Detecting Implementation Bugs in Graph Convolutional Network based Node Classifiers

Yiwei Wang*, Wei Wang*, Yujun Cai†, Bryan Hooi*, Beng Chin Ooi*
*School of Computing, National University of Singapore, Singapore
Email: {y-wang,wangwei,bhooi,ooibc}@comp.nus.edu.sg
†Nanyang Technological University, Singapore
Email: yujun001@e.ntu.edu.sg

***Abstract*—Graph convolutional networks (GCNs) have achieved state-of-the-art performance on the task of node classification. However, the performance of GCNs is prone to implementation bugs that do not explicitly produce compile-time or run-time errors but degrade their effectiveness heavily. These bugs are hard to detect, since the way in which the node attributes and graph structures contribute to the outputs is complicated, non-transparent, and not traceable by humans. To address this issue, we propose a systematic approach with formal justifications to detect implementation bugs in GCN based node classifiers. Our approach is based on the idea of Metamorphic Testing, which does not check input-output relations for a single input, but for input-output pairs. To speed up our approach, we design a pipeline system, which synchronizes the workload on CPUs and GPUs adaptively and processes them simultaneously. Our empirical study shows that our approach is able to identify over 80% of the synthetic mutants and two real-world bugs in GCN implementations. In addition, our pipeline system can achieve more than 10× speedup over the sequential system that leaves the CPU/GPU idle when using the other.**

***Keywords*-Graph Convolutional Networks, Metamorphic Testing, Implementation Bugs, Node Classification**

I. Introduction

Node classification refers to the problem of classifying nodes (such as documents) in a graph (such as a citation network), where labels are only available for a subset of nodes. It is a fundamental task for evaluating the machine learning models on graphs, and meanwhile supports miscellaneous practical applications, e.g., learning molecular fingerprints [1] and predicting entity properties [2]. On this task, graph convolutional networks (GCNs) have made breakthrough advancements and become the default solution due to its state-of-the-art effectiveness [3], for which a comprehensive survey can be found in [4].

When a GCN based node classifier fails, for example in Fig. 1, people may train the model with more data since they believe that more training data can improve its performance [5]. However, this does not take effect if the implementation is wrong. To address this issue, in this paper, we focus on detecting implementation bugs of GCNs, which do not explicitly induce compile-time or run-time errors but heavily degrade their performance, in an effective and efficient manner.

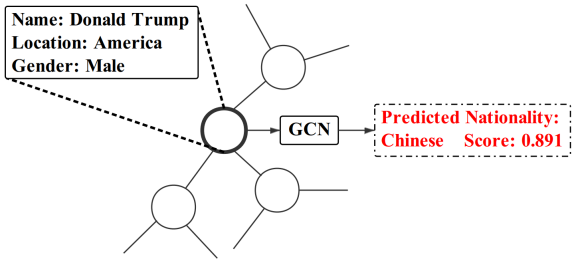


Figure 1: *A GCN based node classifier makes a wrong prediction on the entity property.* A possible fundamental cause is an implementation bug, which needs to be detected first. But due to the poor explainability of GCNs, it is hard to determine whether the error arises from bugs, or from other problems, such as insufficient training data.

To verify software, the traditional techniques build 'test cases' which contain '$input$'-'$output$' pairs. They supply the '$input$' to the program under test (PUT) and check whether the output matches their expectations. However, this mechanism is generally infeasible for GCNs, because finding one (or a few) instances that do not meet the expectations, such as the case in Fig. 1, does not necessarily indicate the presence of an implementation bug. In principle, GCNs are a form of neural network, which classifies different nodes by transforming the node attributes and graph structures via a multi-layer nonlinear function. This function is mathematically complicated, and how inputs contribute to the outputs is not traceable by humans. Many causes, e.g., adversarial examples and deficient data, can account for the failure as in Fig. 1; it is difficult to determine which is true, because GCNs lack complete interpretability and theoretical support [6].

Although significant effort has been devoted to developing GCN models, the approaches for testing them are under-explored [7]. In this field, we argue that Metamorphic Testing (MT) [8] is useful, since it does not need the correct

output of a single input, which is generally missing for GCN models. MT builds a test case containing two pairs: $input_a$-$output_a$ and $input_b$-$output_b$. $input_b$ is created from $input_a$ so that $output_a$ and $output_b$ have to meet a relation for the correct implementation, which is called a Metamorphic Relation (MR). However, developing and justifying MRs for GCNs is challenging, because the $input - output$ relationships of GCN models are mathematically complicated.

GCNs are neural networks (NNs) for processing graph data. Some MRs have been proposed to verify other NNs on image data [9] and time series [10]. However, it is highly challenging to extend these MRs to GCNs, because the graph data is more irregular and noisy, and exhibits more complex relations among different samples (nodes). In addition, GCNs learn semantic representations for nodes through the 'message-passing' mechanism on graphs [3], which is distinct from other NNs.

To shed light on this problem, in this paper, we propose six MRs for detecting implementation bugs in GCN based node classifiers. We justify them in theory and validate their effectiveness through Mutation Testing experiments [11]. Inspired by the similarities between image pixels and node attributes, we extend two MRs proposed for convolutional neural networks (CNNs) in [9] to GCNs. Next, by analyzing the 'message-passing' mechanism and the 'receptive field' of GCNs [12], we propose four novel MRs for GCNs.

It is not straightforward to run MT for GCNs because even the same inputs can lead to different outputs in subsequent runs due to random parameter initialization [3]. A solution to this problem is to fix the random seeds, but this may not work when the model is trained on GPUs [13], [14]. To enable the usage of our proposed MRs on GPUs, we test the implementation without the fixed random seeds. We propose a heuristic solution towards this issue based on the observation that the correct implementations have a much lower standard deviation (std.) of training/validation losses among multiple trials than that of incorrect ones. This holds for miscellaneous GCN implementations and datasets. Thus, we set a threshold for the maximum std. that a normal program can have, and claim that bugs exist if the threshold is broken.

Our proposed approach can be executed by repeatedly generating inputs following MRs, running the programs, and comparing the outputs. Considering that different iterations do not have flow dependencies and each iteration contains workload on both CPUs and GPUs, we design a pipeline scheme to accelerate the testing, which accelerates the CPU running via multi-threading, executes the tasks on CPUs and GPUs simultaneously. We build a queue to store the data generated by CPUs and feed it into GPUs, which help to accommodate irregularities of workload on hybrid hardware.

We evaluate our method on detecting implementation bugs of the vanilla GCN [3], JKNet [15], and LGCN [16]. We utilize popular graph datasets, such as Cora, Citeseer, Pubmed [17], Coauthor-CS, and Coauthor-Physics [18] to train and test the models. Our experimental results show that our method effectively detects mutants and real-world bugs. Moreover, we observe significant efficiency improvements due to our pipeline system, which conducts MT more than 30 times per second when processing the graphs of 10 million edges.

Our contributions are as follows:

1) We propose a set of Metamorphic Relations (MRs) for the GCN based node classifiers. We extend two MRs from those designed for CNNs [9] and provide four novel MRs by analyzing the 'message-passing' scheme and the 'receptive field' of GCNs, all of which are justified by our theoretical analysis.
2) We design a pipeline to accelerate Metamorphic Testing by simultaneously using CPUs and GPUs.
3) We conduct the Mutation Testing experiments to evaluate our approaches. Empirical results show that our approach can identify over 80% of the synthetic mutants and two real-world bugs in GCN implementations. In addition, our pipeline system achieves more than $10\times$ speedup over the sequential system that uses CPUs and GPUs in an asynchronous manner.

