Graph Classification with Minimum DFS Code: Improving Graph Neural Network Expressivity

Jhalak Gupta
Indian Institute of Technology Indore, India
cse170001024@iiti.ac.in

Arijit Khan
Nanyang Technological University, Singapore
arijit.khan@ntu.edu.sg

***Abstract*—Graph neural networks (GNNs) generally follow a recursive neighbors aggregation scheme. Recent GNNs are not powerful than the 1-Weisfeiler Lehman test, which is a necessary but insufficient condition for graph isomorphism, hence limiting their abilities to utilize graph structures properly. Moreover, deep GNNs with many convolutional layers suffer from over-smoothing, thus cannot capture long-range dependencies. As a result, downstream applications, such as graph classification, are impacted. To this end, we design GNNs on top of the minimum DFS code, which is a canonical form of a graph, and being *injective* it captures the graph structure precisely along with node and edge labels. Due to the sequential structure of the minimum DFS code, we employ state-of-the-art RNNs (LSTM, BiLSTM, GRU) and Transformer-based sequence classification techniques. While one can compute the minimum DFS code efficiently in practice, LSTM, BiLSTM, GRU, and Transformers capture long-term dependencies in arbitrary length sequences. We also consider a novel variant of the minimum DFS code, which is not injective, but it reduces the complexity of the feature space, increases generalizability, and also improves the classification performance over many real-world graph datasets. Our thorough empirical comparisons with six real-world network datasets demonstrate the accuracy and efficiency of our methods. We have open-sourced our solution framework [17] in which one can plug in different graph datasets and get their classification results. This will benefit researchers and practitioners, biologists, social scientists, and data scientists, among others.**

I. Introduction

Graph data are pervasive in many domains, e.g., social networks, knowledge graphs, road networks, computer vision, software engineering, and natural language processing. Developing machine learning tools for classifying networks can be observed in cheminformatics [46], [11] and bioinformatics [38], malware detection [13], telecommunication networks, internet-of-things [10], trajectories and social networks [19].

Given a set of graphs with different structures and sizes, the graph classification problem predicts the class labels of unseen graphs [11], [51], [52]. This is challenging because network data contain graphs with different numbers of nodes and edges, and a generic node order is often not available. Graphs do not have regular grid structures, since the neighborhood size of each node differs. The lack of ordered vector representation complicates machine learning on graphs, and makes it difficult to build a classifier over the graph space [52], [51].

In the past, graph kernels (e.g., random walks, shortest paths, subtrees, graphlets, and subgraph kernels) [43], [14], [39] and structural feature (e.g., frequent and significant subgraphs) [48], [35], [27] based classification methods were developed. Learning task-relevant graph features were considered in [11], [31]. Attention-based graph classification was proposed in [23] to focus on small but informative parts of the graph. Due to the challenges of feature engineering, together with the hardness of subgraph mining and isomorphism testing, deep learning methods have become popular.

The concept of graph neural network (GNN) was first proposed in [40], [36], which extended neural networks for processing of graph data. These early works are called graph recurrent neural networks (GraphRNN): They learn a node's representation by propagating neighbor information in an iterative manner until a stable fixed point is reached. Encouraged by the success of convolutional neural networks (CNNs) in computer vision, their end-to-end learning and ability to extract localized spatial features, a large number of graph convolutional neural networks (GCNs) [22], [25], [52], [51], [12] are recently developed. GCNs follow a recursive neighbors aggregation (or message passing) model. As one layer in the GCN aggregates its 1-hop neighbors, after k rounds of aggregation, a node is represented by its feature vector that captures the structural information in its k-hop neighborhood. The graph feature is derived by a readout function or pooling on node features.

It is proven in [47], [30] that recent GNN variants are not powerful than the 1-WL (Weisfeiler Lehman) test [45], [29] in distinguishing non-isomorphic graphs, where the WL test is a *necessary but insufficient* condition for graph isomorphism [7], thus limiting their abilities to adequately exploit graph structures. Our work is motivated by the following question: Can we improve the discriminative/representational power of GNNs than the existing GCNs? To this end, we employ the *minimum DFS code* [49], that is a canonical form of a graph, and captures the graph structure precisely along with node and edge labels. The canonical form of a graph is a sequence – two graphs are isomorphic iff they have the same canonical form. Minimum DFS code encodes a graph into a unique edge sequence by doing a depth-first search (DFS). Constructing the minimum DFS code is equivalent to solving graph isomorphism, and it has the worst-case computation cost of $\mathcal{O}(n!)$, n being the number of nodes. However, real-world graphs are sparse and have node and edge labels, which prune the search space significantly, thus we can efficiently compute

978-1-6654-3902-2/21/$31.00 ©2021 IEEE

the minimum DFS code in practice [34], [49], [15], [26].

Due to the sequential nature of the minimun DFS code, we employ state-of-the-art RNN-based sequence classification techniques (*Long short-term memory* (LSTM) [20], *Bidirectional LSTM* (BiLSTM) [37], and *Gated recurrent unit* (GRU) [8]) and Transformers [42] for downstream graph classification. Our RNN and Transformer-based sequence classification over minimum DFS codes has following benefits compared to existing GCNs, which employ adjacency matrix-based graph representations (e.g., graph Laplacian and degree matrices). **(i)** LSTM, BiLSTM, GRU, and Transformers can captures long-term dependencies in arbitrary-length sequences, and have achieved great success in a wide range of sequence learning tasks including language modeling and speech recognition [44]. On the other hand, a deep GCN with many convolutional layers suffers from the *over-smoothing* problem, and generally cannot capture long-range interactions [2], [24]. **(ii)** The minimum DFS code is unique for all isomorphic graphs, whereas there are $\mathcal{O}(n!)$ mappings from a graph to its adjacency matrix, therefore GCN designs must be node permutation-invariant.

To further improve the generalization capacity of our solution, we consider a novel variant of the minimum DFS code, which is not injective, but it significantly improves the classification quality as shown in our experiments.

Our contributions and roadmap. The main contributions of this paper are as follows:

- To the best of our knowledge, ours is the first work that employs the minimum DFS code for graphs classification. The minimum DFS code, being injective, aims at improving the expressivity of GNNs.
- We employ state-of-the-art RNN-based sequence classification techniques (LSTM, BiLSTM, GRU) and the encoder layer of Transformer over minimum DFS codes for graph classification. While minimum DFS codes, LSTM, BiLSTM, GRU, and Transformers are well-established techniques, our technical novelty lies in properly integrating them for graph classification. Our method can capture long-range dependencies in the graph space. We also consider a novel variant of the minimum DFS code, which is not injective, but it significantly improves the classification accuracy over many networks (§III).
- We present thorough experimental comparisons over six real-world graph datasets from different categories. We measure the accuracy and efficiency of our methods, and demonstrate that our accuracy results are significantly higher than those of existing GCNs, based on several metrics (§IV).

II. Background on Expressivity of Graph Neural Networks

We start with a few definitions. An undirected labeled graph G is defined as a triple $G = (V, E, L)$ where V is the set of nodes, $E \subseteq V \times V$ is the set of undirected edges, and L is a labeling function that maps a node or an edge to a label. For a node v, its neighbors are denoted by $N(v)$.

