On The Effect of Graph Representation of Source Code in Bug Detection

Amir Makhshari
University of British Columbia
Vancouver, BC, Canada
amirosein@ece.ubc.ca

Mifta Sintaha
University of British Columbia
Vancouver, BC, Canada
msintaha@ece.ubc.ca

***Abstract*—With the popularity of using machine learning in source code analysis, the new research trend is to reduce development time through automation. One such branch is enhancing software development through automation using ML techniques. Representing source code as graphs for the machine learning models to capture the structure of the source code has become a popular approach recently due to the ability of graphs to hold extra information about the source code. This paper aims to explore the effectiveness of different graph-based representations and various GNN models in detecting simple one-off errors studied in the DeepBugs paper.**

***Index Terms*—Graph Neural Network, Defect Prediction, Bug Detection, Machine Learning in Software Engineering**

I. Introduction

Software bug is a flaw or error introduced by developers that cause a failure in the system. Bug detection has been one of the most challenging research areas in software engineering research. There are many different features of source code that can be leveraged for bug detection. First of all, developers tend to use meaningful names for variable names. A lot of bug detection tools do not make use of the semantic similarities of tokens in the source code and they only rely on syntactical similarities. Second, the structure of source code can also add a lot of meaningful connections among tokens while a lot of ML-based bug detection tools only consider the surrounding tokens of a given token. Recently, there have been different ML-based approaches to help with this task that leverage each of these meaningful features of the source code [1], [2]. In this work we aim to preserve the semantics of source code tokens while exploring the effect of different graph-based representation of the source code and GNN models on the task of bug detection.

The main contributions of this paper are:

- Using three variations of GNN models for bug detection and studying the results.
- Using two variations of graph-based source code representation for bug detection and studying the results.

We propose the specific graph structure and GNN variation that can achieve the highest accuracy when using graph-based representation for detecting the bug patterns studied by Deepbugs [2].

II. Related Work

A. Automatic Bug Detection

There have been many studies on automatic bug detection which are usually either rule-based approach or a learning-based approach. In this paper, we compare our work with the state-of-the-art learning-based approaches for automatic bug detection. Bugram [3] uses n-gram models to rank methods and then extracts the top-ranked methods as the buggy method. Token sequences are ranked by probability The low probability token sequences are marked as a bugs. Pradel et. al proposed Deepbugs [2], which is a deep learning technique that uses a name-based bug detection approach for detecting three kinds of bugs. Their tool is capable of detecting one-off errors like wrong operator or operand, and swapped method arguments. The approach uses the semantic information conveyed by the identifier names to learn an embedding that maps identifiers to vectors using Word2Vec [4] neural network. From both these approaches, our study differs in deep learning technique of bug detection. Similar to DeepBugs, we use the same goal and dataset for our task but we use a graph based learning model R-GCN whereas their study uses a sequence based RNN model to detect bugs. Li et. al [5] uses Program Dependence Graph (PDG) and Data Flow Graph (DFG) as global context for bug detection to connect the buggy method with other relevant methods. Their approach uses an attention-based Gated Recurrent Unit (GRU) layer to encode and emphasize the order of the nodes in an AST path and an attention Convolution layer for the final classification. The context of their approach is much larger than ours, however, this study focuses more on the effectiveness of a graph representation learning approach due to its increasing popularity and similarity to AST of source code.

B. Graph Representation of Source Code

Graph representation of source code has been gaining popularity due to the presence of deep semantic information provided by the relations between nodes. Allamanis et. al. [6] put forth the insight that graphs are able to leverage the syntactic and semantic relations between the nodes via edges and also consider long-range dependencies. This provided us with a strong motivation to verify the effect of graph representation learning on the task of bug detection. Their

study were on detection of *VarMisuse* and prediction of *VarNaming* tasks whereas ours considered bug detection as a task for verifying the effect of graph representation learning. Hoppity [1] uses graph representation with additional data flow edges and a GNN model for automatic program repair while our study focuses on bug detection and effectiveness of graph representation. W. Wang et. al. [7] used heterogeneous graphs for representing source code on the task of code comment generation and method naming. Their study compared heterogeneous graph representation performance for two tasks and found that it outperformed the baselines. Our study revealed that for simple one-off bugs, however, graph representation learning for both homogeneous and heterogeneous graphs is not as effective as sequence-based approach due to the simple syntactic changes in the graph. We believe that for more complex bugs, graph-based representation and learning will be more effective when coupled with sufficient context.