II. Related Work

Machine Learning Testing aims to detect differences between existing and required behaviors of machine learning (ML) systems [7], in which detecting implementation bugs is a fundamental task to ensure that an application uses ML models correctly [19]. ML applications belong to 'non-testable programs' [20], of which the expected output cannot be ascertained easily [21]. Some existing work has made an effort to test 'non-testable programs'. In [22], the authors attempt to develop formal mathematical specifications to verify the ML systems. It recognizes the difficulty of verifying ML implementations, but the method needs the developers to learn a new formal language and cannot scale to a wide range of ML algorithms simply. As another option, [23] uses multiple-implementation to test the k-Nearest-Neighbor (kNN) and Naive Bayes (NB) algorithms. [24] uses assertion checking to test ML applications. Metamorphic Testing [8] is found to be excellent to address the problem [25], and offers superior empirical performance among different methods [26]. A survey discussing various methods for testing 'non-testable programs' can be found in [27].

Some work has made progress on conducting MT for ML based applications. The authors of [28] and [21] propose MRs for kNN and NB algorithms. [26] conducts MT for the support vector machines with a linear kernel, while [9] does it for non-linear kernels.

Although recent years have witnessed emerging applications of neural networks [29], MT for NN implementations is not well studied. [30] uses MT to 'validate' the DL models but does not provide theoretical justifications. [9] and [10]

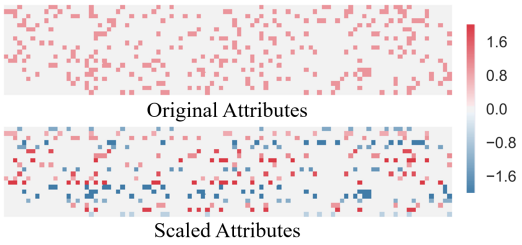


Figure 2: *MR-1 scales the node attributes by non-zero factors.* Each row represents the attributes of a node. The classification results and the losses remain the same whether the GCN is trained and tested on the original or scaled data.

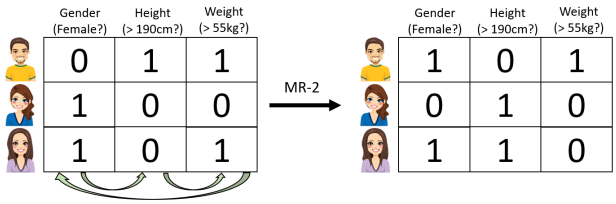


Figure 3: *MR-2 permutes the attributes of nodes.* Each row represents the attributes of a node. After the permutation, attributes changed. For example, the second row becomes a man taller than 190cm while weighing lower than 55kg from a woman shorter than 190cm. However, we prove that their classification results remain the same whether the GCN is trained on the original or permuted data.

propose MRs with formal guarantees for CNNs processing image data and recurrent neural networks (RNNs) for time series respectively. However, to the best of our knowledge, research on MT for GCNs, which deal with graph data, is still absent.

Node classification is a fundamental ML task on graph data [17]: for example, predicting protein types in a protein interaction graph [31]. GCNs have achieved state-of-the-art performance for node classification [3], [32], [33], [16], leading to tremendous interest. But the testing of GCNs is still under-explored. To make a step in filling this gap, we propose our MT approach for detecting the implementation bugs in the GCN based node classifiers.

III. Metamorphic Relations for Graph Convolutional Network based Node Classifiers

In this section, we present our methodology. We first introduce how GCNs are used to conduct node classification briefly [3]. Then we introduce our proposed metamorphic relations (MRs) and provide proof for them. Finally we present our pipeline design for efficiently executing Metamorphic Testing.

A. GCN based Node Classification

We define a graph as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the node set consisting of all the N nodes, and $\mathcal{E} = \mathcal{V} \times \mathcal{V}$ is the set of edges linking nodes. Let $\mathbf{A}$ be the adjacency matrix of the given graph, $A_{ij} = 1$ indicates that edge (i, j) exists between node i and node j, and $A_{ij} = 0$, otherwise.

We consider the well-established GCN model in [3] for node classification, which first adds self-loops to the adjacency matrix as $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, then defines the $(l+1)$th layer by:

$$\mathbf{H}^{(l+1)} = \sigma(\mathbf{D}^{-1/2}\tilde{\mathbf{A}}\mathbf{D}^{-1/2}\mathbf{H}^{(l)}\mathbf{W}^{(l)}). \tag{1}$$

Here $\mathbf{W}^{(l)}$ is a trainable weight matrix for layer l, $\mathbf{D}$ is the diagonal matrix of degrees ($D_{ii} = \sum_j \tilde{A}_{ij}$), and $\sigma(\cdot)$ is a non-linear activation function such as the rectified linear unit (ReLU) [34]. The first layer uses node attributes $\mathbf{X}$ as input, i.e., $\mathbf{H}^{(0)} = \mathbf{X}$, of which the ith row $\mathbf{x}_i$ is the attribute of node i. In the output layer, the final-layer hidden representation $\mathbf{H}^{(L)}$ is passed through a softmax function to obtain the classification scores for each node:

$$\mathbf{P} = softmax(\mathbf{H}^{(L)}), \tag{2}$$

where L is the number of layers. The ith row $\mathbf{p}_i$ of $\mathbf{P}$ is the predicted probability vector of node i belonging to different classes. The ground truth label of each node is stored in the matrix $\mathbf{Y}$, $Y_{ic} = 1$ means node i belongs to class c, and $Y_{ic} = 0$, otherwise. Note that GCNs in practice are limited to 2 or 3 layers [3].

A GCN system typically consists of two modes: training and testing [35]. In the former period, the model is trained to update the parameters $\mathbf{W}_l^{(l)}$, while in the latter one $\mathbf{W}_l^{(l)}$ is fixed. The loss function for training GCNs is as following:

$$\mathcal{L} = \mathcal{L}_{cls} + \alpha\mathcal{L}_{reg}, \tag{3}$$

where

$$\mathcal{L}_{cls} = -\frac{1}{\#\mathcal{V}_T}\sum_{i\in\mathcal{V}_T}\sum_c Y_{ic}\log P_{ic}, \tag{4}$$

and

$$\mathcal{L}_{reg} = \left\|\mathbf{W}^{(0)}\right\|_2. \tag{5}$$

$\mathcal{V}_T$ is the labeled node set in the training data, $\#\mathcal{V}_T$ counts the cardinality of $\mathcal{V}_T$. P_{ic} is the predicted probability of node i being classified as class c. And α is a hyper-parameter for adjusting the effects of the regression loss, which is set to 1 by default. $\left\|\mathbf{W}^{(0)}\right\|_2$ computes the ℓ_2-norm of $\mathbf{W}^{(0)}$.

Apart from the training set $\mathcal{V}_T$, denote the validation set as $\mathcal{V}_V$ and testing set as $\mathcal{V}_{test}$. The intersection between any two of them is empty.

From the perspective of the software implementing GCNs, we assume it outputs the prediction $\mathbf{p}_i, \forall i \in \mathcal{V}$, the loss $\mathcal{L}$ on the node sets $\mathcal{V}_T$ and $\mathcal{V}_V$ at each training step, and the loss on $\mathcal{V}_{test}$ during testing.