Graph Isomorphism. Given two graphs $G = (V, E, L)$ and $Q = (V', E', L')$, a graph isomorphism is a *bijective function* $M : V' \to V$ such that (1) $\forall v \in V', L'(v) = L(M(v))$, and (2) $\forall (v_1, v_2) \in E', (M(v_1), M(v_2)) \in E$, and $L'(v_1, v_2) = L(M(v_1), M(v_2))$.

Graph isomorphism is not known to be in polynomial or **NP**-complete. Recently, Babai prove that graph isomorphism is solvable in quasipolynomial time: on n-node input graphs, Babai's algorithm runs in time $n^{p(\log n)}$ for some polynomial $p()$ [3], [16].

Weisfeiler-Lehman Algorithm. We next discuss the 1-WL algorithm, which is a *necessary but insufficient* condition for graph isomorphism over node-labeled graphs (edge labels are not considered). In each iteration $t \geq 0$, 1-WL computes a node coloring $c_L^{(t)} : V \to \Sigma$, which depends on the coloring from the earlier round. In iteration 0, we set $c_{L(v)}^{(0)} = L(v)$. In iteration $t > 0$, we compute:

$$c_{L(v)}^{(t)} = HASH\left(\left(c_{L(v)}^{(t-1)}, \{\{c_{L(u)}^{(t-1)} | u \in N(v)\}\}\right)\right) \quad (1)$$

$\{\{\ldots\}\}$ represents a multiset and $HASH$ maps bijectively the above pair to a unique color in Σ, which was not used in previous rounds. If the number of colors between two rounds does not change, i.e., the cardinalities of the images of $c_L^{(t)}$ and $c_L^{(t+1)}$ remain the same, a stable coloring has reached and 1-WL terminates. Termination (i.e., stable coloring) is guaranteed after at most $|V|$ iterations. To test if two graphs G and Q are isomorphic, we run the 1-WL algorithm parallelly on both graphs. If the two graphs have a different number of nodes colored $l \in \Sigma$ at some iteration, 1-WL decides that the graphs are non-isomorphic. Even though 1-WL cannot distinguish all non-isomorphic graphs, it can correctly test isomorphism for a wide class of graphs [4].

GCNs and Expressivity. Graph convolutional neural networks (GCNs) generally apply recursive neighborhood aggregation (or message passing). This operation consists of AGGREGATE and COMBINE functions [47]:

$$a_v^{(k)} = AGGREGATE^{(k)}\left(\{p_u^{(k-1)} : u \in N(v)\}\right)$$
$$p_v^{(k)} = COMBINE^{(k)}\left(p_v^{(k-1)}, a_v^{(k)}\right) \quad (2)$$

Here, $p_v^{(k)}$ denotes the k-th layer feature vector at the v-th node. The AGGREGATE function aggregates features of neighboring nodes to derive the feature vector $a_v^{(k)}$ for layer k. The COMBINE function combines the previous node feature $p_v^{(k-1)}$ with aggregated node features $a_v^{(k)}$ to output the node feature $p_v^{(k)}$ of the k-th layer. Finally, the READOUT function aggregates node features from the final layer to compute the entire graph's representation p_G.

$$p_G = READOUT(p_v^{(k)} | v \in V) \quad (3)$$

It has been proven in [47], [30] that the aforementioned GCN architecture does not have more power in terms of distinguishing between non-isomorphic graphs than the 1-WL test. Xu et al. [47] further show that if aggregation and readout functions

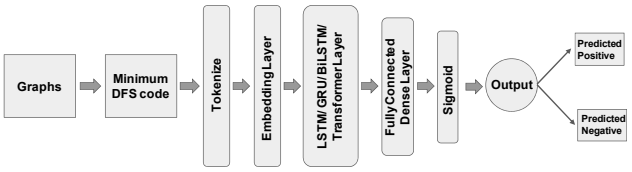
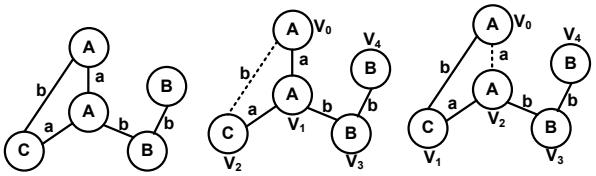


Fig. 1. Minimum DFS code-based graph classification



(a) Input graph (b) Two DFS codes

Fig. 2. DFS code construction

of a GCN are injective, as in the *graph isomorphism network* (GIN), then its discriminative/representational power is equal to the power of 1-WL test. In particular, GIN implements the aggregate and combine functions as follows:

$$p_v^{(k)} = MLP^{(k)}\left((1+\epsilon^{(k)})p_v^{(k-1)} + \sum_{u\in N(v)} p_u^{(k-1)}\right) \quad (4)$$

where $\epsilon^{(k)}$ is a learnable parameter, and MLP is a multi-layer perceptron with non-linearity. For readout, the embedded node features of all layers are added and then concatenated to derive the final graph feature p_G.

$$p_G^{(k)} = sum\left(p_0^{(k)}, p_1^{(k)}, \ldots, p_N^{(k)}\right)$$
$$p_G = concatenate\left(\{p_G^{(k)}\}|k = 0, 1, \ldots, K\right) \quad (5)$$

As stated earlier, 1-WL test is a *necessary but insufficient* condition for graph isomorphism over node-labeled graphs.

GCNs work with adjacency matrix-based graph representations, including graph Laplacian and degree matrices. Other disadvantages of GCNs are as follows. **(i)** Deep GCNs with many convolutional layers suffer from *over-smoothing*, and are unable to capture long-range interactions [2], [24]. **(ii)** There are $\mathcal{O}(n!)$ mappings from a graph to its adjacency matrix, therefore GCN designs must be node permutation-invariant.

To overcome the aforementioned challenges with existing GCNs, in this work we design an orthogonal and novel framework for RNN and Transformer-based graph classification via minimum DFS codes.

III. Minimum DFS Code Based Graph Classification

We describe the minimum DFS code in § III-A and our classification approach in § III-B. Our framework is presented in Figure 1. In § III-C, we additionally consider a novel variant of the minimum DFS code, which further improves the classification accuracy.

A. Minimum DFS Code

The minimum DFS code [49] is a canonical form of a graph that captures the graph structure precisely along with node and edge labels. We represent an edge $e = (u, v)$ with a 5-tuple: $(u, v, L(u), L(e), L(v))$; u and v are the two corresponding node ids (based on time-stamps as discussed below) with node labels $L(u)$ and $L(v)$, respectively, and $L(e)$ is the edge label. A list of 5-tuples for all edges of a graph is referred to as a code. We construct a *DFS code* [49] by performing a depth-first search (DFS), starting from any node in the graph. During the DFS traversal, we assign to each node an id which is based on the time-stamp when it is discovered (that is, the time when the node is first visited during traversal). Therefore, the starting node is assigned an id 0. If an edge (u, v) is in the DFS traversal, then $u < v$, where u, v are assigned node ids based on time-stamps, and we refer to such an edge a *forward edge*. Edges that are not part of the DFS traversal, are called *backward edges*.

