*ECAI 2023*
*K. Gal et al. (Eds.)*

*doi:10.3233/FAIA230468*

# Graph Neural Networks for Mapping Variables Between Programs

**Pedro Orvalho** [a;*], **Jelle Piepenbrock** [b,c], **Mikoláš Janota** [c] **and Vasco Manquinho** [a]

[a]INESC-ID, IST, Universidade de Lisboa, Portugal
[b]Radboud University Nijmegen, The Netherlands
[c]Czech Technical University in Prague, Czechia
ORCiD ID: Pedro Orvalho https://orcid.org/0000-0002-7407-5967, Jelle Piepenbrock https://orcid.org/0000-0002-8385-9157, Mikoláš Janota https://orcid.org/0000-0003-3487-784X, Vasco Manquinho https://orcid.org/0000-0002-4205-2189

**Abstract.** Automated program analysis is a pivotal research domain in many areas of Computer Science — Formal Methods and Artificial Intelligence, in particular. Due to the undecidability of the problem of program equivalence, comparing two programs is highly challenging. Typically, in order to compare two programs, a relation between both programs' sets of variables is required. Thus, mapping variables between two programs is useful for a panoply of tasks such as program equivalence, program analysis, program repair, and clone detection. In this work, we propose using graph neural networks (GNNs) to map the set of variables between two programs based on both programs' abstract syntax trees (ASTs). To demonstrate the strength of variable mappings, we present three use-cases of these mappings on the task of *program repair* to fix well-studied and recurrent bugs among novice programmers in introductory programming assignments (IPAs). Experimental results on a dataset of 4166 pairs of incorrect/correct programs show that our approach correctly maps 83% of the evaluation dataset. Moreover, our experiments show that the current state-of-the-art on program repair, greatly dependent on the programs' structure, can only repair about 72% of the incorrect programs. In contrast, our approach, which is solely based on variable mappings, can repair around 88.5%.

## 1 Introduction

The problem of program equivalence, i.e., deciding if two programs are equivalent, is undecidable [33, 6]. On that account, the problem of repairing an incorrect program based on a correct implementation is very challenging. In order to compare both programs, i.e., the correct and the faulty implementation, program repair tools first need to find a relation between both programs' sets of variables. Besides *program repair* [1], the task of mapping variables between programs is also important for *program analysis* [40], *program equivalence* [8], *program clustering* [27, 39], and *clone detection* [15].

Due to a large number of student enrollments every year in programming courses, providing feedback to novice students in *introductory programming assignments* (IPAs) requires substantial time and effort by the faculty [41]. Hence, there is an increasing need for systems capable of providing automated, comprehensive, and personalized feedback to students in programming assignments [12, 10, 11, 1]. *Semantic program repair* has become crucial to provide feedback to each novice programmer by checking their IPAs submissions using a pre-defined test suite. Semantic program repair frameworks use a correct implementation, provided by the lecturer or submitted by a previously enrolled student, to repair a new incorrect student's submission. However, the current state-of-the-art tools on semantic program repair [10, 1] for IPAs have two main drawbacks: (1) require a perfect match between the control flow graphs (loops, functions) of both programs, the correct and the incorrect one; and (2) require a bijective relation between both programs' sets of variables. Hence, if one of these requirements is not satisfied, then, these tools cannot fix the incorrect program with the correct one.

For example, consider the two programs presented in Figure 1. These programs are students' submissions for the IPA of printing all the natural numbers from 1 to a given number $n$. The program in Listing 1 is a semantically correct implementation that uses a for-loop to iterate all the natural numbers until $n$. The program in Listing 2 uses a while-loop and an auxiliary function. This program is semantically incorrect since the student forgot to initialize the variable $j$, a frequent bug among novice programmers called *missing expression/assignment* [35]. However, in this case, state-of-the-art program repair tools [10, 1] cannot fix the buggy program, since the control flow graphs do not match either due to using different loops (for-loop vs. while-loop) or due to the use of an auxiliary function. Thus, these program repair tools cannot leverage on the correct implementation in Listing 1 to repair the faulty program in Listing 2.