B. Metamorphic Relations

To test CNNs processing image data, [9] proposes some MRs by scaling the RGB values of image pixels and permuting the RGB channels. We argue that the attributes of nodes in graph data are similar to the pixels of images, both of which act as the original features to describe the input samples. Hence, we extend two MRs for CNNs [9] to GCNs, leading to:

- MR-1: Scaling the attributes of any nodes by non-zero factors.
- MR-2: Permutation of the attribute dimensions.

GCNs classify a node not only using its own attributes but also the graph structures and the attributes of other nodes, which contribute to the prediction results through a 'message-passing' mechanism across GCN layers. Typically, GCNs output the classification probability of node i as follows:

$$\mathbf{p}_i = f(\{\mathbf{x}_j\}_{j\in\mathcal{V}_i}, G_i = (\mathcal{V}_i, \mathcal{E}_i), \{\mathbf{W}^l\}_{l=0}^{L-1}), \tag{6}$$

where G_i is the subgraph corresponding to the receptive field of node i [12], which is the $(L+1)$-hop neighborhood of node i (including itself) for the GCNs defined in Eq. (1), and $f(\cdot)$ is the function for passing the attributes $\{\mathbf{x}_j\}_{j\in\mathcal{V}_i}$ to node i. We observe that in $\mathcal{G}_i$, only the 1-hop neighbors are connected to node i directly. Hence, for a pair of nodes i and j, we intuitively have their predictions exchanged if we exchange their attributes: $\mathbf{x}_i$ and $\mathbf{x}_j$, and their 1-hop neighbors:

- MR-3: Exchanges of the attributes and neighbors of nodes.

As shown in Eq. (6), to predict the class of node i, $f(\cdot)$ takes $\{\mathbf{x}_j\}_{j\in\mathcal{V}_i}$ and $G_i = (\mathcal{V}_i, \mathcal{E}_i)$ as the inputs, i.e., the nodes and edges out of $\mathcal{G}_i$ do not influence $\mathbf{p}_i$. Then, we propose the following MRs, which do not influence the predictions of the node i if the modifications happen out of the G_i, i.e., the receptive field of node i.

- MR-4: Adding or removing the edges.
- MR-5: Adding or removing the nodes.
- MR-6: Changing the node attributes.

Next, we conduct theoretical analysis and present formal justifications for our MRs. Although the analysis is conducted on the GCNs defined in Eq. (1), they can be extended to other GCN variants trivially because they also follow the 'message-passing' framework. By default, we assume that in the training mode, the variable initialization and dropout are the same for different inputs, for the ease of theoretical analysis.

1) MR-1: Scaling the attributes of any nodes by non-zero factors: Given the node attributes $\mathbf{x}_i^{(a)}, \forall i \in \mathcal{V}$, if we scale them by non-zero factors $\alpha_i \neq 0, \forall i$:

$$\mathbf{x}_i^{(b)} \leftarrow \mathbf{x}_i^{(a)} * \alpha_i, \tag{7}$$

$$\mathbf{X}_a = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \mathbf{W}_a^{(0)} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$\pi = \begin{bmatrix} 3 \\ 1 \\ 2 \end{bmatrix}, \mathbf{\Pi} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{H}_a^{(1)} = \mathrm{ReLU}\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix}\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}\right) = \begin{bmatrix} 8 & 10 \\ 3.5 & 5 \\ 3.5 & 5 \end{bmatrix}$$

$$\mathbf{H}_b^{(1)} = \mathrm{ReLU}\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix}\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 1 & 2 \\ 3 & 4 \end{bmatrix}\right) = \begin{bmatrix} 8 & 10 \\ 3.5 & 5 \\ 3.5 & 5 \end{bmatrix}$$

Figure 4: *An example of MR-2.* The values of the first-layer representations (red) are the same for the original input (a), and the manipulated one (b).

we have the following relations between the outputs corresponding to the inputs a and b respectively:

$$\mathbf{P}_a = \mathbf{P}_b, \mathcal{L}_a = \mathcal{L}_b. \tag{8}$$

An example visualizing MR-1 is shown in Fig. 2.

Proof: GCN systems apply linear normalization on the attribute vectors before feeding them to the neural network [3]:

$$\mathbf{x}_i' = \mathbf{x}_i / \sum_j X_{ij}, \tag{9}$$

where X_{ij} is the jth element of $\mathbf{x}_i$. This step is beneficial to the stability of training, as introduced in [3]. After this normalization, we have:

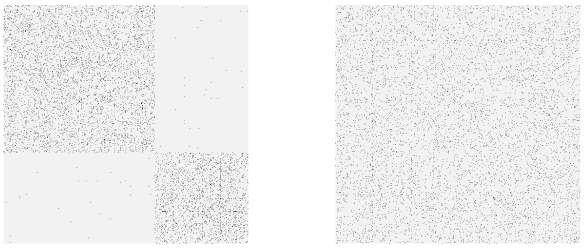
$$\mathbf{x}_i'^{(a)} = \mathbf{x}_i^{(a)} / \sum_j X_{ij}^{(a)} = \alpha_i \mathbf{x}_i^{(a)} / \sum_j \alpha_i X_{ij}^{(a)} \tag{10}$$

$$= \mathbf{x}_i^{(b)} / \sum_j X_{ij}^{(b)} = \mathbf{x}_i'^{(b)}. \tag{11}$$

Then, because the graph structure G is not changed, if $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$, we have $\forall i$:

$$\begin{aligned} \mathbf{p}_i^{(a)} &= f(\{\mathbf{x}_j^{(a)}\}_{j\in\mathcal{V}_i}, G_i = (\mathcal{V}_i, \mathcal{E}_i), \{\mathbf{W}_a^l\}_l) \\ &= f(\{\mathbf{x}_j^{(b)}\}_{j\in\mathcal{V}_i}, G_i = (\mathcal{V}_i, \mathcal{E}_i), \{\mathbf{W}_b^l\}_l) \\ &= \mathbf{p}_i^{(b)}. \end{aligned}$$

Then, we have $\mathcal{L}_a = \mathcal{L}_b$ if $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$. This leads to $\frac{\partial \mathcal{L}_a}{\partial \mathbf{W}_a^{(l)}} = \frac{\partial \mathcal{L}_b}{\partial \mathbf{W}_b^{(l)}}, \forall \mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}$. Before the tth training iteration, we assume $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$. Then, after iteration t, the weights $\mathbf{W}_a^{(l)}(\mathbf{W}_b^{(l)})$ is updated by the gradient $\frac{\partial \mathcal{L}_a}{\partial \mathbf{W}_a^{(l)}}(\frac{\partial \mathcal{L}_b}{\partial \mathbf{W}_b^{(l)}})$. Thus, with $\frac{\partial \mathcal{L}_a}{\partial \mathbf{W}_a^{(l)}} = \frac{\partial \mathcal{L}_b}{\partial \mathbf{W}_b^{(l)}}$, the relation $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$ still holds before the next iteration, i.e., iteration $(t+1)$. Next, since $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$ holds before the first iteration (the same initialization holds for inputs a and b), we have $\mathbf{P}_a = \mathbf{P}_b$, $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$ and $\mathcal{V}_V$ at each training step, and $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_{test}$ during testing. ■



(a) Original Adjacency Matrix (b) After exchanges

Figure 5: *MR-3 exchanges neighbors and attributes of nodes.* The predictions are equivalent but exchanged in the same order as the inputs, whether the GCN is trained and tested on the original or manipulated data.