III. Background

In the literature, there are various studies of using graph representation and GNN models for bug detection and program repair. Graphs of source code are often generated by the augmentation of Abstract Syntax Tree (AST) of the source code with data-flow or control-flow edges. The addition of these edges retain a lot of semantic information. A very recent paper introduced Hoppity [1], which handles not only bug localization, but also generates patches of the buggy code of JavaScript code. Hoppity uses a GNN model to map the program graph into a representation in a fixed dimensional space. It uses a LSTM based GNN model to give out a fixed version of the code which is a learned graph. However, Hoppity only focuses on bug fixes, whereas we want to focus on only bug detection. We also investigated another paper by Y. Wang et al. [8] which detects static bugs in popular Java projects but it uses a RNN model as the representation learning for the source code. For bug classification tasks, it uses a GNN model.

From these papers, we realized that it is hard to get a sense of the better program representation. They are also not compared against the same setup - some target specific languages and some use varying representation and classification models. Therefore, in this paper, we aim to investigate the effect of graph representation of source code on the task of bug detection. The bugs that we detected are the one-off bugs in DeepBugs [2], which include wrong binary operator, wrong binary operand and swapped arguments in functions. The graph types and models used in this approach are outlined below:

A. Graph Types

In this paper, we conducted studies on the different types of graphs to see its effect on bug detection. The two types of graphs used in this study are shown in Figure 1.

Homogeneous Graphs. Homogeneous graphs are graphs that contain the same type of nodes and all edges represent relationships of the same type. Figure 3 shows the homogeneous graphs for each bug types where each node is considered same and each edge is assumed to have the same relationship type.

Heterogeneous Graphs. Heterogeneous graphs are graphs that contain different types of nodes and edges. The different types of nodes and edges tend to have different types of attributes that are designed to capture the characteristics of each node and edge type. As shown in Figure 4, the heterographs for the three bug types have many different relation types to each of the nodes.

B. GNN Models

To classify the bugs, we use three different models, where two of them are implemented on homogeneous graphs and remaining on heterogeneous graphs

Graph Convolutional Network (GCN). GCN is a type of Convolutional Network that works on graphs and take advantage of their structural information. A node in a graph can send or receive messages with its connected neighbours and this phenomenon is called message passing. Figure 1 illustrates the graph message passing between neighbouring nodes. The node features are aggregated by taking the average of its neighbouring features and passed through a Neural Network to produce a new vector. Mathematically, GCN follows the formula below:

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

$H^{(l)}$ is the lth layer in the network, σ is the non-linearity, and W denotes the weight matrix for this layer. D and A, as commonly seen, represent degree matrix and adjacency matrix, respectively. The $\sim$ is a kind of re-normalization where a self-connection is added to each node of the graph, and build the corresponding degree and adjacency matrix. The shape of the input $H^{(0)}$ is $N \times D$, where N is the number of nodes and D is the number of input features [9].

Graph Attention Network (GAN). In GCN, a graph convolution operation outputs the normalized sum of the node features of neighbours. However, GAN introduces the attention mechanism as a substitute for the normalized convolution operation.

$$z_i^{(l)} = W^{(l)}h_i^{(l)} \tag{2}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)}|z_j^{(l)})) \tag{3}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in\mathcal{N}(i)}\exp(e_{ik}^{(l)})} \tag{4}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right) \tag{5}$$

Equation (2) is a linear transformation of the lower layer embedding and learnable weight matrix $W^{(l)}$. In equation (3), a pair-wise un-normalized attention score between two neighbors is computed. It first performs a concatenation of the z embeddings of the two nodes, then takes a dot product of it and a learnable weight vector $\vec{a}^{(l)}$, and applies a LeakyReLU

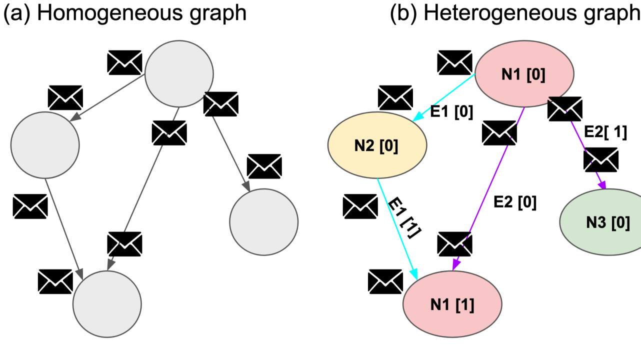

Figure 1: This figure shows an example of how nodes perform message passing throughout the graph. (a) In homogeneous graphs all nodes and edges are from the same type. (b) In heterogeneous graphs, there are categories of nodes (N1, N2, N3) and categories of edges (E1, E2). Each node or edge has a certain index in their category. For example, N1[0] means the first node of the N1 node category. The message being passed is the feature vector of the nodes.

in the end. Finally a softmax is applied to normalize the attention scores of node's incoming edges and the embeddings are aggregated together scaled by the attention scores. In this paper, both GCN and GAN are used for classifying homogeneous graphs [9].

