GraphGen: A Scalable Approach to Domain-agnostic Labeled Graph Generation

Nikhil Goyal∗
Indian Institute of Technology, Delhi
nikhil01sep@gmail.com

Harsh Vardhan Jain∗
Indian Institute of Technology, Delhi
harshvardhanjain7@gmail.com

Sayan Ranu
Indian Institute of Technology, Delhi
sayanranu@cse.iitd.ac.in

Abstract

Graph generative models have been extensively studied in the data mining literature. While traditional techniques are based on generating structures that adhere to a pre-decided distribution, recent techniques have shifted towards learning the distribution directly from the data. While learning-based approaches have imparted significant improvement in quality, some limitations remain to be addressed. First, learning graph distributions introduces additional computational overhead, which limits their scalability to large graph databases. Second, many techniques only learn the structure and do not address the need to also learn node and edge labels, which encode important semantic information and influence the structure itself. Third, existing techniques often incorporate domain-specific rules and lack generalizability. Fourth, the experimentation of existing techniques is not comprehensive enough due to either using weak evaluation metrics or focusing primarily on synthetic or small datasets. In this work, we develop a domain-agnostic technique called GraphGen to overcome all of these limitations. GraphGen converts graphs to sequences using minimum DFS codes. Minimum DFS codes are canonical labels and capture the graph structure precisely along with the label information. The joint complex distributions between structure and semantic labels are learned through a novel LSTM architecture. Extensive experiments on million-sized, real graph datasets show our new model scalable and do not address the need to also learn node and edge labels, which encode important semantic information and influence the structure itself. Third, existing techniques often incorporate domain-specific rules and lack generalizability. Fourth, the experimentation of existing techniques is not comprehensive enough due to either using weak evaluation metrics or focusing primarily on synthetic or small datasets. In this work, we develop a domain-agnostic technique called GraphGen to overcome all of these limitations. GraphGen converts graphs to sequences using minimum DFS codes. Minimum DFS codes are canonical labels and capture the graph structure precisely along with the label information. The joint complex distributions between structure and semantic labels are learned through a novel LSTM architecture. Extensive experiments on million-sized, real graph datasets show our new model scalable and do not \( n \) and \( m \) denote the number of nodes and edges respectively. Any other variable is a hyper-parameter of the respective model. We consider a model scalable if its complexity is linear in \( n \) and \( m \).

| Techniques | Domain-agnostic | Node Labels | Edge Labels | Scalable | Complexity |
|------------|-----------------|-------------|-------------|----------|------------|
| TaGNN[4]   | O               | ✓           | ✓           | ✓        | O(m^2)     |
| GVAE[3]    | O               | ✓           | ✓           | ✓        | O(m^2)     |
| GraphVAE[34] | O             | ✓           | ✓           | ✓        | O(m^2)     |
| DeepSAGE[9] | O               | ✓           | ✓           | ✓        | O(m^2)     |
| GRAPHSN[12] | O              | ✓           | ✓           | ✓        | O(m^2)     |
| GraphGen   | O               | ✓           | ✓           | ✓        | O(m^2)     |
| TaGenese   | O               | ✓           | ✓           | ✓        | (m/2)n     |

A key area that lacks development is the ability to directly learn patterns, both local as well as global, from a given set of observed graphs and use this knowledge to generate graphs instead of making prior assumptions. In this work, we bridge this gap by developing a domain-agnostic graph generative model for labeled graphs without making any prior assumption about the dataset. Such an approach reduces the usability barrier for non-technical users and allows for a learning-based approach[8, 16, 19, 21, 34, 40]. The apriori assumption introduces several limitations. First, due to pre-selecting the family of graphs, i.e., distributions modeling some structural properties, they cannot adapt to datasets that either do not fit well or evolve to a different structural distribution. Second, these models only incorporate structural properties and do not look into the patterns encoded by labels. Third, these models assume the graph to be homogeneous. In reality, it is possible that different local communities (subgraphs) within a graph adhere to different structural distributions.

Learning-based graph generative models: Addressing the above outlined limitations, recent techniques have shifted towards a learning-based approach[8, 16, 19, 21, 34, 40]. While impressive progress has been made through this approach, there is scope to further improve the performance of graph generative models. We outline the key areas we target in this work.
• **Domain-agnostic modeling**: Several models have been proposed recently that target graphs from a specific domain and employ domain-specific constraints[7, 15, 20, 22, 24, 31, 39] in the generative task. While these techniques produce excellent results on the target domain, they do not generalize well to graphs from other domains.

• **Labeled graph generation**: Real-world graphs, such as protein interaction networks, chemical compounds, and knowledge graphs, encode semantic information through node and edge labels. To be useful in these domains, we need a generative model that jointly captures the relationship between structure and labels (both node and edge labels).

• **Data Scalability**: It is not uncommon to find datasets containing millions of graphs[14]. Consequently, it is important for any generative model to scale to large training datasets so that all of the information can be exploited for realistic graph generation. Many of the existing techniques do not scale to large graph databases[8, 19, 21, 34, 40]. Often this non-scalability stems from dealing with exponential representations of the same graph, complex neural architecture, and large parameter space. For example, LGGAN[8] models graph through its adjacency matrix and therefore has $O(n^3)$ complexity, where $n$ is the number of nodes in the graph. GraphRNN[40] is the most scalable among existing techniques due to employing a smaller recurrent neural network (RNN) of $O(m^2)$ complexity, where $M$ is a hyper-parameter. In GraphRNN, the sequence representations are constructed by performing a large number of breadth-first-search (BFS) enumerations for each graph. Consequently, even if the size of the graph dataset is small, the number of unique sequences fed to the neural network is much larger, which in turn affects scalability.

Table 1 summarizes the limitations of learning-based approaches. GraphGen addresses all of the above outlined limitations and is the first technique that is domain-agnostic, assumption-free, models both labeled and unlabeled graphs, and data scalable. Specifically, we make the following contributions.

- We propose the problem of learning a graph generative model that is assumption-free, domain-agnostic, and captures the complex interplay between semantic labels and graph structure (§ 2).

- To solve the proposed problem, we develop an algorithm called GraphGen that employs the unique combination of graph canonization with deep learning. Fig. 1 outlines the pipeline. Given a database of training graphs, we first construct the canonical label of each graph[17, 38]. The canonical label of a graph is a string representation such that two graphs are isomorphic if and only if they have the same canonical label. We use DFS codes for canonization. Through canonization, the graph generative modeling task converts to a sequence modeling task, for which we use Long Short-term Memory (LSTM)[13] (§ 3).