Given a DFS traversal, the next step is to define a total order of all the edges in the graph. Forward edges appear in their natural order based on when they are traversed during the DFS. A backward edge from a node must appear before any forward edge from that node. In fact, any backward edge from a node must appear right after the node is introduced in the DFS code. Among the backward edges from the same node u of the form (u, v) and (u, v'), (u, v) has a higher order if $v < v'$, where u, v, v' are assigned node ids based on time-stamps.

Example 1. *Consider an input graph in Figure 2(a). The left one in Figure 2(b) shows a DFS code:* $\langle(V_0, V_1, A, a, A)(V_1, V_2, A, a, C)(V_2, V_0, C, b, A)(V_1, V_3, A, b, B)(V_3, V_4, B, b, B)\rangle$. *The backward edge is shown as a dotted edge, while the forward edges are solid edges. The right one in Figure 2(b) presents another DFS traversal, the DFS code is* $\langle(V_0, V_1, A, b, C)(V_1, V_2, C, a, A)(V_2, V_0, A, a, A)(V_2, V_3, A, b, B)(V_3, V_4, B, b, B)\rangle$.

Since each graph may have multiple DFS traversals, we follow the order defined in [49] to compare pairs of code lexicographically. The lexicographic order is a linear order defined as follows. If $A = (a_0, a_1, \ldots, a_m)$ and $B = (b_0, b_1, \ldots, b_n)$ are the codes, then $A \leq B$ iff either of the following is true.

$$(1)\, \exists t, 0 \leq t \leq \min(m, n), \forall k < t, a_k = b_k, a_t <_e b_t,$$
$$(2)\, \forall 0 \leq k \leq m, a_k = b_k, \text{and } n \geq m.$$

Assume the forward edge set and the backward edge set for A and B are $E_{A,f}$, $E_{A,b}$, $E_{B,f}$, and $E_{B,b}$, respectively. $a_t = (u_a, v_a, L(u_a), L(a_i), L\ (v_a)) <_e b_t = (u_b, v_b, L(u_b), L(b_j), L(v_b))$ iff one of the following is true.

$(1)\, a_t \in E_{A,b}$ *and* $b_t \in E_{B,f}$,
$(2)\, a_t \in E_{A,b}$ *and* $b_t \in E_{B,b}$ *and* $v_a < v_b$,
$(3)\, a_t \in E_{A,b}$ *and* $b_t \in E_{B,b}$ *and* $v_a = v_b$ *and* $L(a_i) < L(b_j)$,
$(4)\, a_t \in E_{A,f}$ *and* $b_t \in E_{B,f}$ *and* $u_b < u_a$,
$(5)\, a_t \in E_{A,f}$ *and* $b_t \in E_{B,f}$ *and* $u_b = u_a$, *and* $L(u_a) < L(u_b)$,
$(6)\, a_t \in E_{A,f}$ *and* $b_t \in E_{B,f}$ *and* $u_b = u_a$, *and* $L(u_a) = L(u_b)$ *and* $L(a_i) < L(b_j)$,

(7) $a_t \in E_{A,f}$ *and* $b_t \in E_{B,f}$ *and* $u_b = u_a$, *and* $L(u_a) = L(u_b)$, *and* $L(a_i) = L(b_j)$, *and* $L(v_a) < L(v_b)$.

The minimum DFS code is the one with the minimum lexicographic order. Finally, each graph can be represented by the corresponding minimum DFS code, and vice versa.

Theorem 1. *[49] Given two graphs G and Q, G is isomorphic to Q if and only if they have the identical minimum DFS code.*

Minimum DFS Code Computation. The algorithm to compute the minimum DFS code consists of four major steps. **(1)** Select the nodes with the minimum label as the candidate roots. **(2)** Construct the DFS spanning tree from each root, always visit edges with smaller labels first, and if two edges have the same label, then visit the ones whose second node has smaller label. **(3)** Insert the backward edges to complete the construction of DFS codes. **(4)** Among the DFS codes constructed as above, pick the lexicographically minimum one.

The worst-case computation cost for the minimum DFS code is $\mathcal{O}(n!)$, n being the number of nodes. However, steps 1-2 prunes the search space significantly, and thereby we can efficiently compute the minimum DFS code in practice.

Example 2. *For the graph in Figure 2(a), the minimum DFS code is* $\langle (V_0, V_1, A, a, A)(V_1, V_2, A, a, C)\ (V_2, V_0, C, b, A)(V_1, V_3, A, b, B)(V_3, V_4, B, b, B)\rangle$, *corresponding to the DFS traversal in Figure 2(b) (left). Computing this minimum DFS code is not expensive. The only possible starting nodes for the minimum DFS code are the two nodes with label A, which is lexicographically the smallest. Moreover, their one-hop neighbors are sufficient to decide which one will be the starting node for the minimum DFS code. Clearly,* $(V_0, V_1, A, a, A)(V_1, V_2, A, a, C)$ *must be the first two edges in the minimum DFS code as they are lexicographically the smallest.*

B. RNN and Transformer-based Classification

We employ RNN and Transformer-based sequence classification models over minimum DFS codes. RNNs are powerful tools for modeling sequential data such as time series and sentences. An RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen earlier. For instance, each minimum DFS code from a graph can be considered as a "sequence", and every 5-tuple (denoting an edge) in it as a "word". We instantiate our vocabulary with all seen words (i.e., all distinct 5-tuples present in the dataset), then we convert each unique word in our sequences into a unique integer using the vocabulary (that is, we "tokenize" each distinct 5-tuple into a unique integer or token), and finally we embed each integer into a 32-dimensional vector using an embedding layer [53]. We process the sequence of vectors using an RNN/ Transformer layer. Therefore, *the novelty of our work is to employ the minimum DFS code (which is injective), coupled with the RNN/ Transformer, for graph classification that results in higher accuracy than existing GCNs. Later we also consider a novel variant of the minimum DFS code, which is not injective, but it significantly improves the classification accuracy, as demonstrated in our experimental results*. The result of the RNN/ Transformer layer is passed through a dense layer with one hidden unit and finally a sigmoid function for classification. We train our network in Figure 1 using the binary cross-entropy loss function along with the ADAM optimizer.

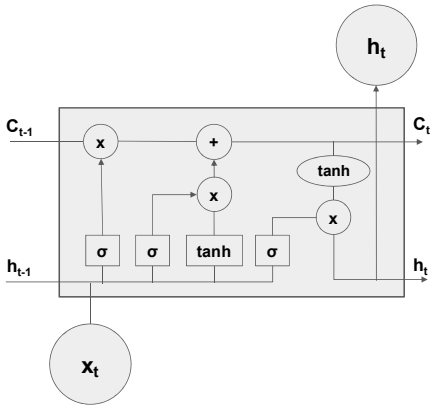


Fig. 3. Architecture of an LSTM cell

Standard RNNs suffer from the *vanishing gradient* problem [20], which makes learning over long data sequences difficult. More recent RNNs, including LSTM [20], BiLSTM [37], GRU [8] solve this problem, and they are capable of learning long-term dependencies. We apply them for classification of minimum DFS codes. As an alternative to RNNs, we also consider the encoder layer of Transformers [42]. A Transformer model attends all words in the sequence and uses the attention mechanism, thus it has extremely long-term memory. We briefly discuss them in the following.