Relational Graph Convolutional Network (R-GCN). R-GCN is used on heterogeneous graphs where the nodes and edges are of different types. In R-GCN, different edge types use different weights and only edges of the same relation type are associated with the same projection weight $W^{(l)}$. The hidden representation of entities in $(l+1)^{th}$ layer in R-GCN is formulated in the following equation:

$$h_i^{l+1} = \sigma\left(W_0^{(l)} h_i^{(l)} + \sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)}\right) \quad (6)$$

Here N_i^r denotes the set of neighbor indices of node i under relation $r \in R$ and $c_{i,r}$ is a normalization constant. However, this is not scalable for graphs with very high relational data.

$$W_r^{(l)} = \sum_{b=1}^{B} a_{rb}^{(l)} V_b^{(l)} \quad (7)$$

So a basis decomposition is applied as outlined in the above equation where the weight $W_r^{(l)}$ is a linear combination of basis transformation $V_b^{(l)}$ with coefficients $a_{rb}^{(l)}$.

IV. METHODOLOGY

Our goal is to study the effect of graph representation of source code in the task of bug detection. As previous studies [2] have shown, supervised machine learning approaches can be used for bug detection and perform reasonably good. In order to narrow down the problem, we focus on single line buggy or correct code and we consider only three certain bug patterns used in [2]. Focusing only on this limited space, allows us to study the effect of graph representation in a more fine-grained manner. To this end, we address the following research questions in this work:

- `RQ1`: How accurate can graph representation of source code be for detecting the bug patterns studied in DeepBugs [2]?
- `RQ2`: Which types of GNN models and graph structures are more helpful in bug detection??
- `RQ3`: How do different graph representations perform in detecting different bug patterns?

In order to answer these questions, we followed the steps described in Figure 2. Our dataset, model, and source code is available in our replication package [10].

A. Dataset

In this section we describe how we collect data and discuss some features of the dataset we use.

1) Data Collection: This step includes collecting a large code corpus that will be used for training and validation of our proposed automatic bug detector. Since we want to focus on supervised learning, we need to have a labeled dataset with each code snippet labelled as *buggy* or *correct* and use them for training.

For this step, we use a JavaScript labelled dataset used in [2], which also has been used by other studies. This dataset consists of the AST of buggy and correct single line statements with their labels. As observed by previous studies [11], a huge amount of existing code is very likely to be correct. Thus, in this dataset, a big corpus of code has been extracted from GitHub and labeled as correct. However, the buggy code is created artificially by simple code transformations that each meant to represent a certain bug pattern. We used 150K JavaScript files and folded the dataset such that 80% was used for training and 20% for testing.

2) Bug patterns: In this section, we briefly introduce the bug patterns we have studied in this paper and describe the features they have in our dataset.

Swapped arguments. This bug pattern is about the use of arguments in JavaScript function calls. In our dataset, all function calls with two or more arguments have been added to the correct examples. We have to note that for function calls with more than two arguments, only the first and the second arguments have been considered. Below we can see a correct example of these extracted function calls:

```
this.retry_(id, xhrIo)
```

The wrong examples are generated by swapping the order of first and second arguments:

```
this.retry_(xhrIo, id)
```

In addition, for each data point, some additional contextual data are added to the data stored for that function call example in the dataset. First of all, besides the called function (callee), the base object (base) is added if the call is a method call. Otherwise, an empty string is considered as the base node. Also, the formal parameters of the callee is considered. Again, empty strings are added if it is unavailable. Additionally, types

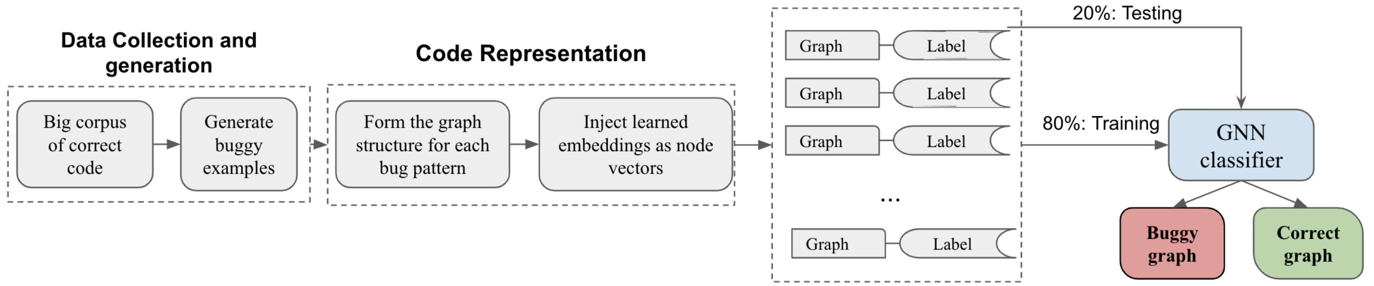

Figure 2: Overview of our approach

of first and second arguments are also added. If the arguments are values other than literals, an empty strings is added as the argument type.