2) MR-2: Permutation of the attribute dimensions: A permutation can be specified by a permutation vector $\boldsymbol{\pi}$, such that $\pi_i = j$ means the jth element is moved to the ith position. For the simplicity of expression, we introduce the permutation matrix[1] $\mathbf{\Pi}$. For any permutation vector $\boldsymbol{\pi}$, the corresponding $\mathbf{\Pi A}$ is to permute the rows of $\mathbf{A}$ in the order of $\boldsymbol{\pi}$, while $\mathbf{A\Pi}^T$ is to permute its columns in order $\boldsymbol{\pi}$. Note that $\mathbf{\Pi}$ is a unitary matrix.

Given the input attribute matrix $\mathbf{X}_a$, if we permute the attribute dimensions in the order of $\boldsymbol{\pi}$:

$$\mathbf{X}_b \leftarrow \mathbf{X}_a\mathbf{\Pi}^T, \tag{12}$$

we have $\mathbf{P}_a = \mathbf{P}_b$ and $\mathcal{L}_a = \mathcal{L}_b$, if $\mathbf{W}_b^{(0)} = \mathbf{\Pi}\mathbf{W}_a^{(0)}$ holds after the initialization. An example for visualizing MR-2 is given in Fig. 3. Another example showing the details of computation inside GCNs with MR-2 is presented in Fig. 4.

Proof: The first-layer representation corresponding to input b is:

$$\begin{aligned}\mathbf{H}_b^{(1)} &= \sigma(\mathbf{D}^{-1/2}\tilde{\mathbf{A}}\mathbf{D}^{-1/2}\mathbf{X}_b\mathbf{W}_b^{(0)}) \\ &= \sigma(\mathbf{D}^{-1/2}\tilde{\mathbf{A}}\mathbf{D}^{-1/2}\mathbf{X}_a\mathbf{\Pi}^T\mathbf{\Pi}\mathbf{W}_a^{(0)}) \\ &= \sigma(\mathbf{D}^{-1/2}\tilde{\mathbf{A}}\mathbf{D}^{-1/2}\mathbf{X}_a\mathbf{W}_a^{(0)}) \\ &= \mathbf{H}_a^{(1)}\end{aligned}$$

Then, for $l > 1$, assuming $\mathbf{H}_a^{(1-1)} = \mathbf{H}_b^{(1-1)}$ and $\mathbf{W}_a^{(l-1)} = \mathbf{W}_b^{(l-1)}$, we have:

$$\begin{aligned}\mathbf{H}_b^{(l)} &= \sigma(\mathbf{D}^{-1/2}\tilde{\mathbf{A}}\mathbf{D}^{-1/2}\mathbf{H}_b^{(l-1)}\mathbf{W}_b^{(l-1)}) \\ &= \sigma(\mathbf{D}^{-1/2}\tilde{\mathbf{A}}\mathbf{D}^{-1/2}\mathbf{H}_a^{(l-1)}\mathbf{W}_a^{(l-1)}) \\ &= \mathbf{H}_a^{(l)}.\end{aligned}$$

Since $\mathbf{H}_a^{(1)} = \mathbf{H}_b^{(1)}$, we have $\mathbf{P}_b = softmax(\mathbf{H}_b^{(L)}) = softmax(\mathbf{H}_a^{(L)}) = \mathbf{P}_a$, $\mathcal{L}_a = \mathcal{L}_b$ if $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l > 0$ and $\mathbf{W}_b^{(0)} = \mathbf{\Pi}\mathbf{W}_a^{(0)}$. Next, the analysis is as same as that of MR-1. We have $\mathbf{P}_a = \mathbf{P}_b$, $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$ and $\mathcal{V}_V$ at each training step, and $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_{test}$ during testing. ■

[1]https://en.wikipedia.org/wiki/Permutation_matrix

$$\mathbf{A}_b = \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}0&0&0\\0&0&1\\0&1&0\end{bmatrix}\begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} = \begin{bmatrix}0&0&1\\0&0&0\\1&0&0\end{bmatrix}$$

$$\mathbf{X}_b = \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}0&1&1\\1&0&0\\1&0&1\end{bmatrix} = \begin{bmatrix}1&0&1\\0&1&1\\1&0&0\end{bmatrix}, \mathbf{H}_a^{(1)} = \begin{bmatrix}8&10\\3.5&5\\3.5&5\end{bmatrix}$$

$$\mathbf{H}_b^{(1)} = \text{ReLU}\left(\begin{bmatrix}0.5&0&0.5\\0&1&0\\0.5&0&0.5\end{bmatrix}\begin{bmatrix}1&0&1\\0&1&1\\1&0&0\end{bmatrix}\begin{bmatrix}1&2\\3&4\\5&6\end{bmatrix}\right) = \begin{bmatrix}3.5&5\\8&10\\3.5&5\end{bmatrix}$$

Figure 6: *An example of MR-3.* The original input (a) and the permutation order $\boldsymbol{\pi}$ are the same as those shown in Fig. 4. The first-layer representations (red) of the generated input (b) are those of the original one permuted in order $\boldsymbol{\pi}$

3) MR-3: Exchanging attributes and neighbors of nodes: An exchange is formally defined as an action that swaps two elements and does not change the others. It is proven that a chain of exchanges composes a permutation [36].

Considering the input $\mathbf{A}_a, \mathbf{X}_a$, if we permute (a chain of exchanges) the neighbors and features of nodes in the order of $\boldsymbol{\pi}$:

$$\mathbf{A}_b \leftarrow \mathbf{\Pi}\mathbf{A}_a\mathbf{\Pi}^T, \mathbf{X}_b \leftarrow \mathbf{\Pi}\mathbf{X}_a, \tag{13}$$

we have $\mathbf{P}_b = \mathbf{\Pi}\mathbf{P}_a$. Additionally, we have $\mathcal{L}_a = \mathcal{L}_b$ if we permute the labels $\mathbf{Y}_b \leftarrow \mathbf{\Pi}\mathbf{Y}_a$ accordingly. An example of MR-3 is visualized in Fig. 5, while another example showing its computation inside GCNs is presented in Fig. 6.

Proof: Note that $\mathbf{\Pi}$ is unitary. For $l \geq 1$, assuming $\mathbf{H}_b^{(1-1)} = \mathbf{\Pi}\mathbf{H}_a^{(1-1)}$ and $\mathbf{W}_a^{(l-1)} = \mathbf{W}_b^{(l-1)}$, we have:

$$\begin{aligned}\mathbf{H}_b^{(l)} =& \sigma(\mathbf{D}_b^{-1/2}\tilde{\mathbf{A}}_b\mathbf{D}_b^{-1/2}\mathbf{H}_b^{(l-1)}\mathbf{W}_b^{(l-1)}) \\ =& \sigma(\mathbf{\Pi}\mathbf{D}_a^{-1/2}\mathbf{\Pi}^T\mathbf{\Pi}\tilde{\mathbf{A}}_a\mathbf{\Pi}^T\mathbf{\Pi}\mathbf{D}_a^{-1/2}\mathbf{\Pi}^T\mathbf{\Pi}\mathbf{H}_a^{(l-1)}\mathbf{W}_a^{(l-1)}) \\ =& \sigma(\mathbf{\Pi}\mathbf{D}_a^{-1/2}\tilde{\mathbf{A}}_a\mathbf{D}_a^{-1/2}\mathbf{H}_a^{(l-1)}\mathbf{W}_a^{(l-1)}) \\ =& \mathbf{\Pi}\sigma(\mathbf{D}_a^{-1/2}\tilde{\mathbf{A}}_a\mathbf{D}_a^{-1/2}\mathbf{H}_a^{(l-1)}\mathbf{W}_a^{(l-1)}) \\ =& \mathbf{\Pi}\mathbf{H}_a^{(l)}\end{aligned}$$