- We perform an extensive empirical evaluation on real million-sized graph databases spanning three different domains. Our empirical evaluation establishes that we are significantly better in quality than the state-of-the-art techniques across an array of metrics, more robust, and 4 times faster on average (§ 4).

2 Problem Formulation

As a notational convention, a graph is represented by the tuple $G = (V, E)$ consisting of node set $V = \{v_1, \cdots, v_n\}$ and edge set $E = \{(v_i, v_j) | v_i, v_j \in V\}$. Graphs may be annotated with node and edge labels. Let $L_e : V \rightarrow \mathbb{V}$ and $L_e : E \rightarrow \mathbb{E}$ be the node and edge label mappings respectively where $\mathbb{V}$ and $\mathbb{E}$ are the set of all node and edge labels respectively. We use the notation $L(v)$ and $L(e)$ to denote the labels of node $v$ and edge $e$ respectively. We assume that the graph is connected and there are no self-loops.

The goal of a graph generative model is to learn a distribution $p_{model}(G)$ over graphs, from a given set of observed graphs $G = \{G_1, G_2, \cdots, G_m\}$ that is drawn from an underlying hidden distribution $p(G)$. Each $G_i$ could have a different number of nodes and edges, and could possibly have a different set of node labels and edge labels. The graph generative model is effective if the learned distribution is similar to the hidden distribution, i.e., $p_{model}(G) \approx p(G)$. More simply, the learned generative model should be capable of generating similar graphs of varied sizes from the same distribution as $p(G)$ without any prior assumption about structure or labeling.

2.1 Challenges

- **Large and variable output space**: The structure of a graph containing $n$ nodes can be characterized using an $n \times n$ adjacency matrix, which means a large output space of $O(n^2)$ values. With node and edge labels, this problem becomes even more complex as the adjacency matrix is no longer binary. Furthermore, the mapping from a graph to its adjacency matrix is not one-to-one. A graph with $n$ nodes can be equivalently represented using $n^2$ adjacency matrices corresponding to each possible node ordering. Finally, $n$ itself varies from graph to graph.

- **Joint distribution space**: While we want to learn the hidden graph distribution $p(G)$, defining $p(G)$ itself is a challenge since graph structures are complex and composed of various properties such as node degree, clustering coefficients, node and edge labels, number of nodes and edges, etc. One could learn a distribution over each of these properties, but that is not enough since these distributions are not independent. The key challenge is therefore to learn the complex dependencies between various properties.

- **Local and global dependencies**: The joint distribution space itself may not be homogeneous since the dependence between various graph parameters varies across different regions of a graph. For example, not all regions of a graph are equally dense. We overcome these challenges through GraphGen.
GraphGen: A Scalable Approach to Domain-agnostic Labeled Graph Generation
WWW ’20, April 20–24, 2020, Taipei, Taiwan

3 GRAPHGEN: Generative model for labeled graphs

Instead of using adjacency matrices to capture graph structures, we perform graph canonization to convert each graph into a unique sequence. This step provides two key advantages. First, the length of the sequence is |E| for graph G(V, E) instead of |V|^2. Although in the worst case, |E| = |V|, in real graphs, this is rarely seen. Second, there is a one-to-one mapping between a graph and its canonical label instead of |V|! mappings. Consequently, the size of the output space is significantly reduced, which results in faster and better modeling.

3.1 DFS Codes: Graphs to Sequences

First, we formally define the concept of graph canonization.

**Definition 1 (Graph Isomorphism).** Graph G1 = (V1, E1) is isomorphic to G2 = (V2, E2) if there exists a bijection φ such that for every vertex v ∈ V1, φ(v) ∈ V2 and for every edge e = (u, v) ∈ E1, φ(e) = (φ(u), φ(v)) ∈ E2. If the graphs are labeled, we also need to ensure that the labels of mapped nodes and edges are the same, i.e., L(φ(v)) = L(v) and L(φ(e)) = L(e).

**Definition 2 (Canonical Label).** The canonical label of a graph is a string representation such that two graphs have the same label if and only if they are isomorphic to each other.

We use DFS codes[38] to construct graph canonical labels. DFS code encodes a graph into a unique edge sequence by performing a depth first search (DFS). The DFS traversal may start from any node in the graph. To illustrate, consider Fig 2(a). A DFS traversal starting at v0 on this graph is shown in Fig 2(b) where the bold edges show the edges that are part of the DFS traversal. During the DFS traversal, we assign each node a timestamp based on when it is discovered; the starting node has the timestamp 0. Fig. 2(b) shows the timestamps if the order of DFS traversal is (v0, v1), (v1, v2), (v1, v2). We use the notation t_u to denote the timestamp of node u. Any edge (u, v) that is part of the DFS traversal is guaranteed to have t_u < t_v, and we call such an edge a forward edge. On the other hand, the edges that are not part of the DFS traversal, such as (v3, v0) in Fig 2(b), are called backward edges. In Fig 2(b), the bold edges represent the forward edges, and the dashed edge represents the backward edge. Based on the timestamps assigned, an edge (u, v) is described using a 5-tuple (t_u, t_v, L_u, L_v, L_{uv}), where L_u and L_v denote the node and edge labels of u and v respectively.

A given DFS traversal, our goal is to impose a total ordering on all edges of the graph. Forward edges already have an ordering between them based on when they are traversed in DFS. To incorporate backward edges, we enforce the following rule:

- Any backward edge (u, u') must be ordered before all forward edges of the form (u, v).

- Any backward edge (u, u') must be ordered after the forward edge of the form (w, u), i.e., the first forward edge pointing to u.
- Among the backward edges from the same node u of the form (u, u') and (u, u''), (u, u'') has a higher order if t_u'' < t_u'.

With the above rules, corresponding to any DFS traversal of a graph, we can impose a total ordering on all edges of a graph and can thus convert it to a sequence of edges that are described using their 5-tuple representations. We call this sequence representation a DFS code.

**Example 3.1.** Revisiting Fig. 2(b), the DFS code is [(0, 1, X, a, X) (1, 2, X, a, Z) (2, 0, Z, b, X) (1, 3, X, b, Y)]. A second possible DFS traversal is shown in Fig. 2(c), for which the edge sequence is [(0, 1, X, a, X) (1, 2, b, Z) (2, 0, Z, a, X) (0, 3, X, b, Y)].

Since each graph may have multiple DFS traversals, we choose the lexicographically smallest DFS code based upon the lexicographical ordering proposed in[38]. Hereon, the lexicographically smallest DFS code is referred to as the minimum DFS code. We use the notation F(G) = S = [s_1, ..., s_m] to denote the minimum DFS code of graph G, where m = |E|.