LSTM. Long Short-Term Memory is a specialized RNN to mitigate the gradient vanishing problem (Figure 3). LSTMs learn long-term dependencies using gates, which can learn what information in the sequence is important to keep or throw away. LSTMs have three gates: input, forget, and output.

An LSTM cell works in three steps. The first step is to decide which information to be removed from the cell in that particular step. It looks at the previous information along with the current input and computes the following via a sigmoid function (σ).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{6}$$

f_t is the forget gate, it decides which information to delete from the previous time step that is no more important. h_{t-1} denotes the output from the LSTM cell at time step $t-1$, i.e., the output of previous cell and x_t denotes the input to the LSTM cell at time step t. W_f and b_f denote matrices and vectors of parameters, respectively.

The second step in LSTM is to decide how much this cell adds to the current state. This step consists of two parts. First is the sigmoid function whose value varies between 0 to 1, it helps to decide which information to pass through. Second is the tanh function whose value varies between -1 to 1, which

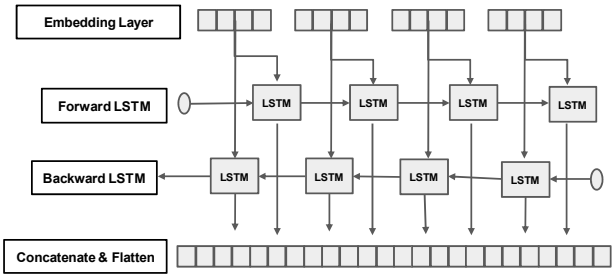


Fig. 4. Architecture of BiLSTM

gives weight to the information being passed, deciding their level of importance.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (7)$$

i_t is the input gate that decides which information to let through based on its significance in the current time step. h_{t-1} is the output from LSTM cell at time step $t-1$, i.e., the output of previous cell, and x_t denotes the input to LSTM cell at time step t. W_i, W_c, b_i, and b_c are parameter matrices and vectors.

The third step in LSTM is to decide what part of the current cell state makes it to the output. Starting with the sigmoid function, which decides what parts of the cell state make it to the output, we next put the cell state through tanh function to normalize the values between -1 and 1, and multiply it by the output of the sigmoid gate.

$$o_t = \sigma(W_O \cdot [h_{t-1}, x_t] + b_O)$$
$$h_t = o_t \cdot \tanh(\tilde{C}_t) \quad (8)$$

o_t is the output gate that allows the passed-in information to impact the output in the current step. h_{t-1} and h_t denote the output from the LSTM cell at time step $t-1$ and t, respectively, and x_t denotes the input to the LSTM cell at time step t. W_o and b_o denote parameter matrices and vectors.

BiLSTM. Bidirectional LSTM is an extension of LSTM which can further increase the performance of the model on the problem of sequence classification. BiLSTMs aggregate previous and upcoming input information of a certain time step in LSTM cells. The main difference is that instead of just one LSTM, there are two LSTMs: one taking the input in a forward direction, and the other one in a backward direction, that is, a reversed copy of the input sequence (Figure 4). BiLSTMs effectively increase the amount of information, thereby improving the context available to the algorithm, for example, knowing what edges immediately follow and precede an edge in a minimum DFS code.

GRU. Gated Recurrent Unit (GRU) is a similar, but a more sophisticated version of the LSTM, which is also capable of learning long-term dependencies. Here, the major variation from LSTM is that the forget gate and the input gate are combined into a single gate, called the update gate (Figure 5). The hidden state and cell state are also merged. The resulting model is much easier to understand than normal LSTM

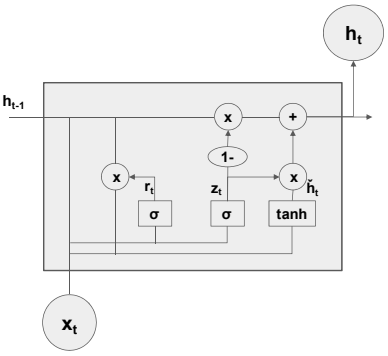


Fig. 5. Architecture of a GRU cell

models, and it is gaining popularity. Due to less number of states, this is less complex and hence faster than LSTM.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$
$$\tilde{h}_t = \tanh(W \cdot [r_t \cdot h_{t-1}, x_t] + b_h)$$
$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \quad (9)$$

Here, $\tilde{h}_t$ is the candidate activation vector, z_t is the update gate and r_t is the reset gate. h_{t-1} and h_t denote the output from GRU cell at time step $t-1$ and t, respectively. x_t denotes the input to GRU cell at time step t. W and b denote parameter matrices and vectors.

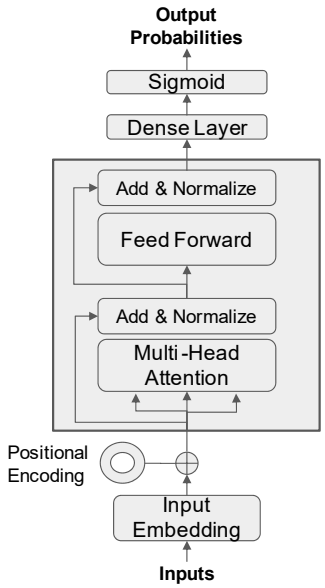


Fig. 6. Architecture of an encoder of the Transformer

Transformer. Transformer was introduced to solve the sequence-to-sequence tasks efficiently while handling long-term dependencies with ease. In its structure, it neither uses the convolutional neural network (CNN), nor the recurrent neural network (RNN), but only the mechanism of self-attention, specifically, multi-head self-attention. The original Transformer model consists of an encoder and a decoder. For the classification task, we only use its encoder. There are two sub-layers in the encoder layer. The initial sub-layer performs a multi-head self-attention mechanism, and the latter

TABLE I
PROPERTIES OF DATASETS

datasets	category	#classes & distribution	#node-labels	#edge-labels	(avg.) #nodes per graph	(avg.) #edges per graph
MUTAG	bioinformatics	2 [inactive:63, active:125]	7	4	18	20
NCI-H23	bioinformatics	2 [inactive:2000, active:500]	15	3	29	31
Tox21_AR	bioinformatics	2 [no-toxic:8553, toxic:369]	41	4	18	18
PTC_FR	bioinformatics	2 [no-toxic:230, toxic:121]	19	4	15	15
IMDB-BINARY	social network	2 [action:500, romance:500]	65	1	20	97
DBLP	collaboration	2 [DBDM:1500, CVPR:1500]	11083	3	11	20

is a position-wise fully connected feed-forward network. A residual connection is employed around each of the two sub-layers, followed by a normalization layer. The inputs are first embedded into d-dimensional vectors as we cannot directly use strings. Since, unlike RNNs, the model contains no recurrence and no convolutional network, for the model to remember how sequences are fed into it, it must provide some information about the relative or absolute positions of the tokens in the sequence – this is what the positional embedding does. These positional embeddings are added to the embedded d-dimensional vector representation of each word.

In our encoder layer, we use two attention heads and small dimensions (d=32) for projection heads and feed-forward network. The Transformer layer outputs one vector for each time step of the input sequence. Here, we take the mean of Transformer outputs at all time steps and use a two-layered feed-forward network over it to classify minimum DFS codes as shown in Figure 6.