Wrong binary operator. The next two bug patterns are about mistakes in writing binary operations. Consider the below code snippet as the correct example of a binary operation:

```
for(var i=0;i<len;i++)if(validBits>=8)
```

Wrong binary operator is a mistake related to using the wrong operator in binary operations. In the below example of wrong binary operator, the correct >= operation is replaced by another operator which is >.

```
for(var i=0;i<len;i++)if(validBits>8)
```

In the dataset we use, wrong binary operator examples are created by replacing the correct operator with a random operator.

Wrong binary operand. Wrong binary operand is a mistake related to using the wrong left or right operand in binary operations. If we consider the correct binary operation discussed for the previous bug pattern, in the example below, the correct *validBits* operand is replaced by another operand which is *bitsSoFar*.

```
for(var i=0;i<len;i++)if(bitsSoFar>8)
```

In our dataset, wrong binary operand examples are created by replacing the correct operand with a random operand occurring in the same file. In this example, *bitsSoFar* occurs in previous lines in the code and is chosen as the wrong operand. We should note that the operand to be mutated (right or left operand) will be chosen randomly.

In addition, some contextual data are also extracted for each binary operation in our dataset. The parent (`IfStatement` in this case) and grand parent (`ForStatement` in this case) are extracted from the AST of the subject binary operation line in the code. In addition, the operand types are also added to the data for each data point, if their types are known. Since JavaScript is a loosely typed language, some operand types are left as *unknown* in our dataset.

B. Graph Representation of Source Code

The most prominent bug detection approaches such as [2] use sequential representation of source code. However, in this study, we represent source code as a graph and *learn* a representation for the graph that can help GNN models in detecting buggy and correct graphs. In order to do that, we follow below steps:

Initial graph representation. We start with constructing an initial graph representation for each code snippet. Our dataset consists two types of code statements: binary statements and function call statements. For each of these statements we use a dedicated graph structure.

As it is discussed in section III, there are two main approaches to build graphs. We implemented both homogeneous and heterogeneous graph structures of the source code to study the effect of graph structures on the results. Thus, we have a dedicated homogeneous and heterogeneous graph for binary operations and for function calls. As an initial structure for graphs, we started from the AST of each code snippet. Then, similar to previous studies [1], [6], [8] that embed the AST with some additional nodes and edges, we tried to follow the same approach by adding meaningful connections between nodes. Our goal for adding or removing any connection is to leverage the message passing process to achieve better accuracy in bug detection.

Figures 34 show how we construct homogeneous and heterogeneous graphs for all binary operations and function calls. All the graphs show the correct version of the code and the red circles in each bug pattern graph show the nodes that should change in the graph in order to represent that certain bug pattern.

Homogeneous graph of code. As Figure 3 shows, most homogeneous graphs are similar to the AST version of the code statements. As it is discussed in section III, none of the nodes or edges have any sort of label. Thus, in these graphs, all nodes and edges are the same except the node features which is the Word2Vec embedding of the token assigned to that node. As the red nodes show, all three bug patterns would have the same structure as the correct version with only changing the features of one node in wrong binary operator graph, two nodes for wrong binary operand graph, and four nodes for swapped arguments graph.

Heterogeneous graph of code. Figure 4 depicts how the homogeneous graphs are evolved to be heterogeneous. One important change that we applied, was making the nodes and

