CFG_NEXT that model the possible order of statement execution. Moreover, the variable nodes (close to leaf nodes) are connected by LAST_WRITE and CALCULATED_FROM edges. These edges model where the same variable has been written the last time and from which variables is was constructed. Additionally, we use a CALLS edge that model function calls / recursion (not present in this example). Thereafter, since the task is function name prediction, we need to prevent data leakage. For this, we mask the occurrences of the function name in its definition as well as in recursive calls.

Comparison to Bieber et al. (2022). We largely follow them and build upon their code base. The most significant difference is the avoidance of unnecessary sequentialism. Specifically, (1) their CFG_NEXT edges connect instructions sequentially while ours form a dependency graph, and (2) we omit their LAST_READ edges. Moreover, we address commutative properties of the basic python operations (And, Or, Add, Mult, BitOr, BitXor, and BitAnd). This can also be observed in Fig. M.2b, where we name the edges for the inputs to these operations input and concatenate :<order> if the operation is non-commutative and <order> > 1. Last, we do not reference the tokenized source code and choose a less verbose graph similar to OGB.

Sequentialization. Reconstructing the code from AST is a straightforward procedure. Following our discussion above, we need to acknowledge the hierarchical structure to retrieve valid programs. Fortunately, this hierarchical structure is provided by the AST. However, similar to the example above, we do not model all dependencies for an immediate sequentialization. However, as stated before, these dependencies are not important for semantics. Thus, we most certainly map more semantically equivalent programs to the same directed graph, as if we would compare to a graph construction that models all dependencies.