C. Improving Performance with Minimum DFS Code Variant

In every edge representation as a 5-tuple, that is, $(u, v, L(u), L(e), L(v))$, the minimum DFS code includes the two corresponding node ids based on their time-stamps. This imposes a constraint that a particular subgraph pattern (e.g., an edge, a cycle, a path, a clique, etc.) would occur at a specific location in the input graph. However, in the context of graph classification, it is often beneficial if one simply detects the presence of a pattern in a graph, irrespective of its location. This is because in real classification tasks, we rarely deal with the exact isomorphic graphs, but rather graphs that are similar, and share many identical patterns. Past literature about kernel and structural feature-based graph classification were based on similar ideas [48], [35], [27], [43], [14], [39].

To this end, we develop a variant of the minimum DFS code, where we still consider edges in the minimum lexicographic order; however, each edge $e = (u, v)$ is represented with a 3-tuple: $(L(u), L(e), L(v))$, instead of 5-tuple representation. As earlier, each minimum DFS code variant is considered as a "sequence", and every 3-tuple (denoting an edge) in it as a "word". We instantiate our vocabulary with all seen words, then we convert each unique word in our sequences into a unique integer using the vocabulary, that is, we tokenize each distinct word into a unique integer or token, and finally we embed each integer into a 32-dimensional vector [53]. This ensures that our RNN/ Transformer-based classification models can employ the "presence" of a pattern in a graph as a feature, and may disregard its location. It also reduces the complexity of the feature space and improves generalizability. While such minimum DFS code variant is no longer injective, it improves the classification performance over many real-world graph datasets as shown in our experiments.

IV. EXPERIMENTAL RESULTS

We conducted experiments to measure the accuracy and efficiency of our framework, and compared them against four state-of-the-art GNNs, using six real-world networks.

V. EXPERIMENTAL SETUP

We employed Keras [53] and TensorFlow v2.4.1 deep learning libraries [1] to build LSTM, GRU, BiLSTM, and Transformer models. The NetworkX library [18] was used to load network data as Python objects. To perform experiments, we used the Central Processing Unit (CPU) of Google Colab [5] whose specifications are: Intel(R) Xeon(R) Processor with two cores @ 2.20GHz and 13GB RAM. Our codebase and datasets are at [17].

Datasets. We used six real-world graph datasets [1] from three different categories (Table I).

The first four datasets belong to bioinformatics benchmarks: *MUTAG* [9] contains mutagenic aromatic and heteroaromatic nitro compounds classified according to their mutagenic effects on a bacterium. *Tox21_AR* [46] dataset consists of qualitative toxicity measurements of compounds. *PTC_FR* [41] contains chemical compounds classified according to their carcinogenicity on female rats. *NCI-H23* is a dataset of anti-cancer screens for cell lung cancer over small molecules. For experiments on *NCI-H23*, we randomly sample 500 active compounds and 2000 inactive compounds [48].

The fifth dataset belongs to the social networks benchmark: *IMDB-BINARY* [50] is a movie collaboration dataset in which each graph corresponds to an ego-network for each actress/actor and is derived from the "Action" and "Romance" genre movies in the IMDB. In these graphs, each node corresponds to an actress/actor and an edge connects them if they appear in the same movie.

The sixth network belongs to the research collaboration category: *DBLP* consists of bibliography data in computer science. Each paper in *DBLP* is associated with several attributes, e.g., abstract, authors, year, venue, title, and references. A graph is built for each paper as follows [32]. (1) Each paper ID is a node; (2) if a paper P.A cites another paper P.B, there is an edge between P.A and P.B; (3) each keyword in the title is

[1] https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets

also a node; (4) each paper ID node is connected to keyword nodes of the paper; and (5) for each paper, its keyword nodes are fully connected with each other. Our task is to classify a paper if it belongs to DBDM (database and data mining) or CVPR (computer vision and pattern recognition) field. We randomly selected 1500 graphs from each class.

Node and edge labels. In the first four bioinformatics datasets, the node labels are chemical atoms, e.g., N, C, O, etc., and the edge labels are types of chemical bonds, e.g., single bond, double bond, etc. In *IMDB-BINARY*, as there were no existing node labels and edge labels, we assigned node label as the degree of the respective node [15], and we assumed that all edges have the same label. In our final dataset, *DBLP*, the node labels are keywords or Paper IDs, and each edge denotes the citation relationship between papers or keyword relations in the title which are of three types: W2W (word-to-word), W2P (word-to-paper), and P2P (paper-to-paper).

Hyperparameters selection. We performed 5-fold cross-validation. All data splits were stratified, i.e., class proportions were preserved. We employed the Binary Cross-Entropy loss function and the ADAM optimizer. The learning rate was selected from $\{0.01, 0.001, 0.0001\}$ and the number of epochs was selected from (1, 350). The batch size was set as 64. For each dataset, we instantiated our vocabulary with all seen words (i.e. all distinct words present in the processed dataset). For RNN models, that is, LSTM, GRU, and BiLSTM, we used one layer of the respective type with 50 internal units. For the Transformer model, we used one layer of the Transformer block with 32 hidden units in the feed-forward network.

Competing methods. Recent GCNs perform a recursive neighborhood aggregation via message passing. We considered four state-of-the-art GCNs [12], that develop more sophisticated pooling operations for graphs. We followed the same GCN architectures as in the original papers, since these settings also performed well (for that specific GCN) in our experiments. Hyperparameters for the competing methods were also tuned via the ADAM optimizer.

GCN+GAP [33], [25]. This architecture consists of graph convolutional layers [22], followed by a global average pooling (GAP) layer, and a softmax classifier. Convolutional layers extract localized spatial features from graphs, and the GAP layer enforces correspondences between the feature maps and the classes. The global average pooling has no parameter to optimize, thus it acts as a structural regularizer. We used the GCNN+GAP architecture as in [33]: three graph convolutional layers of size 128, 256, and 512, respectively, followed by a GAP layer, and a softmax classifier.

DGCNN [52] consists of graph convolutional layers, followed by a *SortPooling* layer, classic convolutional and dense layers, and finally a softmax classifier. The convolutional layers extract localized spatial features from the input graph, and define a sorting order among nodes. The SortPooling layer sorts nodes based on the previously defined order and selects the top-k nodes. The classic convolutional and dense layers read the sorted graph representation and predict the class label via the softmax classifier. Following [52], our DGCNN architecture has four graph convolution layers with 32, 32, 32, 1 output channels, respectively. We set the k of SortPooling such that 60% of the graphs have nodes more than k. The remaining layers consist of two 1-D convolutional layers and one dense layer. The first 1-D convolutional layer has 16 output channels followed by a MaxPooling layer with filter size 2 and step size 2. The second 1-D convolutional layer has 32 output channels, filter size 5, and step size 1. The dense layer has 128 hidden units, followed by a softmax layer as the output layer.