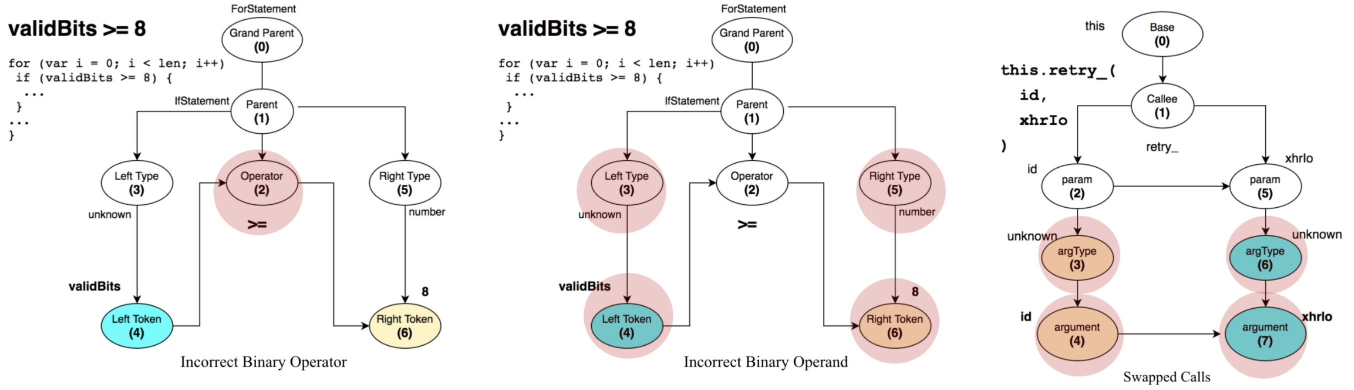

Figure 3: **Homogeneous Graph**

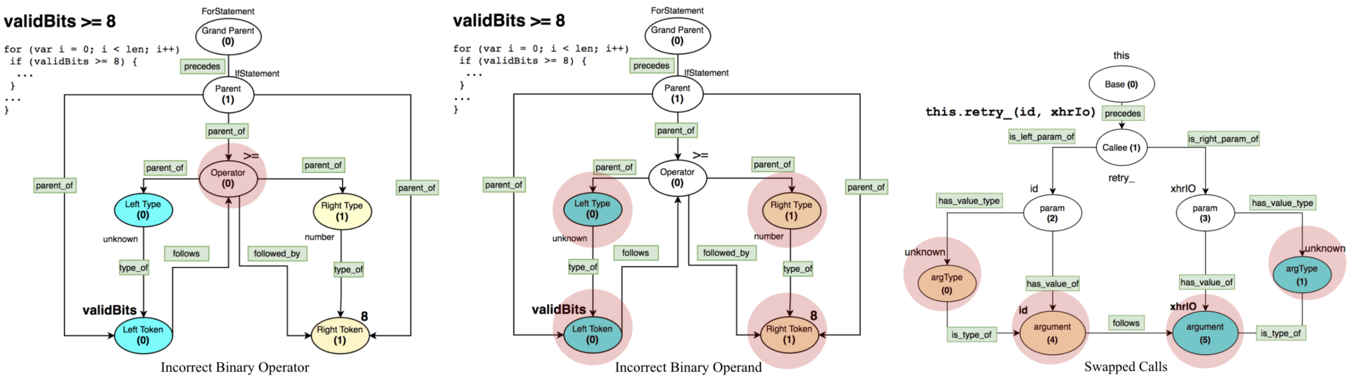

Figure 4: **Heterogeneous Graph**

connections meaningful, by creating categories of nodes and edges by adding meaningful labels to them.

For node categories, in binary operation graphs, we put the parent and grand parent nodes to the same category of parent[0] and parent[1]. We did the same for the base and callee of function calls by grouping them as callee[0] and callee[1]. This way the model interprets these nodes as they are both related to the same category. We followed the same approach for binary operation tokens, their types, and function arguments, their types, as well as function parameters in method calls. Also, regarding edge categories, for all bug patterns, meaningful labels such as *parent_of*, *type_of*, *follows* and *followed_by*, *parent_of*, and *is_param_of* have been added to edges.

Also, we removed some edges and replaced them with other edges. First of all, regarding binary operations, we observed a stronger relation between token types and the binary operator compared to token types and parent node in our dataset. Thus, we altered the binary operation graphs accordingly. First, a new edge from the token types to the operator has been added. This change helped our model in detecting wrong binary operator bugs. Also, the edges from the parent node to the token type nodes have been removed as they could mislead the model to less important aspects. Since usually both left and right token types are similar in correct versions, it is easier for the model to distinguish an invalid type just by the node's own features and without looking at the parent node. Also, as we investigated the binary operation examples, we could see a fairly reasonable relation between the parent node and the left and right token. For example, depending on whether the parent node is a *if* or a *while* statement, the actual tokens usually will be changed based on them. Thus, we concluded that adding two new edges that connect the parent to the left and right tokens might be helpful.

There are some edge additions and deletion on the graph of method calls too. First of all, we removed the edge that connects two parameters of the function as it is not so related to the existence of any bug based on the examples we observed in the dataset. In addition, we connect the method parameters to called arguments as it was only connected to types of called argument before. Our empirical investigation showed that function parameters and called arguments can be very related. In section V, we show how all these changes improve the performance of our model.

Node embedding. Since machine learning models rely on

vector presentation of the input, we first embed graphs' nodes with a previously learnt vector of real numbers. The goal in this step is to preserve nodes' semantics so that nodes with similar properties, such as lexical or syntactical similarities, would lie closer to each other in the vector space. We used the same embedding as the Deepbugs study uses for each identifier and also each literal [2]. Also, whether each token is an identifier (ID) or a literal (LIT) is embedded in the representation of that token.