Because $\mathbf{X}_b = \mathbf{\Pi}\mathbf{X}_a$ and $\mathbf{Y}_b = \mathbf{\Pi}\mathbf{Y}_a$, we have

$$\mathbf{P}_b = softmax\left(\mathbf{H}_b^{(L)}\right) = \mathbf{\Pi}softmax\left(\mathbf{H}_a^{(L)}\right) = \mathbf{\Pi}\mathbf{P}_a, \mathcal{L}_a = \mathcal{L}_b, \tag{14}$$

if $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$. Next, the analysis is as the same as that of MR-1. We have $\mathbf{P}_b = \mathbf{\Pi}\mathbf{P}_a$, $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$ and $\mathcal{V}_V$ at each training step, and $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_{test}$ during testing. ■

4) MR-4: Adding or removing the edges: For the ease of expression, we define the binary diagonal matrix $\mathbf{T}_{\mathcal{V}_x}$ for the node set $\mathcal{V}_x$, with $T_{ii} = 1$ indicating $i \in \mathcal{V}_x$ and $T_{ii} = 0$, otherwise. And we define the function $d(i, j)$ counting the distance between nodes i and j. Then, given node set $\mathcal{V}_x$ and a positive constant l, we define the edge set:

$$\mathcal{E}_{\mathcal{V}_x}^{(l)} = \{(i,j) \in \mathcal{E} | d(k,i) > l \ \&\ d(k,j) > l, \forall k \in \mathcal{V}_x\}. \tag{15}$$

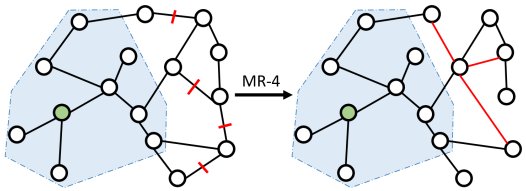


Figure 7: *MR-4 adds/removes edges.* For the considered node (green), adding/removing edges (red) outside its receptive field (blue) does not influence its classification results.

$$\mathbf{X}_a = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}, \mathbf{X}_b = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \mathbf{H}_a^{(1)} = \begin{bmatrix} 8 & 10 \\ 3.5 & 5 \\ 3.5 & 5 \end{bmatrix}$$

$$\mathbf{H}_b^{(1)} = \text{ReLU}\left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \right) = \begin{bmatrix} 8 & 10 \\ 5.5 & 7 \\ 5.5 & 7 \end{bmatrix}$$

Figure 8: *An example of MR-6.* The adjacency matrix is the same as that shown in Fig. 4. The first-layer representations (red) of node 1 remains the same before and after the changes on the attributes of nodes 2 and 3 induced by MR-6, since nodes 2 and 3 are out of the receptive field of node 1.

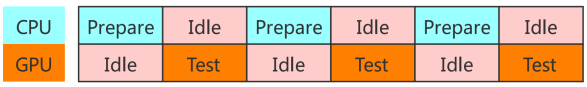


Figure 9: *Metamorphic Testing can be processed by in a sequential system consisting of the loops of the stages: 'Prepare' and 'Test'.* 'Prepare' fits CPUs since it mainly contains logical controls and serial instructions, while 'Test' fits GPUs since it contains intensive computation workload, which can be processed in parallel units of GPUs.

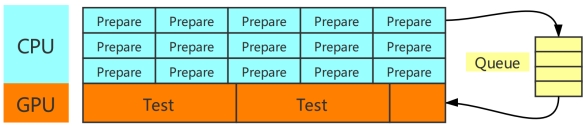


Figure 10: *We propose a pipeline design for Metamorphic Testing, to improve efficiency.* Computation on CPUs and GPUs are processed simultaneously. CPU computations are accelerated by multi-threading. A queue is built to store the data from CPUs and feed them to GPU, so as to stabilize the performance of the system.

And given edge (i,j), we define the node set:

$$\mathcal{V}_{(i,j)}^{(l)} = \{k \in \mathcal{V} | d(k,i) > l \ \& \ d(k,j) > l\}, \tag{16}$$

Given the input $\mathbf{G}_a$, if we add/remove edge $(i,j) \in \mathcal{E}_{\mathcal{V}_T}^{(L)}$, leading to $\mathbf{G}_b$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$, and $\mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_b$ during training. Furthermore, if we add/remove edge $(i,j) \in \mathcal{E}_{\mathcal{V}_T \cup \mathcal{V}_V}^{(L)}$, there is additionally $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_V$ during training. If we add/remove edge $(i,j) \in \mathcal{E}_{\mathcal{V}_{test}}^{(L)}$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_{test}$, and $\mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_b$ during testing. Moreover, during testing, if we add/remove edge (i,j), we have $\mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_b$. An example visualizing MR-4 is given in Fig. 7.

Proof: From Eq. (6), given node set $\mathcal{V}_x$, we know that the changes happening to edge $(i,j) \in \mathcal{E}_{\mathcal{V}_x}^{(L)}$, does not influence $G_k, k \in \mathcal{V}_x$, which is the $(L+1)$-hop neighborhood subgraph of node k, and thus does not influence $\mathbf{p}_k, \forall k \in \mathcal{V}_x$, if $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$. Next, the analysis is similar to that of MR-1. If adding/removing edge $(i,j) \in \mathcal{E}_{\mathcal{V}_T}^{(L)}$ induces $\mathcal{G}_b$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$, and $\mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_{(i,j)}^{(L)}} \mathbf{P}_b$ during training. The other statements also hold with the similar analysis. ■

5) MR-5: Adding or removing the nodes: Given the node set $\mathcal{V}_x$ and a positive constant l, we define the node set:

$$\mathcal{V}_{\mathcal{V}_x}^{(l)} = \{i | d(i,j) > l, \forall j \in \mathcal{V}_x\}. \tag{17}$$

Given node i, and a positive constant l, we define the node set:

$$\mathcal{V}_i^{(l)} = \{j \in \mathcal{V} | d(j,i) > l\}. \tag{18}$$

Given the input $\mathbf{G}_a$, if we add/remove node $i \in \mathcal{V}_{\mathcal{V}_T}^{(L+1)}$, leading to $\mathbf{G}_b$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$, and $\mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_b$ during training. Furthermore, if we add/remove node $i \in \mathcal{V}_{\mathcal{V}_T \cup \mathcal{V}_V}^{(L+1)}$, there is additionally $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_V$ during training. If we add/remove node $i \in \mathcal{V}_{\mathcal{V}_{test}}^{(L+1)}$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_{test}$, and $\mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_b$ during testing. Moreover, during testing, if we add/remove node i, we have $\mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_b$.