**Theorem 1.** F(G) is a canonical label for any graph G.[38]

**Example 3.2.** The DFS code for Fig. 2(b) is smaller than Fig. 2(c) since the second edge tuple (1, 2, X, a, Z) of Fig. 2(b) is smaller than (1, 2, Z, b, X) of Fig. 2(c) as a < b. Since the first edge tuple in both sequences are identical the tie is broken through the second tuple.

3.1.1 Computation Cost and Properties Constructing the minimum DFS code of a graph is equivalent to solving the graph isomorphism problem. Both operations have a worst-case computation cost of O(|V|^3) since we may need to evaluate all possible permutations of the nodes to identify the lexicographically smallest one. Therefore an important question arises: How can we claim scalability with a factorial computation complexity? To answer this question, we make the following observations.

- **Labeled graphs are ubiquitous:** Most real graphs are labeled. Labels allow us to drastically prune the search space and identify the minimum DFS code quickly. For example, in Fig. 2(a), the only possible starting nodes are v0 or v1 as they contain the lexicographically smallest node label "X". Among them, their 2-hop neighborhoods are enough to derive that v0 must be the starting node of the minimum DFS code as (1, 2, X, a, Z) is the lexicographically smallest possible tuple after (0, 1, X, a, X) in the graph in Fig 2(a). We empirically substantiate this claim in § 4.3.2.
- **Invariants:** What happens if the graph is unlabeled or has less diversity in labels? In such cases, vertex and edge invariants can be used as node and edge labels. Invariants are properties of nodes and edges that depend only on the structure and not on graph representations such as adjacency matrix or edge sequence. Examples include node degree, betweenness centrality, clustering coefficient, etc. We empirically study this aspect further in § 4.4.
- **Precise training:** Existing techniques rely on the neural network to learn the multiple representations that correspond to the same graph. Many-to-one mappings introduce imprecision and bloat the modeling task with redundant information. In GRAPHGEN, we feed precise one-to-one graph representations, which allows us to use a more lightweight neural architecture.
with multiple graph representations are handled algorithmically, which leads to significant improvement in quality and scalability.

### 3.2 Modeling Sequences

#### 3.2.1 Model Overview

Owing to the conversion of graphs to minimum DFS codes, and given that DFS codes are canonical labels, modeling a database of graphs \( G = \{ G_1, \ldots, G_M \} \) is equivalent to modeling their sequence representations \( S = \{ F(G_1), \ldots, F(G_M) \} = \{ s_1, \ldots, s_M \} \), i.e., \( p_{model}(G) = p_{model}(S) \). To model the sequential nature of the DFS codes, we use an auto-regressive model to learn \( p_{model}(S) \). At inference time, we sample DFS codes from this distribution instead of directly sampling graphs. Since the mapping of a graph to its minimum DFS code is a bijection, i.e., \( G = F^{-1}(F(G)) \), the graph structure along with all node and edge labels can be fully constructed from the sampled DFS code.

Fig. 3 provides an overview of GraphGen model. For graph \( G(V, E) \) with \( F(G) = S = \{ s_1, \ldots, s_M \} \), the proposed auto-regressive model generates \( s_i \) sequentially from \( i = 1 \) to \( i = m = |E| \). This means that our model produces a single edge at a time, and since it belongs to a DFS sequence, it can only be between two previously generated nodes (backward edge) or one of the already generated nodes and an unseen node (forward edge), in which case a new node is also formed. Consequently, only the first generated edge produces two new nodes, and all subsequent edges produce at most one new node.

#### 3.2.2 Model Details

We now describe the proposed algorithm to characterize \( p(S) \) in detail. Since \( S \) is sequential in nature, we decompose the probability of sampling \( S \) from \( p(S) \) as a product of conditional distribution over its elements as follows:

\[
p(S) = \prod_{i=1}^{m+1} p(s_i | s_1, \ldots, s_{i-1}) = \prod_{i=1}^{m+1} p(s_i | s_{<i}) \tag{1}
\]

where \( m \) is the number of edges and \( s_{m+1} \) is end-of-sequence token EOS to allow variable length sequences. We denote \( p(s_i | s_1, \ldots, s_{i-1}) = p(s_i | s_{<i}) \) in further discussions.

Recall, each element \( s_i \) is an edge tuple of the form \( (t_u, t_v, L_u, L_v, L_e) \), where \( e = (u, v) \). We make the simplifying assumption that timestamps \( t_u, t_v \), node labels \( L_u, L_v \), and edge label \( L_e \) at each generation step of \( s_i \) are independent of each other. This makes the model tractable and easier to train. Note that this assumption is not dependent on data and hence is not forming prior bias of any sort on data. Mathematically, Eq. 1 reduces to the follows:

\[
p(s_i | s_{<i}) = p((t_u, t_v, L_u, L_e, L_v)) \mid s_{<i}) = p(t_u | s_{<i}) \times p(t_v | s_{<i}) \times p(L_u | s_{<i}) \times p(L_e | s_{<i}) \times p(L_v | s_{<i}) \tag{2}
\]

Eq. 2 is extremely complex as it has to capture complete information about an upcoming edge, i.e., the nodes to which the edge is connected, their labels, and the label of the edge itself. To capture this highly complex nature of \( p(s_i | s_{<i}) \) we propose to use expressive neural networks. Recurrent neural networks (RNNs) are capable of learning features and long term dependencies from sequential and time-series data[30]. Specifically, LSTMs[13] have been one of the most popular and efficient methods for reducing the effects of vanishing and exploding gradients in training RNNs. In recent times, LSTMs and stacked LSTMs have delivered promising results in cursive handwriting[9], sentence modeling[5] and, speech recognition[10].

In this paper, we design a custom LSTM that consists of a state transition function \( f_{trans} \) (Eq. 3), an embedding function \( f_{emb} \) (Eq. 3), and five separate output functions (Eqs. 4-8) for each of the five components of the 5-tuple \( s_i = (t_u, t_v, L_u, L_v, L_e) \). (\( -M \) in the following equations represents sampling from multinomial distribution)

\[
h_i = f_{trans}(h_{i-1}, f_{emb}(s_{i-1})) \tag{3}
\]

\[
t_u \sim \mathcal{M} \quad \theta_{t_u} = f_{t_u}(h_i) \tag{4}
\]

\[
t_v \sim \mathcal{M} \quad \theta_{t_v} = f_{t_v}(h_i) \tag{5}
\]

\[
L_u \sim \mathcal{M} \quad \theta_{L_u} = f_{L_u}(h_i) \tag{6}
\]

\[
L_v \sim \mathcal{M} \quad \theta_{L_v} = f_{L_v}(h_i) \tag{7}
\]

\[
s_i = \text{concat}(t_u, t_v, L_v, L_u, L_e) \tag{9}
\]

Here, \( s_i \in \{0, 1\}^k \) is the concatenated component wise one-hot encoding of the real edge, and \( h_i \in \mathbb{R}^d \) is an LSTM hidden state vector that encodes the state of the graph generated so far. An embedding function \( f_{emb} \) is used to compress the sparse information possessed by the large edge representation \( (s_i) \) to a small vector of real numbers (Eq. 3). Given the graph state, \( h_i \), the output of the functions \( f_{t_u}(h_i), f_{t_v}(h_i), f_{L_u}(h_i), f_{L_v}(h_i) \) i.e. \( \theta_{t_u}, \theta_{t_v}, \theta_{L_u}, \theta_{L_v} \) respectively represent the multinomial distribution over possibilities of five components of the newly formed edge tuple. Finally, components of the newly formed edge tuple are sampled from the multinomial distributions parameterized by respective \( \theta \)s and concatenated to form the new edge \( (s_i) \) (Eq. 9).