For identifiers, the embeddings are learnt via CBOW variant of Word2Vec [4], which trains a neural network that can predict a word via its' sorounding tokens. For this training process, the vocabulary has been limited to 10,000 tokens by discarding infrequent tokens. Also, for the number of surrounding tokens to consider, the size of 20 (10 left and 10 right) has been chosen.

However, we have to note that for identifier types (such as *boolean* or *int*) and also AST node types (such as *Call expression* or *While statement*), a one-hot-encoding representation has been used.

Graph representation learning and classification. This is where *representation learning* takes place. In other words, a machine learning model takes the graphs alongside the nodes' embedding vectors and makes the representation of each node better based on the graph connections. This step is done through a process called "message passing" in Graph Neural Networks which will ultimately cause each node to consider a fair number of structurally connected nodes in its embedding. More details about this step for the GNN models we used have been discussed in section III.

After having the perfect graph representation for labelled each code snippet, our GNN implementation uses this representation as well as graph-level labels for training a bug detection classifier.

C. Implementation

We implemented our approach on top of the Deepbugs implementation. We start with Deepbugs' correct corpus of code and as the buggy code snippets are generated, we form and save the graph structure on-the-fly for both correct and buggy versions. We use the DGL library [9] to implement the graph structures and classifiers. We use a batch size of 100 graphs at a time with 16 hidden layers for each of the classifiers used in this study. The input feature length is 200 as the Word2Vec learned tokens extracted from DeepBugs [2] dataset was of the same length. The operator, node type and type vectors were of length 30 which were then padded to keep up with the feature vector length of 200. The learning rate was about 0.005 and the models were trained on 16GB Intel Core i7 processor using the CPU. The training for each of the models was done in 30 epochs which takes around 20 minutes to complete.

V. Evaluation and Discussions

To address the research questions, we followed the methodology described in the previous section to measure the accuracy of the graph-based bug detection model. The models are given two kinds of graphs - one for binary operation and another for function calls. As separate models are used to classify each of the graphs, we report the accuracy of each of the model's accuracy in classifying the bugs. We also report the accuracy of Homogeneous graph representation of source code which will be discussed further in next sections. In the following sections, we use g_{buggy} and $g_{correct}$ to denote the buggy and correct graphs.

A. RQ1. How accurate can graph representation of source code be for detecting the bug patterns studied in DeepBugs [2]?

For classifying the bug patterns, we formulated it as a graph classification task to represent each graph as g_{buggy} and $g_{correct}$. We used separate models for each of the bug patterns since using the same model for classifying three bug patterns and their correct versions was not yielding a good accuracy. Using separate models also makes it a fair comparison with DeepBugs [2] where they reported their accuracy for each bug pattern. We also used the accuracy reported for Word2Vec [4] embeddings in DeepBugs. We found that using random embeddings yielded a lower accuracy, therefore, we used the accuracy of WordtoVec embeddings in DeepBugs as the baseline for comparison.

We report the accuracy of our approach and DeepBugs approach in Table I. As shown in the table, for R-GCN classifier on Heterographs, the bug pattern of "Wrong Binary Operator" gave the highest accuracy (80.13%) among the three bug patterns, followed by "Wrong Binary Operand" (73.17%) and the lowest accuracy for "Swapped Arguments" (58.07%).

Compared to Deepbugs, the accuracy of the R-GCN model is lower for all the bug patterns. One reason for the lower accuracy can be using a structure dependent classifier for predicting bugs. As shown in Figure 4, both g_{buggy} and $g_{correct}$ use the same graph structure but the only aspect that is different is the feature vector changed in the buggy nodes denoted by the red dots, while the graph structure remains exactly the same. Therefore it is evident from the results that, for detecting simple one-off errors as used in DeepBugs [2], graph-based representation cannot accurately help in detecting bugs compared to a sequence-based approach.

B. RQ2. Which types of GNN models and graph structures are more helpful in bug detection?

In Table I, we try to show the type of graph representation that is suitable for source code representation of bugs by reporting the accuracy of both graph types - homogeneous and heterogeneous graph representations of the three bugs. In the homogeneous graph, each of the nodes are considered the same type derived from the basic AST representation of the bug pattern, with edges flowing from parent to child nodes. The only extra edge added in homogeneous graph is the edges that link the binary expression. Initially, we ran the GCN model on the graphs and found the accuracy was very low with "Swapped Arguments" being the highest (50.89%) followed by