Proof: From Eq. (6), given node set $\mathcal{V}_x$, we know that the change that affects node $i \in \mathcal{V}_{\mathcal{V}_x}^{(L+1)}$, does not influence the $G_k, k \in \mathcal{V}_x$, which is the $(L+1)$-hop neighborhood subgraph of node i, and thus does not influence $\mathbf{p}_k, \forall k \in \mathcal{V}_x$, if $\mathbf{W}_a^{(l)} = \mathbf{W}_b^{(l)}, \forall l$. Next, the analysis is similar to that of MR-1. If adding/removing node $i \in \mathcal{V}_{\mathcal{V}_T}^{(L+1)}$ induces $\mathcal{G}_b$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$, and $\mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_b$ during training. The other statements also hold with similar analysis. ■

6) MR-6: Changing the node attributes: Given the input $\mathbf{G}_a$, if we change the attributes of node $i \in \mathcal{V}_{\mathcal{V}_T}^{(L+1)}$, leading to $\mathbf{G}_b$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_T$, and $\mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_b$ during training. Furthermore, if we change node $i \in \mathcal{V}_{\mathcal{V}_T \cup \mathcal{V}_V}^{(L+1)}$, there is additionally $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_V$ during training. If we change node $i \in \mathcal{V}_{\mathcal{V}_{test}}^{(L+1)}$, we have $\mathcal{L}_a = \mathcal{L}_b$ on $\mathcal{V}_{test}$, and $\mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L+1)}} \mathbf{P}_b$ during testing. Moreover, during testing, if we change node i, we have $\mathbf{T}_{\mathcal{V}_i^{(L)}} \mathbf{P}_a = \mathbf{T}_{\mathcal{V}_i^{(L)}} \mathbf{P}_b$. An example showing the computation inside GCNs with MR-6 is presented in Fig. 8.

The proof is as same as that of MR-5.

C. Pipeline for Efficiency

The Metamorphic Testing for the node classifiers can be processed in the sequences of 'Prepare' and 'Test', as visualized in Fig. 9. 'Prepare' consists of generating test cases following MRs, data pre-processing and comparing the outputs obtained in the past iterations, while in the 'Test' stage the program under test (PUT) takes the inputs processed in 'Prepare' to run the GCN model and return the outputs. Typically, 'Prepare' is run on CPUs and 'Test' is run on GPUs, since the former one consists of massive logical controls and serial instructions, while the latter one is compute-intensive and can be conducted in a parallel manner simply. Hence, we design a pipeline to execute the computation on CPUs and GPUs simultaneously, so as to better utilize the computation resources, as shown in Fig. 10. For the computation on CPUs, we use multi-threading techniques to accelerate the computation. Moreover, a queue is used to store the generated inputs from CPUs and feed them into GPUs for the testing, which helps the system to process iterations stably. Empirical results show that our pipeline system achieves much better efficiency than the sequential system.

We also note that an implementation choice is to utilize the multi-threading mechanism offered by Python to build the pipeline. However, limited by the Global Interpreter Lock (GIL) in the Python language, only one thread can be executed at any point in time [37]. In addition, due to the overheads in the Python multi-threading, such as task management, this implementation is found to operate even slower than the sequential one, which is illustrated in our empirical results.

IV. Empirical Results

In this section, we first evaluate the effectiveness of our proposed MRs for detecting implementation bugs. Next, we test the efficiency of our proposed pipelines for implementing our MT approach.

A. Experimental Settings

We conduct Mutation Testing [11] to test the effectiveness of our proposed MRs. Mutation Testing synthetically changes the lines of a program to inject bugs. A mutant is a program containing one bug. A general intuition behind Mutation Testing is that the generated mutants represent the typical errors that the programmers make [11]. Mutation Testing is widely used in testing the effectiveness of Metamorphic Testing [25], [9].

We take the Python implementation of the vanilla GCN [3], JKNet [15], and LGCN [16] as the programs under test (PUT), of which the details are presented in Tab. I. We use the Mutation Testing tool Mutpy [38] developed for Python to generate mutants. For each mutant implementation, we run it with the original data and the data manipulated following our MRs. The outputs are analyzed to check if any MR is violated. If violations exist, we claim that the mutant is 'killed', i.e., the implementation bugs in the program are identified.

```
115.      r_inv = np.power(rowsum, -1).flatten()
115.      r_inv = np.power(rowsum, -2).flatten()
```

Figure 11: *The original code (top) and mutant 9 (below) for GCN.* This mutant breaks the normalization on feature attributes.

In the training mode, even with the same inputs, outputs of the GCN system can vary due to the random initialization and dropout. This is a challenge for MT because MT works by comparing the outputs of subsequent runs on the programs. One common strategy to control randomness is to set a fixed random seed in the program, which makes the outputs deterministic. However, existing work reports that running in GPUs leads to randomness not controlled by the Python seeds [13], [9]. To handle this issue, we conduct Mutation Testing without fixing the random seeds. We set a threshold for the maximum standard deviation $\sigma_{\max}$ of the training/validation losses that a correct program can produce, beyond which the programs are identified to be wrong.

For the comprehensive evaluation, we use three citation networks commonly used on the task of node classification: Citeseer, Cora, and Pubmed [17] and two co-author graphs Coauthor-CS (short as Co-CS) and Coauthor-Physics (short as Co-Phy) [18]. Each of them contains an unweighted adjacency matrix and bag-of-words features. The statistics of these datasets are presented in Tab. II. For each dataset, we do 100 random splits of the the data into train/validation/test sets. For the classes which had more than 100 samples. We choose 20 samples per class for training, 10 samples per class for validation, and 20 samples as test data. For the classes which had less than 100 samples, we chose 20% samples per class for training, 10% samples for validation, and 20% for testing. We repeat each experiment 500 times to report the results.

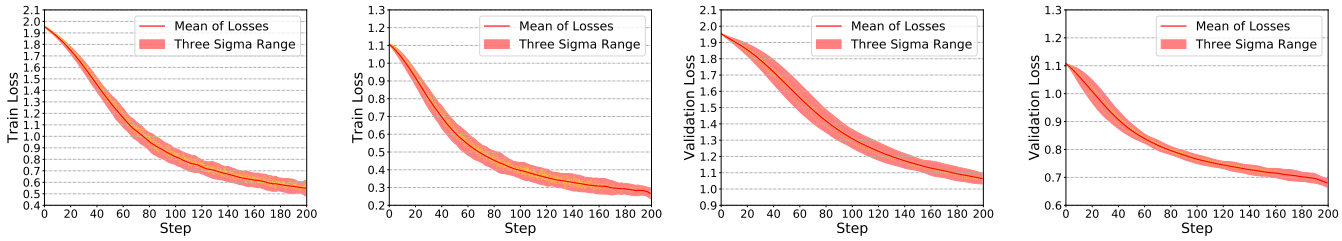
When using the existing programs, we follow the suggested hyper-parameters of the models as the papers suggested originally. All the experiments are conducted on a Linux Server with a 1080ti GPU.

B. Creating the Mutants

We use Mutpy [38] to generate mutants from the implementations of GCN and JKNet implementation. In total, 416 and 523 mutants are created for GCN and JKNet respectively. We discard the mutants giving compile-time errors, run-time errors, equivalent to the original programs, or changing the hyper-parameters. As a result, 27 mutants are left for GCN and 38 for JKNet, of which the types are concluded in Tab. III and IV. Fig. 11 shows an example mutant.