DiffPool [51] generates hierarchical representations of graphs, by learning a differentiable soft cluster assignment for nodes at each layer of a graph neural network (GNN), mapping nodes to a set of clusters, which then form the coarsened input for the next GNN layer. Embedding vectors from the last DIFFPOOL layer are combined to form one single embedding

TABLE II
MINIMUM DFS CODE-BASED GRAPH CLASSIFICATION RESULTS

datasets	performance metrics	LSTM	BiLSTM	GRU	Transformer
MUTAG	AUC-ROC	0.956	0.955	**0.961**	0.960
	AUC-PR	0.981	0.981	0.974	**0.982**
	Avg. Epoch Time (s)	**0.12**	0.16	**0.12**	0.12
NCI-H23	AUC-ROC	0.854	0.848	0.842	**0.868**
	AUC-PR	0.670	0.660	0.653	**0.698**
	Avg. Epoch Time (s)	2.57	3.57	**2.32**	3.29
Tox21_AR	AUC-ROC	0.843	0.837	0.840	**0.843**
	AUC-PR	0.587	0.584	0.589	**0.599**
	Avg. Epoch Time (s)	11.14	16.18	**9.44**	17.61
PTC_FR	AUC-ROC	**0.594**	0.585	0.600	0.588
	AUC-PR	0.452	0.444	0.458	**0.484**
	Avg. Epoch Time (s)	0.27	0.40	**0.26**	0.32
IMDB -BINARY	AUC-ROC	0.812	0.804	0.802	**0.821**
	AUC-PR	0.816	0.810	0.810	**0.829**
	Avg. Epoch Time (s)	8.26	12.39	**7.72**	48.23
DBLP	AUC-ROC	0.597	**0.624**	0.575	0.617
	AUC-PR	0.612	**0.627**	0.575	0.613
	Avg. Epoch Time (s)	0.85	1.12	0.89	**0.81**

TABLE III
MINIMUM DFS CODE VARIANT-BASED GRAPH CLASSIFICATION RESULTS

datasets	performance metrics	LSTM	BiLSTM	GRU	Transformer
MUTAG	AUC-ROC	**0.953**	0.950	0.928	0.937
	AUC-PR	**0.978**	0.977	0.968	0.969
	Avg. Epoch Time (s)	0.12	0.16	**0.11**	0.12
NCI-H23	AUC-ROC	**0.901**	0.900	0.897	0.865
	AUC-PR	**0.764**	0.762	0.748	0.632
	Avg. Epoch Time (s)	2.49	3.43	**2.18**	3.32
Tox21_AR	AUC-ROC	0.843	0.840	0.835	**0.844**
	AUC-PR	0.585	**0.594**	0.572	0.553
	Avg. Epoch Time (s)	10.38	15.62	**8.98**	16.80
PTC_FR	AUC-ROC	**0.691**	0.686	0.617	0.636
	AUC-PR	0.547	**0.560**	0.518	0.533
	Avg. Epoch Time (s)	0.27	0.41	**0.25**	0.32
IMDB -BINARY	AUC-ROC	0.818	0.820	0.808	**0.834**
	AUC-PR	0.816	0.827	0.809	**0.836**
	Avg. Epoch Time (s)	9.38	12.92	**8.78**	48.09
DBLP	AUC-ROC	0.794	0.811	0.806	**0.831**
	AUC-PR	0.811	0.828	0.818	**0.846**
	Avg. Epoch Time (s)	0.90	1.10	0.83	**0.77**

TABLE IV
MINIMUM DFS CODE-BASED GRAPH CLASSIFICATION; WITHOUT CONSIDERING EDGE LABELS

datasets	performance metrics	LSTM	BiLSTM	GRU	Transformer
MUTAG	AUC-ROC	0.966	0.966	0.960	**0.969**
	AUC-PR	0.985	0.984	0.982	**0.986**
NCI-H23	AUC-ROC	0.869	0.866	0.857	**0.870**
	AUC-PR	0.690	0.694	0.680	**0.701**
PTC_FR	AUC-ROC	**0.620**	0.611	0.602	0.599
	AUC-PR	0.479	0.469	**0.483**	0.464
DBLP	AUC-ROC	0.617	**0.630**	0.610	0.626
	AUC-PR	0.607	**0.638**	0.600	0.631

TABLE V
MINIMUM DFS CODE VARIANT-BASED GRAPH CLASSIFICATION; WITHOUT CONSIDERING EDGE LABELS

datasets	performance metrics	LSTM	BiLSTM	GRU	Transformer
MUTAG	AUC-ROC	**0.948**	0.932	0.946	0.944
	AUC-PR	**0.975**	0.967	0.973	0.971
NCI-H23	AUC-ROC	**0.895**	0.886	0.887	0.844
	AUC-PR	**0.734**	0.714	0.706	0.604
PTC_FR	AUC-ROC	**0.661**	0.650	0.644	0.626
	AUC-PR	**0.521**	0.512	0.512	0.518
DBLP	AUC-ROC	0.806	0.818	0.811	**0.832**
	AUC-PR	0.822	0.838	0.815	**0.843**

TABLE VI
MINIMUM DFS CODE COMPUTATION TIME

datasets	min DFS code computation time (avg) per graph in seconds
MUTAG	0.06
NCI-H23	0.06
Tox21_AR	0.05
PTC_FR	0.04
IMDB-BINARY	9.89
DBLP	0.05

vector by taking maximum across each embedding dimension. This final vector is used as input to fully-connected layers, followed by a softmax classifier to predict the class label. We employed one DIFFPOOL layer in our implementation. In a DIFFPOOL layer, the number of clusters is set as 25% of the number of nodes before applying the DIFFPOOL. Every DIFFPOOL layer consists of three GCNs (each of size 64) for embedding generation and another three GCNs (each of size 64) for computing probabilistic assignments. The final embedding vector generated from the DIFFPOOL layer is used as input to three fully-connected layers (each having 50 hidden units), followed by a softmax classifier.

GIN [47]. In graph isomorphism network (GIN), the aggregation functions and the readout functions are made injective, as discussed in § II. GIN's discriminative/representational power is equal to the power of 1-WL test. In our implementation, 4 GIN layers were applied and all MLPs had 2 layers. Batch normalization was applied on every layer. We used the sum function for neighbor aggregation, and also the sum readout function. ADAM optimizer was used with initial learning rate as 0.001, decay rate of 0.5 with a patience of 25 epochs. Batch size was set to 20 and the number of epochs was 350, in which the scores of the converging epoch is reported.

Evaluation metrics. AUC-ROC and AUC-PR [46], [35] were employed to analyze the performance of all classification models. These evaluation metrics are suitable for imbalanced datasets (e.g., ours in Table I). **AUC-ROC** defines the area under the curve that shows the tradeoff between the true positive rate (TPR) and the false positive rate (FPR). In the ROC plot, random classifiers show a diagonal line, thus the baseline AUC-ROC score is 0.5. Higher AUC-ROC score indicates that the curve is more top-left, implying a higher TPR and lower FPR for each threshold, and thereby a better classifier. **AUC-PR** measures the area under the curve that combines precision and recall in a single plot. The baseline of PR is computed by the ratio of positives (P) and negatives (N) as: P/(P + N), e.g., for *MUTAG* the baseline AUC-PR is 125/(125+63) = 0.66, thus 66% correct predictions, on average, among the positive predictions by a random classifier. A higher AUC-PR score indicates a better classifier.

Average epoch time is computed as the total time taken during training of a model divided by the number of epochs. This is used to measure the efficiency of training.