With the above design, the key components that dictate the modeling quality are the multinomial probability distribution parameters \( \theta_{t_u}, \theta_{t_v}, \theta_{L_u}, \theta_{L_v} \) over each edge tuple \( t_u, t_v, L_u, L_v, L_e \). Our goal is, therefore, to ensure that the learned distributions are as close as possible to the real distributions in the training data. Using the broad mathematical architecture explained above, we will now explain the Graph Generation process followed by the sequence modeling / training process.
3.2.3 Graph Generation Given the learned state transition function \( f_{\text{trans}} \), embedding function \( f_{\text{emb}} \), and output functions \( f_{u}, f_{v}, f_{w}, f_{L}, f_{L'} \), Alg. 1 provides the pseudocode for the graph generation process. In Alg. 1, \( S \) stores the generated sequence and is initialized along with other variables (line 1-3). \( S \) represents the concatenation of the real edge sequence. This process of forming a new edge and appending it to sequence \( S \) is repeated until any of the five components is sampled as an EOS. Note, since each of the five components of \( S \) i.e., \( c, e, l, s, h \) are one-hot encodings and can be of different size, their EOS could be of different sizes, in which case they will be compared with their respective EOS to indicate the stopping condition.

3.2.4 Graph Modeling We now discuss how the required inference functions (Eqs. 3-9) are learned from a set of training graphs.

Given an input graph dataset \( G \), Alg. 2 first converts the graph dataset \( G \) to a sequence dataset \( S \) of minimum DFS codes (line 1). We initialize the unlearned neural functions by suitable weights (line 2). Further details about these functions are provided in Sec. 3.2.5. For each edge \( s_j \) in each sequence \( S \), a concatenated component-wise one-hot encoding of the real edge \( s_j \) is fed to the embedding function \( f_{\text{emb}} \), which compresses it and its compressed form along with previous graph state \( S \) is passed to the transition function \( f_{\text{trans}} \). \( \hat{s}_T \) generates the new graph state \( h_t \) (line 8). This new state \( h_t \) is passed to all the five output functions that provide us the corresponding probability distributions over the components of the new edge tuple (line 11). We concatenate these probability distributions to form \( \tilde{S} \in \mathbb{R}^k \) (same size as that of \( s_j \)), which is representative of a new edge (line 12). Finally, this component-wise concatenated probability distribution \( \tilde{S} \) is matched (component-wise) against the real tuple from the DFS code being learned.

This implies that we use ground truth rather than the model’s own prediction during successive time steps in training. In contrast to this, GraphGen uses its own predictions during graph generation time. The accuracy of the model is optimized using the cumulative binary cross-entropy loss function. Here \( s_i(c) \) represents component \( c \in \{tu, tv, lv, L, L'\} \), and \( log \) is taken element-wise on a vector.

### Algorithm 1: Graph generation algorithm

| Input: | LSTM-based state transition function \( f_{\text{trans}} \), embedding function \( f_{\text{emb}} \), five output functions \( f_u, f_v, f_w, f_L, f_{L'} \), empty graph state \( h_0 \), start token SOS, and end token EOS |
|------------------------|----------------|
| Output: | Graph \( G \) |
| 1 | \( S \leftarrow \emptyset \) |
| 2 | \( \hat{S}_0 \leftarrow \text{SOS} \) |
| 3 | \( i \leftarrow 0 \) |
| 4 | repeat |
| 5 | \( h_i = f_{\text{trans}}(h_{i-1}, f_{\text{emb}}(\hat{s}_{i-1})) \) |
| 6 | \( \tilde{S}_i \leftarrow \emptyset \) |
| 7 | for \( c \in \{tu, tv, lv, L, L'\} \) do |
| 8 | \( \theta_c \leftarrow f_c(h_i) \) // Sample \( \theta_c \) from Multinomial distribution parameterized by \( \hat{\theta}_c \) |
| 9 | \( \tilde{s}_i \leftarrow \text{concat} (\tilde{s}_i, c_i) \) |
| 10 | end |
| 11 | \( S \leftarrow S || (\tilde{S}_i) \) // Appending \( \tilde{S}_i \) to sequence \( S \) |
| 12 | until \( \exists c, c_i = \text{EOS} \) |
| 13 | return \( F^{-1}(S) \) // \( F^{-1} \) is mapping from minimum DFS code to graph |

### Table 2: Variables and their sizes

| Variable (one-hot vector) | Dimension |
|--------------------------|-----------|
| \( u, v \) | \( 1 \) |
| \( L_u, L_v \) | \( \max \sqrt{|V| + 1} \) |
| \( L_w \) | \( |E| + 1 \) |

\[
\text{BCE}(\tilde{s}_i, s_i) = -\sum_c \left( s_i[c]^T \log \tilde{s}_i[c] + (1 - s_i[c])^T \log(1 - \tilde{s}_i[c]) \right)
\]

(10)

3.2.5 Parameters State transition function \( f_{\text{trans}} \) in our architecture is a stacked LSTM-Cell, each of the five output functions \( f_u, f_v, f_w, f_L, f_{L'} \) are fully connected, independent Multi-layer Perceptrons (MLP) with dropouts and \( f_{\text{emb}} \) is a simple linear embedding function. The dimension of the one-hot encodings of \( u, v, e, L, L' \) is estimated from the training set. Specifically, the largest values for each element are computed in the first pass while computing the minimum DFS codes of the training graphs. Table 2 shows the dimensions of the one-hot encodings for each component of \( s_i \). Note that each variable has one size extra than their intuitive size to incorporate the EOS token. Since we set the dimensions of \( u, v, e \) to the largest training graph size, in terms of the number of nodes, the largest generated graph cannot exceed the largest training graph. Similarly, setting the dimensions of \( L_u, L_v, L_e \) to the number of unique node and edge labels in the training set means that our model is bound to produce only labels that have been observed at least once. Since the sizes of each component of \( s_i \) is fixed, the size of \( s_i \), which is formed by concatenating these components, is also fixed, and therefore, satisfies the pre-requisite condition for LSTMs to work, i.e., each input must be of the same size. We employ weight sharing among all time steps in LSTM to make the model scalable.