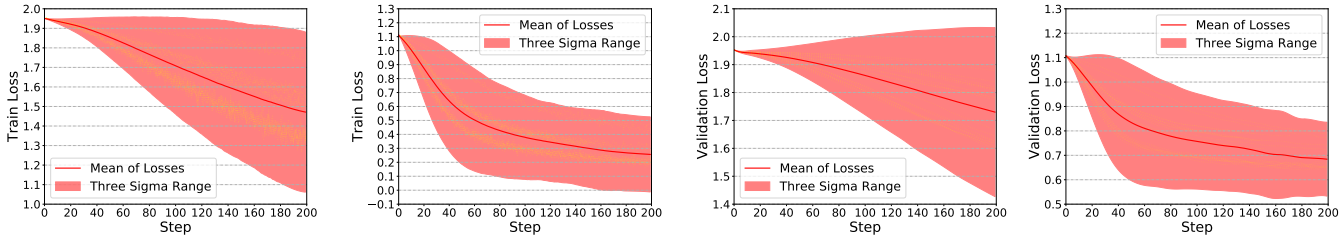
Table I: URLs and commit number of the tested implementations

Model	URL	Commit
GCN [3]	https://github.com/tkipf/gcn	0a505f
JKNet [15]	https://github.com/mori97/JKNet-dgl	63baea
LGCN [16]	https://github.com/divelab/lgcn	46300d



(a) `Cora` (b) `Pubmed` (c) `Cora` (d) `Pubmed`

Figure 12: The training (left) and validation (right) losses of the *normal GCN programs* in different trials without fixed random seeds. Each trial is conducted with the random inputs created by MR-1. Mean and standard deviations at different steps are presented.



(a) `Cora` (b) `Pubmed` (c) `Cora` (d) `Pubmed`

Figure 13: The training (left) and validation (right) losses of the *mutant 9* in different trials without fixed random seeds. Each trial is conducted with the random inputs created by MR-1. Much larger deviations than the losses in Fig. 12 are observed.

Table II: Statistics of the utilized datasets

Dataset	# Nodes	# Edges	# Classes	# Features
`Citeseer`	3,327	4,732	6	3,703
`Cora`	2,708	5,429	7	1,433
`Pubmed`	19,717	44,338	3	500
`Co-CS`	18,333	81,894	67	6,805
`Co-Phy`	34,493	247,962	5	8,415

Table III: Types of the created valid mutants for GCN.

Mutant Type	# Mutants
Change attribute pre-processing	2
Change adjacency pre-processing	4
Change regression loss	5
Change classification loss	5
Change GCN structure	6
Enable dropout during testing	2
Change the labels of data	3
Total	27

Table IV: Types of the created valid mutants for JKNet.

Mutant Type	# Mutants
Change attribute pre-processing	4
Change adjacency pre-processing	5
Change loss functions	8
Change data formats	4
Change JKNet structure	11
Change the data splits	6
Total	38

C. *Effectiveness*

First, we conduct our MT approach with different MRs on both normal implementations and mutants. Note that we run different trials without fixed random seeds. Fig. 12 shows the curves of the training/validation losses versus training steps collected from the correct implementation of GCN with the inputs generated by MR-1, while Fig. 13 shows the curves collected from mutant 9 with MR-1. We observe that the

Table V: The maximum standard deviations $\sigma_{\max}$ of the training/validation losses among different training steps collected from the normal programs are consistently low.

Model	Dataset	The maximum std. among training steps $\sigma_{\max}$
GCN	Citeseer	0.015
	Cora	0.015
	Pubmed	0.014
	Co-CS	0.016
	Co-Phy	0.015
JKNet	Citeseer	0.014
	Cora	0.013
	Pubmed	0.015
	Co-CS	0.015
	Co-Phy	0.014

training and validation loss curves vary slightly on normal programs but much larger on the mutant. We conduct these experiments for the implementations of GCN and JKNet with different datasets. Similar gaps between the normal and abnormal programs are observed. For normal programs, the maximum standard deviation $\sigma_{\max}$ of the training/validation losses among different training steps are reported in Tab. V, all of which are consistently low. Note that the differences of $\sigma_{\max}$ coming from the change of datasets or the network architecture are much smaller than that from the abnormal programs. Thus, given a network architecture, we can set a threshold for the $\sigma_{\max}$ that a correct program can have, which is larger than the $\sigma_{\max}$ that it produces on graph datasets. Then, the implementations holding $\sigma_{\max}$ beyond it are identified to be wrong.

In our experiment, we set the threshold of $\sigma_{\max}$ as 0.03 for both GCN and JKNet to identify the bugs in the training mode, which is larger than the maximum $\sigma_{\max}$ listed in Tab. V. In the testing mode, we only compare the model outputs, e.g., the testing losses, since there is no randomness. For the GCN implementation, the full list of the identified mutants is summarized in Tab. VI. Overall, 81.5% of mutants are identified. Note that the results significantly outperform the MR set consisting of only MR-1 and MR-2 (catching 25.9% mutants), which are extended from the existing MRs for CNNs [9]. Except for the MR set from CNNs, our novel MRs, MR-3 to MR-6, catch 70.4% mutants independently, which takes up 86.4% of all the caught mutants. The results of detecting mutants in JKNet is given in Tab. VII. On average, using one of MRs from 3 to 6 can detect 12.75 mutants, and using one of MRs from 1 to 2 can catch 4 mutants. Overall, there are 84.2% mutants identified for the JKNet implementation.

In addition to synthetic mutants, we apply our method to detect the real-world bugs existing in the existing programs. One bug in the implementation of LGCN is detected, of which the details is shown in Fig. 14 respectively. The reason for the bug in Fig. 14 is that the variables 'pre_indices' and

Table VI: The MT results of our MRs on GCN. The MRs extended from CNNs identify 25.9% mutants (green), while the novel ones that we propose by analyzing the graph data and GCNs identify 70.4% mutants (blue). Overall, 22 out of 27 (81.5%) bugs were caught.

	MR-1	MR-2	MR-3	MR-4	MR-5	MR-6
# Caught Mutants	4	3	9	8	12	11
Catch ratio	25.9%		+70.4% → **81.5%**			

Table VII: The MT results of our MRs on JKNet. The MRs extended from CNNs identify 21.1% mutants (green), while the novel ones that we propose by analyzing the graph data and GCNs identify 71.1% mutants (blue). Overall, 32 out of 38 (84.2%) bugs were caught.

	MR-1	MR-2	MR-3	MR-4	MR-5	MR-6
# Caught Mutants	3	5	9	13	15	14
Catch ratio	21.1%		+71.1% → **84.2%**			

```
14.         pre_indices = indices
15.         candidates = get_candidates(adj, new_add) - indices
16.         if len(candidates) > size - len(indices):
17.             candidates = set(np.random.choice(list(candidates), size-
    len(indices), False))
18.         indices.update(candidates)
```

(a) A faulty example

```
14.         pre_indices = indices.copy()
```

(b) A recommended fix

Figure 14: Our approach detects a real-world bug in the implementation of LGCN.

'indices' share the same object in memory, which leads to incorrect data pre-processing for LGCN. This bug is caught by our MR-4,5,6.

Setting different values of the threshold for $\sigma_{\max}$ can influence the performance of our MT approach. Empirically, we find that a threshold from the range between 0.02 and 0.05 does not reduce the results reported in the above experiments due to the large gaps between the $\sigma_{\max}$ of correct and incorrect programs.