A. Accuracy and Efficiency

We report our accuracy and efficiency results for minimum DFS code-based graph classification in Table II. Clearly, there is no single winner. We noticed that the Transformer-based minimum DFS code classification generally outperforms RNN-based classification methods (LSTM, BiLSTM, and GRU) in both AUC-ROC and AUC-PR scores. This could be due to the attention operation in the Transformer which provides context for any position in the input sequence. In terms of the average epoch time, GRU is the fastest (due to having only two gates), and BiLSTM is often the slowest – as it consists of two LSTMs taking the input in both forward and backward directions. We found that the running times of both LSTM and Transformers are comparable on our datasets. Furthermore, with GPU support that helps in parallelization, the average epoch time in Transformer could improve several folds, since it processes each word independently.

In Table III, we consider the classification of the minimum DFS code variant. Each edge $e = (u, v)$ is represented with a 3-tuple: $(L(u), L(e), L(v))$. This consistently produces better AUC-ROC and AUC-PR scores compared to when the classic minimum DFS code is used. With the exception of *MUTAG* (both scores) and *Tox21_AR* (AUC-PR score), in all remaining datasets the minimum DFS code variant results in better AUC-ROC and AUC-PR. Even in *Tox21_AR*, this produces a higher AUC-ROC score. As reasoned earlier, minimum DFS code variants reduce the complexity of the feature space and improve generalizability. The minimum DFS code variant-based sequence classification is also faster for the same reason.

In Tables IV and V, we present AUC-ROC and AUC-PR scores over four datasets without considering edge labels, that is, we assign same label to all edges in these experiments. It can be seen that when minimum DFS codes (5-tuples) are used, not having edge label information further improves these scores, indicating generalizability in our training process. In contrast, with minimum DFS code variants that employ only 3-tuples, having edge label information improves the accuracy. *In summary, the minimum DFS code variants with edge labels (i.e., Table III) generally result in the best AUC-ROC and AUC-PR scores over our datasets.*

The average time to compute the minimum DFS code per graph is reported in Table VI. We noticed that the overhead due to the minimum DFS code computation per graph is modest compared to the average epoch time. The minimum DFS code for each graph can be computed in parallel, which would further improve the efficiency.

B. Comparison with Existing GCNs

We compared our minimum DFS code-based sequence classification method with four state-of-the-art GCNs [12]: GCN+GAP [33], [25], DGCNN [52], DiffPool [52], and GIN [47]. Table VII reports their accuracy and efficiency comparison results. Our method always outperforms them in terms

TABLE VII
COMPARISON OF MINIMUM DFS CODE-BASED GRAPH CLASSIFICATION WITH STATE-OF-THE-ART GCNS

datasets	performance metrics	GCN + GAP [33]	DGCNN [52]	DiffPool [52]	GIN [47]	Min-DFS Code/Variant + RNN/Transformer [Ours]
MUTAG	AUC-ROC	0.787	0.902	0.838	0.924	**0.969**
	AUC-PR	0.881	0.948	0.901	0.964	**0.986**
	Avg. Epoch Time (s)	0.42	0.41	0.52	0.43	**0.12**
NCI-H23	AUC-ROC	0.512	0.627	0.698	0.861	**0.901**
	AUC-PR	0.215	0.343	0.436	0.611	**0.764**
	Avg. Epoch Time (s)	8.81	5.46	7.84	6.48	**2.49**
Tox21_AR	AUC-ROC	0.624	0.718	0.784	0.781	**0.844**
	AUC-PR	0.058	0.139	0.339	0.257	**0.599**
	Avg. Epoch Time (s)	21.54	17.28	24.21	**14.64**	16.80
PTC_FR	AUC-ROC	0.639	0.630	0.606	0.514	**0.691**
	AUC-PR	0.539	0.528	0.522	0.392	**0.560**
	Avg. Epoch Time (s)	0.76	0.76	0.98	0.60	**0.27**
IMDB -BINARY	AUC-ROC	0.700	0.803	0.790	0.609	**0.834**
	AUC-PR	0.663	0.811	0.787	0.623	**0.836**
	Avg. Epoch Time (s)	2.53	**2.22**	3.05	73.10	48.09
DBLP	AUC-ROC	0.642	0.816	0.705	Out	**0.832**
	AUC-PR	0.598	0.780	0.630	of	**0.846**
	Avg. Epoch Time (s)	5.76	4.66	7.25	Memory	**0.77**

of both AUC-ROC and AUC-PR scores. This is due to the following reasons: (1) Minimum DFS codes are injective, (2) our LSTM, BiLSTM, GRU, and Transformer-based sequence classification methods can capture long-range dependencies in the graph space, and (3) the attention operation in Transformer provides context for any position in the input sequence.

In terms of running times, our method is comparable (or even faster) than these competing approaches, except on *IMDB-BINARY*. This dataset has the largest size graphs among all our datasets, as a result the minimum DFS code per graph is also larger. Hence, our sequential implementation of the Transformer consumes longer times. With GPU parallelization, we believe that the average epoch time in Transformer could improve several folds, since it processes each edge in the minimum DFS code independently.

The above results demonstrate the effectiveness and efficiency of our minimum DFS code-based sequence classification approach, compared to existing GCNs.

VI. CONCLUSIONS AND FUTURE WORK

We investigated the novel direction of minimum DFS code-based graph classification. Two graphs are isomorphic iff they have the same minimum DFS code, which can be efficiently computed in practice. We employed state-of-the-art RNN-based sequence classification techniques (LSTM, BiLSTM, GRU), and the encoder layer of Transformer over minimum DFS codes, for graph classification. Our method can capture long-range dependencies in the graph space. We also consider a novel variant of the minimum DFS code that improves generalizability and classification accuracy over many network datasets. Our experimental results with six real-world graph datasets demonstrate the accuracy and efficiency of our techniques, compared to existing GCNs. Based on our thorough empirical evaluation, we recommend the minimum DFS code variants (i.e., 3-tuple for an edge) with edge label information to produce the best AUC-ROC and AUC-PR scores. We open-sourced our solution framework [17] in which one can play with different graph datasets and get their classification results – this would benefit researchers, practitioners, biologists, social scientists, and data scientists.

Future work could be in several directions. **First**, more expressive GNNs [28], [30], [6] are being developed that replicate the increasingly more powerful k-WL tests. However, such GNNs result in high computational and memory complexity, non-local message passing, and a large number of parameters; and this is still an insufficient condition for graph isomorphism — for every k, there are non-isomorphic graphs indistinguishable by k-WL [7]. One can compare our minimum DFS code-based graph classification approach with them. **Second**, one drawback of the minimum DFS code is that it permits only one label for every node and edge, and the labels must be categorical in nature. Thus, the minimum DFS code verifies only a strict notion of isomorphism between two graphs. In reality, a graph dataset may have multiple features per node and edge, and some of these features could be numerical (e.g., age, salary, etc.). Thus, we may require more flexible means to compute similarity across nodes, edges, and substructures. In future, it would be interesting to consider minimum DFS code variants that can accommodate multiple labels per node and edge, as well as support numerical node and edge labels similarity across graphs.

Acknowledgement. Arijit Khan is supported by MOE Tier1 and Tier2 grants RG117/19, MOE2019-T2-2-042.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," http://tensorflow.org/.