3.2.6 Complexity Analysis of Modeling and Inference. Consistent with the complexity analysis of existing models (Recall Table 1), we only analyze the complexity of the forward and backward propagations. The optimization step to learn weights is not included.

### Algorithm 2: Graph modeling algorithm

| Input: | Dataset of Graphs \( G = \{G_1, \ldots, G_n\} \) |
|------------------------|----------------|
| Output: | Learned functions \( f_{\text{trans}}, f_u, f_v, f_w, f_L, f_{L'}, f_{\text{emb}} \) and embedding function \( f_{\text{emb}} \) |
| 1 | \( G = \{f_{\text{trans}}, f_u, f_v, f_w, f_L, f_{L'}, f_{\text{emb}}\} \) |
| 2 | Initialize \( f_u, f_v, f_w, f_L, f_{L'}, f_{\text{emb}} \) |
| 3 | repeat |
| 4 | for \( j \) from 1 to \( m \) do |
| 5 | \( \tilde{S}_j \leftarrow \text{SOS} \); Initialize \( h_0 \) |
| 6 | \( \text{loss} \leftarrow 0 \) |
| 7 | for \( i \) from 1 to \( m \) do |
| 8 | \( h_i \leftarrow f_{\text{trans}}(h_{i-1}, f_{\text{emb}}(\tilde{S}_{i-1})) \) // \( \hat{s}_i \) will contain component-wise probability distribution vectors of \( \tilde{S}_i \) |
| 9 | \( \theta_c \leftarrow \text{softmax} (\hat{s}_i) \) |
| 10 | for \( c \in \{tu, tv, lv, L, L'\} \) do |
| 11 | \( \theta_c \leftarrow f_c(h_i) \) |
| 12 | \( \tilde{s}_i \leftarrow \text{concat} (\tilde{s}_i, \theta_c) \) |
| 13 | end |
| 14 | end |
| 15 | \( \text{loss} \leftarrow \text{loss} + \text{BCE}(\tilde{s}_i, s_i) \) |
| 16 | end |
| 17 | Back-propagate loss and update weights |
| 18 | until \( \text{stopping criteria} \) // Typically when validation loss is minimized |
All experiments are performed on a machine running dual Intel Xeon Gold 6142 processor with 16 physical cores each, having 1 Nvidia 1080 Ti GPU card with 12GB GPU memory, and 384 GB RAM with Ubuntu 16.04 operating system.

4.1 Experimental setup

All experiments are performed on a machine running dual Intel Xeon Gold 6142 processor with 16 physical cores each, having 1 Nvidia 1080 Ti GPU card with 12GB GPU memory, and 384 GB RAM with Ubuntu 16.04 operating system.

4.1.1 Datasets Table 3 lists the various datasets used for our empirical evaluation. The semantics of the datasets are as follows.

- **Chemical compounds:** The first five datasets are chemical compounds. We convert them to labeled graphs where nodes represent atoms, edges represent bonds, node labels denote the atom type, and edge labels encode the bond order such as single bond, double bond, etc.

- **Citation graphs:** Cora and Citeseer are citation networks; nodes correspond to publications and an edge represents one paper citing the other. Node labels represent the publication area.

- **Enzymes:** This dataset contains protein tertiary structures representing 600 enzymes from the BRENDA enzyme database[32]. Nodes in a graph (protein) represent secondary structure elements, and two nodes are connected if the corresponding elements are interacting. The node labels indicate the type of secondary structure, which is either helices, turns, or sheets. This is an interesting dataset since it contains only three label-types. We utilize this dataset to showcase the impact of graph invariants. Specifically, we supplemet the node labels with node degrees. For example, if a node has label “A” and degree 5, we alter the label to “5, A”.

4.1.2 Baselines We compare GraphGen with DeepGMG[19] and GraphRNN[40]. For GraphRNN, we use the original code released by the authors. Since this code does not support labels, we extend the model to incorporate node and edge labels based on the discussion provided in the paper[40].

4.1.3 Parameters and Training sets For both GraphRNN and DeepGMG, we use the parameters recommended in the respective papers. Estimation of M in GraphRNN and evaluation of DFS codes from graph database in GraphGen is done in parallel using 48 threads. For GraphGen, we use 4 layers of LSTM cells for $f_{trans}$ with hidden state dimension of 256 and the dimension of $f_{emb}$ is set to 92. Hidden layers of size 512 are used in MLPs for $f_u$, $f_v$, $f_{lu}$, $f_e$. We use Adam optimizer with a batch size of 32 for training. We use a dropout of $p = 0.2$ in MLP and LSTM layers. Furthermore, we use gradient clipping to remove exploding gradients and L2 regularizer to avoid over-fitting.

To evaluate the performance in any dataset, we split it into three parts: the train set, validation set, and test set. Unless specifically mentioned, the default split among training, validation and test is 80%, 10%, 10% of graphs respectively. We stop the training when validation loss is minimized or less than 0.05% change in validation loss is observed over an extended number of epochs. Note that both Citeseer and Cora represent a single graph. To form the training set in these datasets, we sample subgraphs by performing random walk with restarts from multiple nodes. More specifically, to sample a subgraph, we choose a node with probability proportional to its degree. Next, we initiate random walk with restarts with restart probability 0.15. The random walks stop after 150 iterations. Any edge that is sampled at least once during random walk with restarts is part of the sampled subgraph. This process is then repeated $X$ times to form a dataset of $X$ graphs. The sizes of the sampled subgraphs range from $1 ≤ |V| ≤ 102$ and $1 ≤ |E| ≤ 121$ in Citeseer and 9 ≤ |V| ≤ 111 and 20 ≤ |E| ≤ 124 in Cora.

4.1.4 Metrics The metrics used in the experiments can be classified into the following categories:

- **Structural metrics:** We use the metrics used by GraphRNN[40] for structural evaluation: (1) node degree distribution, (2) clustering coefficient distribution of nodes, and (3) orbit count distribution, which measures the number of all orbits with 4 nodes. Orbits capture higher-level motifs that are shared between generated and test graphs. The closer the distributions between the generated and test graphs, the better is the quality. In addition, we also compare the graph size of the generated graphs with the test graphs in terms of (4) average number of nodes and (5) average number of edges.