Table I: Accuracy of GNN models on different bug patterns

Classifier	Graph Type	Wrong bin operand	Wrong bin operator	Swapped arguments
GCN	Homogeneous	49.81%	50.00%	50.89%
GAN	Homogeneous	50.00%	50.00%	50.00%
R-GCN	Heterogeneous	**73.17%**	**80.13%**	**58.07%**
Deepbugs NN	n/a (sequence-based)	89.06%	92.1%	94.7%

"Wrong Binary Operator" (50%) and the lowest for "Wrong Binary Operand" (49.81%). We believed that since GCN gives structural attention to the graph representation, and the graph structures were the same in g_{buggy} and $g_{correct}$, it was not able to accurately classify the bugs. To prove this hypothesis, we used a GAN model which is feature dependent as opposed to GCN which is structure dependent. However, we only achieved a 0.2% improvement over GCN for "Wrong Binary Operand" and a 0.89% decrease in accuracy for "Swapped Arguments". From the results, it makes sense that homogeneous graphs are not the correct way to represent source code AST of bugs. Additionally, homogeneous graphs have only one type of relation between the nodes, whereas heterogeneous graphs are able to semantically distinguish the edge types from source node to destination node. This is also proven by the fact that each nodes in an AST are of different types containing variables or control flow, and each edges signify different types of relationship between the nodes.

Furthermore, in heterogeneous graphs, we found that adding meaningful edges between the nodes can increase the accuracy. While the AST of the buggy line is taken as the initial structure, we added meaningful relations for connecting the nodes to one another. For instance, in Figure 4, the graphs for "Wrong Binary Operator" and "Wrong Binary Operand" have a *grand-parent* precedes the *parent* node, while the *parent* node is the parent of all left, right tokens and operator in a binary expression. We also added separate relations to define the type to token relation and connect the values with a *follows* and *followed_by* edge labels. Each of the edge label types semantically defines the relation between the nodes, which yielded an accurate classification. This hypothesis is also supported by [1] where special edges like *ValueLink* and *SuccToken* were added to the g_{buggy} and $g_{correct}$ to connect the node values and leaf nodes.

C. RQ3. How do different graph representations perform in detecting the subject bug patterns?

To address *RQ3*, we highlight the steps to achieve an optimum graph representation of bug patterns.

1) The Role of AST-based Context: As shown in the results, we found that the bug pattern "Swapped Arguments" had the lowest accuracy (58.07%) for classifying bugs, even though in DeepBugs [2], it had the highest accuracy. One reason for such accuracy can be the role of context in the graph representations. In the g_{buggy} and $g_{correct}$ of "Wrong Binary Operator" and "Wrong Binary Operand" shown in Figure 4, we can see two additional nodes besides the affected code statement, such as *grand-parent* and *parent* nodes. These nodes hold the information of the surrounding statement nodes of the buggy line, whether or not the statement is within an `IfStatement`, `ForStatement`, `ReturnStatement` etc. In DeepBugs dataset, the context is provided at two levels which was why the graphs had *parent* and *grand-parent* nodes types only. However, in the "Swapped Arguments" graph, we can only see the method signature and the values passed, with no information about the enclosing node types in the statement. This intuition also holds in a real life scenario where a programmer would require context of the surrounding node to understand the semantic nature of the statement. Therefore, with no context, the GNN model is not able to accurately classify g_{buggy} and $g_{correct}$.

2) The Role of Graph-level Connections: As discussed in *RQ2*, heterographs provide the ability to distinguish the types of relations between the nodes. Therefore, adding meaningful relations is key to represent the AST of the buggy or correct statement in graphs. Initially, we tried to connect the nodes with a generic relation of *precedes* connecting from the source to destination nodes of the AST. The only extra relation we added was the nodes that follow one another in a single statement. However, this yielded a really low accuracy as shown in Table II.

We tried adding more edges to the graph with the goal of increasing accuracy, but this actually lowered the bug detection ability of the GNN model as shown in Table II. We can see that with complex relation types, the accuracy of all three bug patterns are slightly higher than the simple relations but lower than the meaningful relation types.

VI. Limitations

Since we re-use Deepbugs dataset and leverage Word2Vec embeddings, the limitations of these tools apply to our study as well. First of all, in the dataset, tokens are represented as

Table II: Accuracy of R-GCN model for heterographs in different relation types

Relation Types	Wrong bin operand	Wrong bin operator	Swapped arguments
Simple	58.12%	48.07%	50%
Complex	62.02%	50.13%	55.28%
Meaningful	**73.17%**	**80.13%**	**58.07%**

Graph without meaningful edges
Accuracy on incorrect operand: 58%
Accuracy on incorrect operator: 48%

Graph with meaningful edges
Accuracy on incorrect operand: 73%
Accuracy on incorrect operator: 80%

Graph with too many edges
Accuracy on incorrect operand: 62%
Accuracy on incorrect operator: 50%