To overcome these limitations, in this paper, we propose a novel graph program representation based on the structural information of the *abstract syntax trees* (ASTs) of imperative programs to learn how to map the set of variables between two programs using *graph neural networks* (GNNs). Additionally, we present use-cases of program repair where these variable mappings can be applied to repair common bugs in incorrect students' programs that previous tools are not always capable of handling. For example, consider again the two programs presented in Figure 1. Note that having a mapping between both programs' variables (e.g. {`n : l; i : j`}) lets us reason about, on the level of expressions, which program fixes one can perform on the faulty program in Listing 2. In this case, when comparing variable `i` with variable `j` one would find the *missing assignment* i.e., `j = 1`.

[*] Corresponding Author: pmorvalho@inesc-id.pt. This work was done while this author was visiting CIIRC, CTU in Prague.

**Listing 1:** A semantically correct student's implementation.

```
int main(){
    int n, i;
    scanf("%d", &n);
    for(i = 1; i <= n; i++){
        printf("%d\n", i);
    }
    return 0;
}
```

**Listing 2:** A semantically incorrect student's implementation since the variable `j` in the `main` function is not initialized.

```
void loop(int j, int l){
  while (l >= j){
    printf("%d\n", j);
    ++j;
  }
}
int main(){
  int j, l;
  scanf("%d", &l);
  loop(j, l);
  return 0;
}
```

**Figure 1**: Two implementations for the IPA of printing all the natural numbers from 1 to a given number $n$. The program in Listing 2 is semantically incorrect since the variable `j`, which is the variable being used to iterate over all the natural numbers until the number `l`, is not being initialized, i.e., the program has a bug of *missing expression*. The mapping between these programs' sets of variables is {`n` : `l`; `i` : `j`}.

Another useful application for mapping variables between different programs is fault localization. There is a body of research on fault localization [16, 21, 22, 23], that requires the usage of assertions in order to verify programs. Variable mappings can be helpful in sharing these assertions among different programs. Additionally, several program repair techniques (e.g., SEARCHREPAIR [18], CLARA [10]) enumerate all possible mappings between two programs' variables during the search for possible fixes, using a correct program [10] or code snippets from a database [18]. Thus, variable mappings can drastically reduce the search space, by pruning all the other solutions that use a different mapping.

In programming courses, unlike in production code, typically, there is a reference implementation for each programming exercise. This comes with the challenge of comparing different names and structures between the reference implementation and a student's program. To deal with this challenging task, we propose to map variables between programs using GNNS. Therefore, we explore three tasks to illustrate the advantages of using variable mappings to repair some frequent bugs without considering the incorrect/correct programs' control flow graphs. Hence, we propose to use our variable mappings to fix bugs of: *wrong comparison operator*, *variable misuse*, and *missing expression*. These bugs are recurrent among novice programmers [35] and have been studied by prior work in the field of automated program repair [3, 31, 37, 4].

Experiments on 4166 pairs of incorrect/correct programs show that our GNN model correctly maps 83% of the evaluation dataset. Furthermore, we also show that previous approaches can only repair about 72% of the dataset, mainly due to control flow mismatches. On the other hand, our approach, solely based on variable mappings, can fix 88.5%.

The main contributions of this work are:

- A novel graph program representation that is agnostic to the names of the variables and for each variable in the program contains a representative variable node that is connected to all the variable's occurrences;
- We propose to use GNNS for mapping variables between programs based on our program representation, ignoring the variables' identifiers;
- Our GNN model and the dataset used for this work's training and evaluation, will be made open-source and publicly available on GitHub: https://github.com/pmorvalho/ecai23-GNNs-for-mapping-variables-between-programs.

The structure of the remainder of this paper is as follows. First, Section 2 presents our graph program representations. Next, Section 3 describes the GNNS used in this work. Section 4 introduces typical program repair tasks, as well as our program repair approach using variable mappings. Section 5 presents the experimental evaluation where we show the effectiveness of using GNNS to produce correct variable mappings between programs. Additionally, we compare our program repair approach based on the variable mappings generated by the GNN with state-of-the-art program repair tools. Finally, Section 6 describes related work, and the paper concludes in Section 7.