- **Label Accounting metrics:** We compare the distribution of (1) node labels, (2) edge labels, and (3) joint distribution of node labels and degree in generated and test graphs.

- **Graph Similarity:** To capture the similarity of generated graphs in a more holistic manner that considers both structure and labels, we use Neighbourhood Sub-graph Pairwise Distance Kernel (NSPDK)[6]. NSPDK measures the distance between two graphs by matching pairs of subgraphs with different radii and distances. The lower the NSPDK distance, the better is the performance. This quality metric is arguably the most important since it captures the global similarity instead of local individual properties.

---

Table 3: Summary of the datasets. \( \times \) means labels are absent.

| # | Name             | Domain | No. of graphs | \(|V|\) | \(|E|\) | \(|V|\) | \(|E|\)|
|---|------------------|--------|---------------|-------|-------|-------|-------|
| 1 | NCI-H21 [dang[23]] | Chemical | 24k          | [6, 30] | [6, 57] | [6, 57] | 11 3 |
| 2 | Yeast[23]      | Chemical | 47k          | [5, 50] | [5, 57] | [5, 57] | 11 3 |
| 3 | MOLT-4 (Leukemia[23]) | Chemical | 51k          | [6, 111] | [6, 116] | [6, 116] | 11 3 |
| 4 | MCF-7 (Breast[23])   | Chemical | 25k          | [6, 111] | [6, 116] | [6, 116] | 11 3 |
| 5 | ZINC[14]       | Chemical | 3.2M         | [3, 43] | [2, 49] | [2, 49] | 9 4 |
| 6 | Enzymes[4]   | Protein | 97k          | [2, 125] | [2, 149] | [2, 149] | 9 4 |
| 7 | Citeseer[13]  | Citation | 1            | 3312  | 4660  | 6 \( \times \) |
| 8 | Cora[33]       | Citation | 1            | 2798  | 3278  | 6 \( \times \) |
Table 4: Summary of the performance by GraphGen, GraphRNN, and DeepGMG. across 11 different metrics measuring various aspects of quality, robustness and efficiency on 6 datasets. The best performance achieved under each metric for a particular dataset is highlighted in bold font. Any value less than $10^{-3}$ is reported as $\approx 0$. The quality achieved by DeepGMG in Enzymes, Cora, and Citeseer are not reported since loss function for DeepGMG goes to NaN and hence fails to scale.

| Dataset        | Model | Degree | Clustering | Orbit | NSPK  | Avg # Nodes (Gen/Gold) | Avg # Edges (Gen/Gold) | Node Label | Edge Label | Joint Node Label & Degree | Node Label & Degree | Novelt y | Uniqueness | Training Time | # Epochs |
|----------------|-------|--------|------------|-------|-------|------------------------|-----------------------|------------|-----------|---------------------------|---------------------|----------|-------------|---------------|---------|
| Lung           | GraphGen | 0.005 | $\approx 0$ | $\approx 0$ | 0.035 | 32.55/36.68 | 36.67/37.52 | 0.001 | $\approx 0$ | $\approx 0$ | 0.103 | $\approx 100$ | 1h | 500 |
|                | GraphRNN | 0.103 | 0.301 | 0.043 | 0.325 | 6.32/35.88 | 6.35/37.65 | 0.191 | 0.005 | 0.836 | 86% | 45% | 1d | 1900 |
|                | DeepGMG | 0.123 | 0.001 | 0.026 | 0.260 | 11.04/35.88 | 10.26/37.65 | 0.003 | 0.002 | 0.862 | 98% | 98% | 2h | 20 |
| Yeast          | GraphGen | 0.006 | $\approx 0$ | $\approx 0$ | 0.026 | 35.89/32.11 | 35.18/32.32 | 0.001 | $\approx 0$ | $\approx 0$ | 0.030 | 95% | 95% | 1h | 20 |
|                | GraphRNN | 0.512 | 0.153 | 0.026 | 0.597 | 26.85/32.11 | 27.01/32.32 | 0.100 | 0.002 | 0.907 | 95% | 95% | 2h | 630 |
|                | DeepGMG | 0.056 | 0.064 | 0.002 | 0.239 | 34.59/32.11 | 34.05/32.32 | 0.143 | $\approx 0$ | 0.967 | 90% | 90% | 1d | 18 |
| Mixed: Lung + Leukemia + Yeast + Breast | GraphGen | 0.005 | $\approx 0$ | $\approx 0$ | 0.023 | 37.85/37.6 | 39.24/37.64 | 0.001 | $\approx 0$ | $\approx 0$ | 0.100 | 99% | 99% | 11h | 350 |
| Enzymes        | GraphGen | 0.241 | $\approx 0$ | 0.039 | 0.331 | 8.19/37.6 | 7.29/31.44 | 0.102 | 0.010 | 0.879 | 62% | 52% | 1d | 400 |
|                | GraphRNN | 0.090 | 0.13 | 0.038 | 0.607 | 11.87/32.95 | 23.52/34.15 | 0.048 | - | 0.512 | 99% | 97% | 1d | 2000 |
| Citeseer       | GraphGen | 0.009 | 0.083 | 0.100 | 0.020 | 35.90/38.56 | 41.54/37.14 | 0.024 | - | 0.852 | 85% | 95% | 4h | 400 |
|                | GraphRNN | 1.321 | 0.665 | 1.006 | 0.052 | 31.42/48.56 | 77.13/39.14 | 0.063 | - | 0.055 | 62% | 100% | 1d | 1450 |
| Cora           | GraphGen | 0.061 | 0.117 | 0.009 | 0.122 | 48.86/38.39 | 58.62/46.61 | 0.017 | - | 0.0113 | 91% | 98% | 4h | 400 |
|                | GraphRNN | 1.125 | 1.062 | 0.047 | 0.991 | 54.84/58.39 | 226.46/46.61 | 0.085 | - | 0.0113 | 76% | 91% | 9h | 400 |

- **Redundancy checks:** Consider a generative model that generates the exact same graphs as it saw in the training set. Although this model will perform very well in the above metrics, such a model is practically useless. Ideally, we would want the generated graphs to be diverse and similar, but not identical. To quantify this aspect, we check (1) **Novelty**, which measures the percentage of generated graphs that are not subgraphs of the training graphs and vice versa. Note that identical graphs are subgraph isomorphic to each other. In other words, novelty checks if the model has learned to generalize unseen graphs. We also compute (2) **Uniqueness**, which captures the diversity in generated graphs. To compute Uniqueness, we first remove the generated graphs that are subgraph isomorphic to some other generated graphs. The percentage of graphs remaining after this operation is defined as Uniqueness. For example, if the model generates 100 graphs, all of which are identical, the uniqueness is 1/100 = 1%.