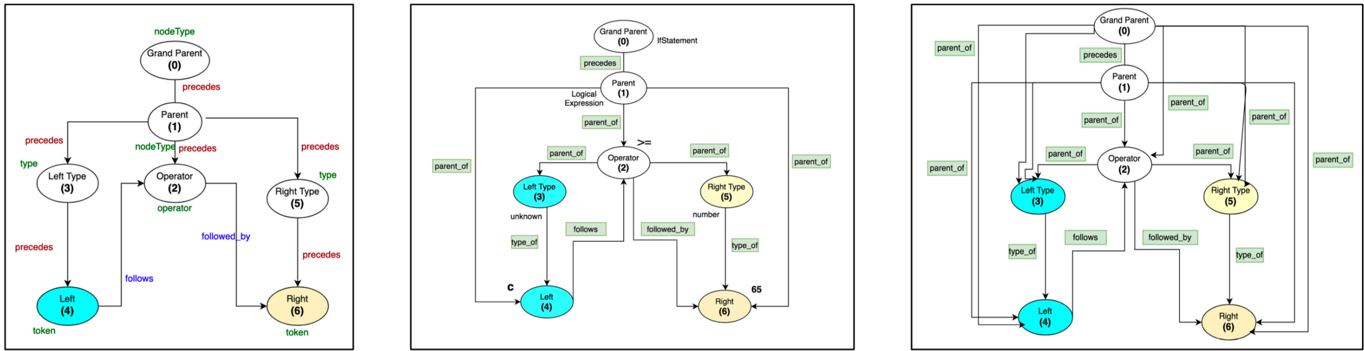

Figure 5: The role of graph-level connections on detecting binary operation bugs.

whole symbols and this might cause some missing semantic information. For example, the tokens *newArray* and *oldArray* are not parsed more and they have different embedding vectors while they share a lot in common. In addition, the vocabulary is limited to 10,000 frequent tokens which can cause the model to not differentiate infrequent tokens from each other. Additionally, like all other ML-based approaches, our results are subject to over-fitting to dataset as well. However, we tried to overcome this issue by using separate training, evaluation, and test sets. This issue can be further resolved by using k-fold cross validation.

Another limitation of our work is to be reliant on one-line code examples. This prevents our findings to be generalized for more realistic contexts. This means we cannot conclude whether GNN models or graph-based representations are better for bug detection compared to RNN models and sequence-based representation. Some factors that are discussed in this paper, such as the amount of contextual data, can be further examined to see whether they can make graph-based approaches perform better even on these simple bug patterns.

VII. Conclusion and Future Work

From this study, we can conclude four things. Firstly, heterogeneous graphs are the way to represent source code compared to homogeneous graphs, as they allow us to leverage the rich syntactic and semantic relations between the nodes and edges in the AST. Secondly, meaningful relations can increase the accuracy of a R-GCN model in classifying bugs and understanding the program structure. Thirdly, context plays a crucial role in improving the accuracy of a model for bug detection, as the intuition hold in a real life scenario when developers use the surrounding context to debug a code. Finally, for simple one-off bugs like wrong binary operator, operand and swapped arguments, graph representation may not be a good way to represent source code with the given context compared to previous sequence-based approaches.

Our study suggests exploring some ideas that might lead into new results and findings. First, exploring the performance of our approach when more context of the source code is given in the dataset can give us a good understanding of when to use graph-based representations and when not to use them. Second, it would be interesting to see how the results can improve for other types of bugs rather than the three simple bug patterns studied here to verify the effectiveness of graph representation further.

References

[1] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning graph transformations to detect and fix bugs in programs,” in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[2] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[3] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, “Bugram: Bug detection with n-gram language models,” *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 708–719, 2016.

[4] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” *arXiv preprint arXiv:1310.4546*, 2013.

[5] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, “Improving bug detection via context-based code representation learning and attention-based neural networks,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360588

[6] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” *arXiv preprint arXiv:1711.00740*, 2017.

[7] W. Wang, K. Zhang, G. Li, and Z. Jin, “Learning to represent programs with heterogeneous graphs,” 2020.

[8] Y. Wang, F. Gao, L. Wang, and K. Wang, “Learning a static bug finder from data,” *CoRR*, vol. abs/1907.05579, 2019. [Online]. Available: http://arxiv.org/abs/1907.05579

[9] *DGL Library*, March 2021, https://www.dgl.ai.

[10] M. Sintaha and A. Makhshari, *Replication Package*, April 2021, https://github.com/msintaha/BugClassificationWithGNN.

[11] P. Bielik, V. Raychev, and M. Vechev, “Phog: probabilistic model for code,” in *International Conference on Machine Learning*. PMLR, 2016, pp. 2933–2942.