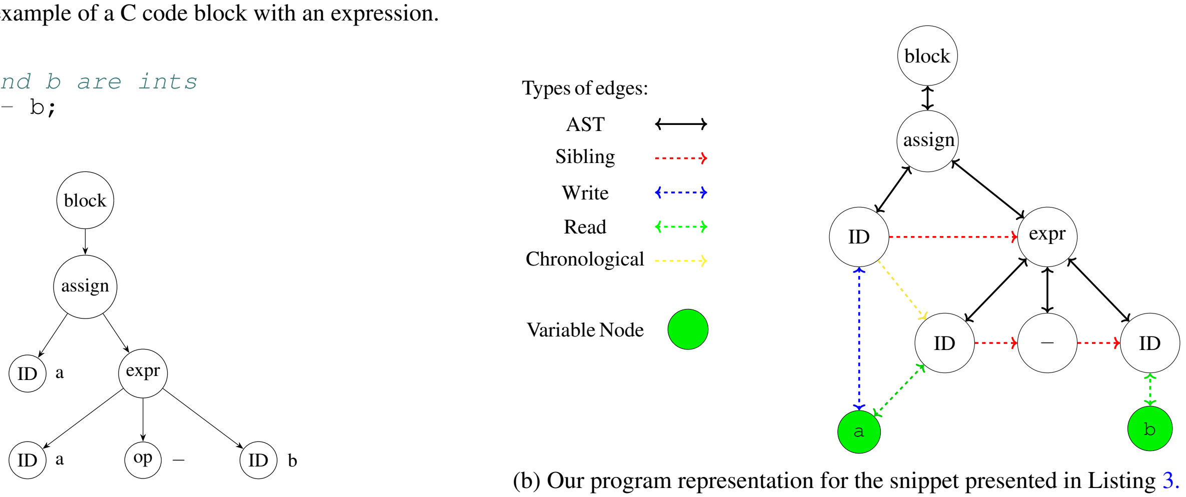
## 2 Program Representations

We represent programs as directed graphs so the information can propagate in both directions in the GNN. These graphs are based on the programs' *abstract syntax trees* (ASTS). An AST is described by a set of nodes that correspond to non-terminal symbols in the programming language's grammar and a set of tokens that correspond to terminal symbols [14]. An AST depicts a program's grammatical structure [2]. Figure 2a shows the AST for the small code snippet presented in Listing 3.

Regarding our graph program representation, firstly, we create a unique node in the AST for each distinct variable in the program and connect all the variable occurrences in the program to the same unique node. Figure 2b shows our graph representation for the small code snippet presented in Listing 3. Observe that our representation uses a single node for each variable in the program, the green nodes `a` and `b`. Moreover, we consider five types of edges in our representation: child, sibling, read, write, and chronological edges. *Child edges* correspond to the typical edges in the AST representation that connect each parent node to its children. Child edges are bidirectional in our representation. In Figure 2b, the black edges correspond to child edges. *Sibling edges* connect each child to its sibling successor. These edges denote the order of the arguments for a given node and have been used in other program representations [3]. Sibling edges allow the program representation to differentiate between different arguments when the order of the arguments is important (e.g. binary operation such as $\leq$). For example, consider the node that corresponds to the operation $\sigma(A_1, A_2, \dots, A_m)$. The parent node $\sigma$ is connected to each one of its children by a child edge e.g. $\sigma \leftrightarrow A_1, \sigma \leftrightarrow A_2, \dots, \sigma \leftrightarrow A_m$. Additionally, each child its connected to its successor by a sibling edge e.g.

**Listing 3:** Small example of a C code block with an expression.

```
{ // a and b are ints
  a = a - b;
}
```



(a) Part of the AST representation of Listing 3.

(b) Our program representation for the snippet presented in Listing 3.

**Figure 2**: AST and our graph representation for the small code snippet presented in Listing 3.

$A_1 \rightarrow A_2, A_2 \rightarrow A_3, \ldots, A_{m-1} \rightarrow A_m$. In Figure 2b, the red dashed edges correspond to sibling edges.