To compute the distance between two distributions, like in GraphRNN[40], we use *Maximum Mean Discrepancy (MMD)[11]. To quantify quality using a particular metric, we generate 2560 graphs and compare them with a random sample of 2560 test graphs. On all datasets except Enzymes, we report the average of 10 runs of computing metric, comparing 256 graphs in a single run. Since the Enzymes dataset contains only 575 graphs, we sample 64 graphs randomly from the test set 40 times and report the average.

4.2 Quality

Table 4 presents the quality achieved by all three benchmarked techniques across 10 different metrics on 6 datasets. We highlight the key observations that emerge from these experiments. Note that, some of the generated graphs may not adhere to the structural properties assumed in §2 i.e., no self loops, multiples edges or disconnected components, so we prune all the self edges and take the maximum connected component for each generated graph.

**Graph and Sub-graph level similarity:** On the NSPDK metric, which is the most important due to capturing the global similarity of generated graphs with test graphs, GraphGen is significantly better across all datasets. This same trend also transfers to orbit count distributions. Orbit count captures the presence of motifs also seen in the test set. These two metrics combined clearly establishes that GraphGen model graphs better than GraphRNN and DeepGMG.

**Node-level metrics:** Among the other metrics, GraphGen remains the dominant performer. To be more precise, GraphGen is consistently the best in both Node and Edge Label preservation, as well as graph size in terms of the number of edges. It even performs the best on the joint Node Label and Degree metric, indicating its superiority in capturing structural and semantic information together. The performance of GraphGen is comparatively less impressive in the Enzymes dataset, where GraphRNN marginally outperforms in the Degree and Clustering Coefficient metrics. Nonetheless, even in Enzyme, among the eight metrics, GraphGen is better in six. This result also highlights the need to not rely only on node-level metrics. Specifically, although GraphRNN models the node degrees and clustering coefficients well, it generates graphs that are much smaller in size and hence the other metrics, including NSPDK, suffers.

**Novelty and Uniqueness:** Across all datasets, GraphGen has uniqueness of at least 98%, which means it does not generate the same graph multiple times. In contrast, GraphRNN has a significantly lower uniqueness in chemical compound datasets. In Lung, only 45% of the generated graphs are unique. In several of the datasets, GraphRNN also has low novelty, indicating it regenerates graphs (or subgraphs) it saw during training. For example, in Cora, Citeseer and, the mixed chemical dataset, at least 30% of the generated graphs are regenerated from the training set. While we cannot pinpoint the reason behind this performance, training on random graph sequence representations could be a factor. More specifically, even though the generative model may generate new sequences, they may correspond to the same graph. While this is possible in GraphGen as well, the likelihood is much less as it is trained on DFS codes that enable one-to-one mapping with graphs.

**Analysis of GraphRNN and DeepGMG:** Among GraphRNN and DeepGMG, DeepGMG generally performs better in most metrics. However, DeepGMG is extremely slow and fails to scale on larger networks due to $O(|V||E|^2)$ complexity. GraphRNN’s major weakness is in learning graph sizes. As visible in Table 4, GraphRNN consistently generates graphs that are much smaller than the test graphs. We also observe that this issue
arises only in labeled graphs. If the labels are removed while retaining their structures, GraphRNN generates graphs that correctly mirror the test graph size distribution. This result clearly highlights that labels introduce an additional layer of complexity that simple extensions of models built for unlabeled graphs do not solve.

4.2.1 Visual Inspection To gain visual feedback on the quality of the generated graphs, in Fig. 4, we present a random sample of 9 graphs from the training set of Lung dataset and those generated by each of the techniques. Visually, GraphGen looks the most similar, while GraphRNN is the most dissimilar. This result is consistent with the quantitative results obtained in Table 4. For example, consistent with our earlier observations, the GraphRNN graphs are of similar sizes and structures. In contrast, GraphGen’s graphs are of similar sizes and structures.

In Fig. 5, we perform the same exercise as above with 6 randomly picked graphs from the Cora dataset and those generated by GraphGen and GraphRNN. Since the graphs in this dataset are much larger, it is hard to comment if GraphGen is better than GraphRNN through visual inspection; the layout of a graph may bias our minds. However, we notice one aspect where GraphGen performs well. Since Cora is a citation network, densely connected nodes (communities) have a high affinity towards working in the same publication area (node label/color). This correlation of label homogeneity with communities is also visible in GraphGen(Fig. 5(b)).

4.3 Scalability

After establishing the clear superiority of GraphGen in quality for labeled graph generative modeling, we turn our focus to scalability. To enable modeling of large graph databases, the training must complete within a reasonable time span. The penultimate column in Table 4 sheds light on the scalability of the three techniques. As clearly visible, GraphGen is 2 to 5 times faster than GraphRNN, depending on the dataset. DeepGMG is clearly the slowest, which is consistent with its theoretical complexity of $O(|V||E|^2)$. If we exclude the sequence generation aspect, GraphRNN has a lower time complexity for training than GraphGen; while GraphRNN is linear to the number of nodes in the graph, GraphGen is linear to the number of edges. However, GraphGen is still able to achieve better performance due to training on canonical labels of graphs. In contrast, GraphRNN generates a random sequence representation for a graph in each epoch. Consequently, as shown in the last column of Table 4, GraphRNN runs for a far larger number of epochs to achieve loss minimization in the validation set. Note that DeepGMG converges within the minimum number of epochs. However, the time per epoch of DeepGMG is 100 to 200 times higher than GraphGen. Compared to GraphRNN, the time per epoch of GraphGen is $\approx 30\%$ faster on average.

4.3.1 Impact of Training Set Size To gain a deeper understanding of the scalability of GraphGen, we next measure the impact of training set size on performance. Figs. 7(a)-7(b) present the growth of training times on ZINC and Cora against the number of graphs in the training set. While ZINC contains 3.2 million graphs, in Cora, we sample up to 1.2 million subgraphs from the original network for the training set. As visible, both GraphRNN and DeepGMG has a much steeper growth rate than GraphGen. In fact, both these techniques fail to finish within 3 days for datasets exceeding 100,000 graphs. In contrast, GraphGen finishes within two and a half days, even on a training set exceeding 3 million graphs. The training times of all techniques are higher in Cora since the graphs in this dataset are much larger in size. We also observe that the growth rate of GraphGen slows at larger training sizes. On investigating further, we observe that with higher training set sizes, the number of epochs required to reach the validation loss minima reduces. Consequently, we observe the trend visible in Figs. 7(a)-7(b). This result is not surprising since the learning per epoch is higher with larger training sets.