[2] U. Alon and E. Yahav, "On the Bottleneck of Graph Neural Networks and its Practical Implications," CoRR abs/2006.05205, 2020.

[3] L. Babai, "Graph Isomorphism in Quasipolynomial Time [Extended Abstract]," STOC, 2016.

[4] L. Babai, P. Erdös, and S. M. Selkow, "Random Graph Isomorphism," SIAM J. Comput., vol. 9, no. 3, 628–635, 1980.

[5] E. Bisong, Google Colaboratory. Apress, 59–64, 2019.

[6] G. Bouritsas, F. Frasca, S. Zafeiriou, and M. M. Bronstein, "Improving Graph Neural Network Expressivity via Subgraph Isomorphism Counting,", CoRR abs/2006.09252, 2020.

[7] J. Cai, M. Fürer, and N. Immerman, "An Optimal Lower Bound on the Number of Variables for Graph Identifications," Comb., vol. 12, no. 4, 389–410, 1992.

[8] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," EMNLP, 2014.

[9] A. K. Debnath, d. C. R. Lopez, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity Relationship of Mutagenic Aromatic and Heteroaromatic Nitro Compounds. Correlation with Molecular Orbital Energies and Hydrophobicity," J. Medicinal Chemistry, vol. 34, 1991.

[10] K. Ding, J. Li, R. Bhanushali, and H. Liu, "Deep Anomaly Detection on Attributed Networks," SDM, 2019.

[11] D. Duvenaud, D. Maclaurin, J. A.-Iparraguirre, R. G.-Bombarelli, T. Hirzel, A. A.-Guzik, and R. P. Adams, "Convolutional Networks on Graphs for Learning Molecular Fingerprints," NeurIPS, 2015.

[12] F. Errica, M. Podda, D. Bacciu, and A. Micheli, "A Fair Comparison of Graph Neural Networks for Graph Classification," ICLR, 2020.

[13] M. Fan, X. Luo, J. Liu, M. Wang, C. Nong, Q. Zheng, and T. Liu, "Graph Embedding based Familial Analysis of Android Malware using Unsupervised Learning," ICSE, 2019.

[14] T. Gärtner, P. A. Flach, and S. Wrobel, "On Graph Kernels: Hardness Results and Efficient Alternatives," COLT, 2003.

[15] N. Goyal, H. V. Jain, and S. Ranu, "GraphGen: A Scalable Approach to Domain-agnostic Labeled Graph Generation," WWW, 2020.

[16] M. Grohe and P. Schweitzer, "The Graph Isomorphism Problem," Commun. ACM, vol. 63, no. 11, 128–134, 2020.

[17] J. Gupta and A. Khan, "Graph Classification with Minimum DFS Code [Code]," https://github.com/DFS-graph-classify/DFS-graph-classification.

[18] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," SciPy, 2008.

[19] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation Learning on Graphs: Methods and Applications," IEEE Data Eng. Bull., vol. 40, no. 3, 2017.

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural computation, vol. 9, no. 8, 1735–1780, 1997.

[21] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation, 3rd Edition, Addison-Wesley, 2007.

[22] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," ICLR, 2017.

[23] J. B. Lee, R. A. Rossi, and X. Kong, "Graph Classification using Structural Attention," KDD, 2018.

[24] Q. Li, Z. Han, and X.-M. Wu, "Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning," AAAI, 2018.

[25] M. Lin, Q. Chen, and S. Yan, "Network In Network", ICLR, 2014.

[26] X. Liu, H. Pan, M. He, Y. Song, X. Jiang, and L. Shang, "Neural Subgraph Isomorphism Counting," KDD, 2020.

[27] O. Macindoe and W. Richards, Graph Comparison Using Fine Structure Analysis, In SocialCom / PASSAT, 2010.

[28] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman, "Provably Powerful Graph Networks," NeurIPS, 2019.

[29] H. L. Morgan, "The Generation of a UniqueMachine Description for Chemical Structures - A Technique Developed at Chemical Abstracts," Service. J. Chem. Document, vol. 5, no. 2, 107–113, 1965.

[30] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks," AAAI, 2019.

[31] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning Convolutional Neural Networks for Graphs," ICML, 2016.

[32] S. Pan, X. Zhu, C. Zhang, and P. S. Yu, "Graph Stream Classification using Labeled and Unlabeled Graphs," ICDE, 2013.

[33] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann, "Explainability Methods for Graph Convolutional Neural Networks," CVPR, 2019.

[34] A. Prateek, A. Khan, A. Goyal, and S. Ranu, "Mining Top-k Pairs of Correlated Subgraphs in a Large Network," PVLDB, vol. 13, no. 9, 1511–1524, 2020.

[35] S. Ranu and A. K. Singh, "GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases," ICDE, 2009.

[36] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," IEEE Trans. Neural Networks, vol. 20, no. 1, 61–80, 2009.

[37] M. Schuster and K. K. Paliwal, "Bidirectional Recurrent Neural Networks," IEEE Trans. Signal Process., vol. 45, no. 11, 2673–2681, 1997.

[38] R. Sharan, S. Suthram, R. M. Kelley, T. Kuhn, S. McCuine, P. Uetz, T. Sittler, R. M. Karp, and T. Ideker, "Conserved Patterns of Protein Interaction in Multiple Species," PNAS, vol. 102, no. 6, 2005.

[39] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-Lehman Graph Kernels," J. Mach. Learn. Res., vol. 12, 2539–2561, 2011.

[40] A. Sperduti and A. Starita, "Supervised Neural Networks for the Classification of Structures," IEEE Trans. Neural Networks, no. 8, vol. 3, 714–735, 1997.

[41] H. Toivonen, A. Srinivasan, R. D. King, S. Kramer, and C. Helma, "Statistical Evaluation of the Predictive Toxicology Challenge," Bioinform., vol. 19, no. 10, 2000-2001, 2003.

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," NeurIPS, 2017.

[43] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph Kernels," J. Mach. Learn. Res., vol. 11, 2010.

[44] Z. Wang, Y. Ma, Z. Liu, and J. Tang, "R-Transformer: Recurrent Neural Network Enhanced Transformer," CoRR abs/1907.05572, 2019.

[45] B. Weisfeiler and A. A. Lehman, "A Reduction of a Graph to a Canonical Form and an Algebra Arising During This Reduction," Nauchno-Technicheskaya Informatsia, vol. 2, no. 9, 12–16, 1968.

[46] Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. S. Pande, "MoleculeNet: A Benchmark for Molecular Machine Learning," CoRR abs/1703.00564, 2017.

[47] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" ICLR, 2019.

[48] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining Significant Graph Patterns by Leap Search," SIGMOD, 2008.

[49] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," ICDM, 2002.

[50] P. Yanardag and S. V. N. Vishwanathan, "Deep Graph Kernels," KDD, 2015.

[51] Z. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical Graph Representation Learning with Differentiable Pooling," NeurIPS, .

[52] M. Zhang, Z. Cui,M. Neumann, and Y. Chen, "An End-to-End Deep Learning Architecture for Graph Classification," AAAI, 2018.

[53] S. Zhu and F. Chollet, "Working with RNNs," https://keras.io/guides/working_with_rnns/.