Regarding the *write and read edges*, these edges connect the `ID` nodes with the unique nodes corresponding to some variable. Write edges are connections between an `ID` node and its variable node. This edge indicates that the variable is being written. Read edges are also connections between an `ID` node and its variable node, although these edges indicate that the variable is being read. In Figure 2b, the blue dashed edge corresponds to a write edge while the green dashed edges correspond to read edges. Lastly, *chronological edges* establish an order between all the `ID` nodes connected to some variable. These edges denote the order of the `ID` nodes for a given variable node. For example, in Figure 2b, the yellow dashed edge corresponds to a chronological edge between the `ID` nodes of the variable `a`. Besides the siblings and the chronological edges, all the other edges are bidirectional in our representation.

*The novelty of our graph representation* is that we create a unique variable node for each variable in the program and connect each variable's occurrence to its unique node. This lets us map two variables in two programs, even if their number of occurrences is different in each program. Furthermore, the variable's identifier is suppressed after we connect all the variable's occurrences to its unique node. This way, all the variables' identifiers are anonymized. Prior work on representing programs as graphs [3, 37, 4] use different nodes for each variable occurrence and take into consideration the variable identifier in the program representation. Furthermore, to the best of our knowledge, combining all five types of edges (sibling, write, read, chronological, and AST) is also novel. Section 5.3 presents an ablation study on the set of edges to analyze the impact of each type of edge.

## 3 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a subclass of neural networks designed to operate on graph-structured data [20], which may be citation networks [7], mathematical logic [9] or representations of computer code [3]. Here, we use graph representations of a pair of ASTs, representing two programs for which we want to match variables, as the input. The main operative mechanism is to perform *message passing* between the nodes, so that information about the global problem can be passed between the local constituents. The content of these messages and the final representation of the nodes is parameterized by neural network operations (matrix multiplications composed with a non-linear function). For the variable matching task, we do the following to train the parameters of the network. After several message passing rounds through the edges defined by the program representations above, we obtain numerical vectors corresponding to each variable node in the two programs. We compute scalar products between each possible combination of variable nodes in the two programs, followed by a softmax function. Since the program samples are obtained by program mutation, the correct mapping of variables is known. Hence, we can compute a cross-entropy loss and minimize it so that the network output corresponds to the labeled variable matching. Note that the network has no information on the name of any object, which means that the task must be solved purely based on the structure of the graph representation. Therefore, our method is invariant to the consistent renaming of variables.

**Architecture Details.** The specific GNN architecture used in this work is the relational graph convolutional neural network (RGCN), which can handle multiple edges or relation types within one graph [34]. The numerical representation of nodes in the graph is updated in the message passing step according to the following equation:

$$\mathbf{x}'_i = \mathbf{\Theta}_{\text{root}} \cdot \mathbf{x}_i + \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_r(i)} \frac{1}{|\mathcal{N}_r(i)|} \mathbf{\Theta}_r \cdot \mathbf{x}_j,$$

where $\mathbf{\Theta}$ are the trainable parameters, $\mathcal{R}$ stands for the different edge types that occur in the graph, and $\mathcal{N}_r$ the neighbouring nodes of the current node $i$ that are connected with the edge type $r$ [32]. After each step, we apply Layer Normalization [5] followed by a Rectified Linear Unit (ReLU) non-linear function.

We use two separate sets of parameters for the message passing phase for the program with the bug and the correct program. Five message passing steps are used in this work. After the message passing phase, we obtain numerical vectors representing every node in both graphs. We then calculate dot products $\vec{a} \cdot \vec{b}$ between the vectors representing variable nodes in the buggy program graph $a \in A$ and the variable nodes from the correct graph $b \in B$, where $A$ and $B$

are the sets of variable node vectors. A score matrix $\mathcal{S}$ with dimensions $|A| \times |B|$ is obtained, to which we apply the softmax function on each row to obtain the matrix $\mathcal{P}$. The values in each row of $\mathcal{P}$ can now be interpreted as representing the probability that variable $a_i$ maps to each of the variables $b_i$.