An obvious question arises at this juncture: Does lack of scalability to larger training sets hurt the quality of GraphRNN and DeepGMG? Figs. 7(c)-7(e) answer this question. In both ZINC and Cora, we see a steep improvement in the quality (reduction in NSPDK and Orbit) of GraphRNN as the training size is increased from $\approx 10,000$ graphs to $\approx 50,000$ graphs. The improvement rate of GraphGen is relatively milder, which indicates that GraphRNN has a greater need for larger training sets. However, this requirement is not met as GraphRNN fails to finish within 3 days, even for 100,000 graphs. Overall, this result establishes that scalability is not only important for higher productivity but also improved modeling.

4.3.2 Impact of Graph Size We next evaluate how the size of the graph itself affects the quality and training time. Towards that end, we partition graphs in a dataset into multiple buckets based on its size. Next, we train and test on each of these buckets individually and measure the impact on quality and efficiency. Figs. 7(f) and 7(h) present the impact on quality in chemical compounds (same as Mixed dataset in Table 4) and Cora respectively. In both datasets, each bucket contains exactly 10,000 graphs. The size of graphs in each bucket is however different, as shown in the $x$-axis of Figs. 7(f) and 7(h). Note that Cora contains larger graphs since the network itself is much larger than chemical compounds. The result for DeepGMG is missing on the larger graph-size buckets since it
fails to model large graphs. GraphRNN runs out of GPU memory (12GB) in the largest bucket size in Cora.

As visible, there is a clear drop in quality (increase in NSPDK) as the graph sizes grow. This result is expected since larger graphs involve larger output spaces, and the larger the output space, the more complex is the modeling task. Furthermore, both GraphGen and GraphRNN convert graph modeling into sequence modeling. It is well known from the literature that auto-regressive neural models struggle to learn long-term dependencies in sequences. When the sequence lengths are large, this limitation impacts the quality. Nonetheless, GraphGen has a more gradual decrease in quality compared to GraphRNN and DeepGMG.

Next, we analyze the impact of graph size on the training time. Theoretically, we expect the training time to grow, and this pattern is reflected in Figs. 7(g) and 7(i). Note that the y-axis is in log scale in these plots. Consistent with previous results, GraphGen is faster. In addition to the overall running time, we also show the average time taken by GraphGen in computing the minimum DFS code. Recall that the worst-case complexity of this task is factorial to the number of nodes in the graph. However, we earlier argued that in practice, the running times are much smaller. This experiment adds credence to this claim by showing that DFS code generation is a small portion of the total running time. For example, in the largest size bucket of chemical compounds, DFS code generation of all 10,000 graphs take 30 seconds out of the total training time of 10 hours. GraphRNN has a similar pre-processing component as well (GraphRNN-M), where it generates a large number of BFS sequences to estimate a hyper-parameter. We find that this pre-processing time, although of polynomial-time complexity, is slower than minimum DFS code computation in chemical compounds. Overall, the results of this experiment support the proposed strategy of feeding canonical labels to the neural model instead of random sequence representations.

### 4.4 Impact of Invariants

Minimum DFS code generation relies on the node and edge labels to prune the search space and identify the lexicographically smallest code quickly. When graphs are unlabeled or have very few labels, this process may become more expensive. In such situations, labels based on vertex invariants allow us to augment existing labels. To showcase this aspect, we use the Enzymes dataset, which has only 3 node labels and no edge labels. We study the training time and the impact on quality based on the vertex invariants used to label nodes. We study four combinations: (1) existing node labels only, (2) node label + node degree, (3) node label + clustering coefficient (CC), and (4) label + degree + CC. Fig. 6 presents the results. As visible, there is a steady decrease in the DFS code generation time as more invariants are used. No such trend, however, is visible in the total training time as additional features may both increase or decrease the time taken by the optimizer to find the validation loss minima. From the quality perspective, both NSPDK and Orbit show improvement (reduction) through the use of invariants. Between degree and CC, degree provides slightly improved results. Overall, this experiment shows that invariants are useful in both improving the quality and reducing the cost of minimum DFS code generation.

### 4.5 Alternatives to LSTMs

For deep, auto-regressive modeling of sequences, we design an LSTM tailored for DFS codes. Can other neural architectures be used instead? Since a large volume of work has been done in the NLP community on sentence modeling, can those algorithms be used by treating each edge tuple as a word? To answer these questions, we replace LSTM with SentenceVAE[5] and measure the performance. We present the results only in Lung and Cora due to space constraints; a similar trend is observed in other datasets as well. In Table 5, we present the two metrics that best summarizes the impact of this experiment. As clearly visible, SentenceVAE introduces a
huge increase in NSPDK distance, which means inferior quality. An even more significant result is observed in Uniqueness, which is close to 0 for SentenceVAE. This means the same structure is repeatedly generated by SentenceVAE. In DFS codes, every tuple is unique due to the timestamps, and hence sentence modeling techniques that rely on repetition of words and phrases, fail to perform well. While this result does not say that LSTMs are the only choice, it does indicate the need for architectures that are custom-made for DFS codes.

5 Concluding Insights

In this paper, we studied the problem of learning generative models for labeled graphs in a domain-agnostic and scalable manner. There are two key takeaways from the conducted study. First, existing techniques model graphs either through their adjacency matrices or sequence representations. In both cases, the mapping from a graph to its representation is one-to-many. Existing models feed multiple representations of the same graph to the model and rely on the model overcoming this information redundancy. In contrast, we construct canonical labels of graphs in the form of minimum invariants, DFS-code construction is a small component of the overall training time (Figs. 7(g) and 7(i)). Consequently, the time complexity of graph canonization is not a practical concern and, feeding more precise graph representations to the training model is a better approach in terms of quality and scalability.

The second takeaway is the importance of quality metrics. Evaluating graph quality is a complicated task due to the multi-faceted nature of labeled structures. It is, therefore, important to deploy enough metrics that cover all aspects of graphs such as local node-level properties, structural properties in the form of motifs and global similarity, graph sizes, node and edge labels, and redundancy in the generated graphs. A learned model is effective only if it performs well across all of the metrics. As shown in this study, GraphGen satisfies this criteria.

The conducted study does not solve all problems related to graph modeling. Several graphs today are annotated with feature vectors. Examples include property graphs and knowledge graphs. Can we extend label modeling to feature vectors? All of the existing learning-based techniques, including GraphGen, cannot scale to graphs containing millions of nodes. Can we overcome this limitation? We plan to explore these questions further by extending the platform provided by GraphGen.

Acknowledgments

The authors thank Lovish Madaan for helpful discussions and comments on the paper. The authors also thank IIT Delhi HPC facility for the computational resources.