D. Results on Efficiency

To test the efficiency of our proposed pipeline system, we generate random graphs with the average degree being 1000 and the binary node attributes of 1000 dimensions. Each node contains 20 random non-zero attributes. We feed the inputs generated by MR-1 and run the testing loops supported by different systems: the sequential system, the pipeline implemented by multi-threading of Python [37], and our pipeline (implemented with Tensorflow [39]). For

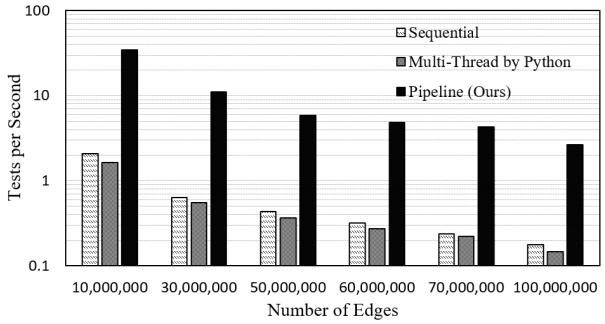


Figure 15: Running speed of Metamorphic Testing on different systems. x-axis is the number of the edges of the input graph. Our pipeline system can achieve at least $10\times$ speedup on the testing speed than the others.

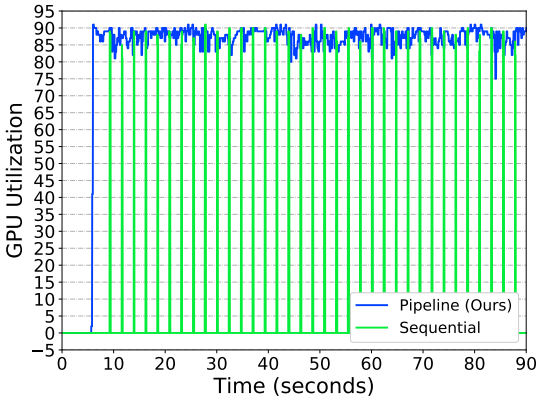


Figure 16: GPU utilization during Metamorphic Testing. The sequential system leaves GPUs idle for long time, while our pipeline keeps high GPU utilization.

the two pipeline implementations, we build 12 threads to generate the test cases efficiently and build a queue of size 20 to accommodate the irregular workload on CPUs and GPUs. The processing speed of different systems is shown in Fig. 15. It is obvious that our pipeline design achieves the best efficiency compared with other methods, and the implementation based on the multi-threading in Python is even slower than the sequential one, due to the overheads in the GIL mechanism of Python multi-threading [37]. We also present the GPU utilization during the running time in Fig. 16. Our pipeline system achieves high GPU utilization consistently, while the sequential one leaves GPUs idle for a long time, which induces serious computation resource waste.

E. Discussion

Our Metamorphic Testing approach offers a tool for debugging the GCN based node classifiers simply and efficiently. Moreover, our pipeline system offers the chance for users to generate test cases and conduct testing simultaneously, which reduces the overall running time of debugging the applications. This framework potentially admits miscellaneous hybrid hardware environments to accelerate computation, such as CPUs, GPUs, and FPGAs. Concretely, our design can coordinate them in an integrated pipeline and conduct the testing iterations efficiently.

Besides the advantages of our design, there still exist some challenging mutants that our MT approach is unable to identify. For example, mutant 231 scales the regression loss function factor by a small factor. Although this is a bug and can degrade the GCN's performance, it does not make GCNs perform extraordinarily different for different inputs generated by our MRs. We hope that the methods proposed later, not limited to Metamorphic Testing, can help identify this kind of subtle changes.

V. Conclusion

Our paper presents six Metamorphic Relations for testing Graph Convolutional Network based node classifiers, with formal justifications for each MR. Besides extending two Metamorphic Relations from CNNs [9] to GCN based classifiers, we propose four novel MRs by analyzing the 'message-passing' scheme and the 'receptive field' of GCNs. We conduct Mutation Testing experiments to test the effectiveness of our Metamorphic Testing approach. Empirical results show that our approach can identify over 80% of the synthetic mutants and two real-world bugs in GCN implementations . We design a pipeline to accelerate the Metamorphic Testing with our proposed MRs. The experiment shows that our pipeline system can achieve more than $10\times$ speedup compared to the sequential system.

Acknowledgments

This paper is supported by NUS ODPRT Grant R252-000-A81-133 and Singapore Ministry of Education Academic Research Fund Tier 3 under MOEs official grant number MOE2017-T3-1-007.

References

[1] S. Kearnes, K. McCloskey, M. Berndl, V. Pande, and P. Riley, "Molecular graph convolutions: moving beyond fingerprints," *Journal of computer-aided molecular design*, vol. 30, no. 8, pp. 595–608, 2016.

[2] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016.

[4] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint arXiv:1901.00596*, 2019.

[5] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Information Fusion*, vol. 42, pp. 146–157, 2018.

[6] P. E. Pope, S. Kolouri, M. Rostami, C. E. Martin, and H. Hoffmann, "Explainability methods for graph convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 772–10 781.

[7] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *arXiv preprint arXiv:1906.10742*, 2019.

[8] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong ..., Tech. Rep., 1998.

[9] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 118–128.

[10] A. Dwarakanath, M. Ahuja, S. Podder, S. Vinu, A. Naskar, and M. Koushik, "Metamorphic testing of a deep learning based forecaster," in *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. IEEE, 2019, pp. 40–47.

[11] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[12] P. Quan, Y. Shi, M. Lei, J. Leng, T. Zhang, and L. Niu, "A brief review of receptive fields in graph convolutional networks," in *IEEE/WIC/ACM International Conference on Web Intelligence-Volume 24800*. ACM, 2019, pp. 106–110.

[13] antares1987, "About deterministic behaviour of gpu implementation of tensorflow," 2017.

[14] jkschin. 2017., "(non) determinism in tf," in *https://github.com/tensorflow/tensorflow/issues/26147*, 2019.

[15] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," *arXiv preprint arXiv:1806.03536*, 2018.

[16] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 1416–1424.

[17] B. London and L. Getoor, "Collective classification of network data." *Data Classification: Algorithms and Applications*, vol. 399, 2014.

[18] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann, "Pitfalls of graph neural network evaluation," *arXiv preprint arXiv:1811.05868*, 2018.

[19] C. Murphy, G. E. Kaiser, and M. Arias, "An approach to software testing of machine learning applications," 2007.

[20] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[21] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[22] D. Selsam, P. Liang, and D. L. Dill, "Developing bug-free machine learning systems with formal mathematics," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3047–3056.

[23] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[24] C. Murphy, K. Shen, and G. Kaiser, "Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles," in *2009 International Conference on Software Testing Verification and Validation*. IEEE, 2009, pp. 436–445.

[25] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2013.

[26] C. Murphy and G. E. Kaiser, "Empirical evaluation of approaches to testing applications without test oracles," 2010.

[27] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[28] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Application of metamorphic testing to supervised classifiers," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 135–144.

[29] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, p. 92, 2019.

[30] J. Ding, X. Kang, and X.-H. Hu, "Validating a deep learning framework by metamorphic testing," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 2017, pp. 28–34.

[31] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 1024–1034.

[32] T. Pham, T. Tran, D. Phung, and S. Venkatesh, "Column networks for collective classification," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[33] C. Zhuang and Q. Ma, "Dual graph convolutional networks for graph-based semi-supervised classification," in *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, 2018, pp. 499–508.

[34] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean *et al.*, "On rectified linear units for speech processing," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 3517–3521.

[35] C. M. Bishop, *Pattern recognition and machine learning*. Springer Science+ Business Media, 2006.

[36] Y. Kosmann-Schwarzbach *et al.*, *Groups and symmetries*. Springer, 2010.

[37] G. VanRossum and F. L. Drake, *The python language reference*. Python Software Foundation Amsterdam, Netherlands, 2010.

[38] K. Halas, "Mutpy 0.4. 0 mutation testing tool for python 3. x," 2017.

[39] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.