## 4 Use-Cases: Program Repair

In this section, we propose a few use-cases on how to use variable mappings for program repair. More specifically, to repair bugs of: *wrong comparison operator*, *variable misuse*, and *missing expression*. These bugs are common among novice programmers [35] and have been studied by prior work in the field of automated program repair [3, 31, 37, 4]. The current state-of-the-art on semantic program repair tools focused on repairing IPAS, such as CLARA [10] and VERIFIX [1], are only able to fix these bugs if the correct expression in the correct program is located in a similar program structure as the incorrect expression in the incorrect implementation. For example, consider again the two programs presented in Figure 1. If the loop condition was incorrect in the faulty program, CLARA and VERIFIX could not fix it, since the control flow graphs do not match. Thus, these tools would fail due to *structural mismatch*.

The following sections present three program repair tasks that take advantage of variable mappings to repair an incorrect program using a correct implementation for the same IPA without considering the programs' structures. Our main goal is to show the usefulness of variable mappings. We claim that variable mappings are informative enough to repair these three realistic types of bugs. Given a buggy program, we search for and try to repair all three types of bugs. Whenever we find a possible fix, we check if the program is correct using the IPA's test suite.

**Bug #1: Wrong Comparison Operator (WCO).** Our first use-case are faulty programs with the bug of wrong comparison operator (WCO). This is a recurrent bug in students' submissions to IPAS since novice programmers frequently use the wrong operator, e.g., `i <= n` instead of `i < n`.

We propose tackling this problem solely based on the variable mapping between the faulty and correct programs, ignoring the programs' structure. First, we rename all the variables in the incorrect program based on the variable mapping by changing all the variables' identifiers in the incorrect program with the corresponding variables' identifiers in the correct implementation. Second, we count the number of times each comparison operation appears with a specific pair of variables/expressions in each program. Then, for each comparison operation in the correct program, we compute the mirrored expression, i.e., swapping the operator by its mirrored operator, and swapping the left-side and right-side of the operation. This way, if the incorrect program has the same correct mirrored expression, we can match it with an expression in the correct program. For example, in the programs shown in Figure 1, both loop conditions would match even if they are mirrored expressions, i.e., `i <= n` and `n >= i`.

Afterwards, we iterate over all the pairs of variables/expressions that appear in comparison operations of the correct program (plus the mirrored expressions) and compare if the same pair of variables/expressions appear the same number of times in the incorrect program, using the same comparison operator. If this is not the case, we try to fix the program using the correct implementation's operator in each operation of the incorrect program with the same pair of variables/-expressions. Once the program is fixed, we rename all the variables based on the reverse variable mapping.

**Bug #2: Variable Misuse (VM).** Our second program repair task are buggy programs with variables being misused, i.e., the student uses the wrong variable in some program location. The wrong variable is of the same type as the correct variable that should be used. Hence, this bug does not produce any compilation errors. This type of bug is common among students and experienced programmers [17, 36]. The task of detecting this specific bug has received much attention from the Machine Learning (ML) research community [3, 37, 42].

Once again, we propose to tackle this problem based on the variable mapping between the faulty program and the correct one, ignoring the programs' structure. We start by renaming all the variables in the incorrect program based on the variable mapping. Then we count the number of times each variable appears in both programs. If a variable, `x`, appears more times in the incorrect program than in the correct implementation, and if another variable `y` appears more times in the correct program, then we try to replace each occurrence of `x` in the incorrect program with `y`. Once the program is fixed, we rename all the variables based on the reverse variable mapping.

**Bug #3: Missing Expression (ME).** The last use-case we will focus on is to repair the bug of *missing expressions/assignments*. This bug is also recurrent in students' implementations of IPAS [35]. Frequently, students forget to initialize some variable or to increment a variable of some loop, resulting in a bug of missing expression. However, unlike the previously mentioned bugs, this one has not received much attention from the ML community since it is more complex to repair this program fault. To search for a possible fix, we start by renaming all the variables in the incorrect program based on the variable mapping. Next, we count the number of times each expression appears in both programs. Expressions that appear more frequently in the correct implementation are considered possible repairs. Then, we try to inject these expressions, one at a time, into the incorrect implementation's code blocks and check the program's correctness. Once the program is fixed, we rename all the variables based on the reverse variable mapping. This task is solely based on the variable mapping between the faulty and the correct programs.

## 5 Experiments

**Experimental Setup.** We trained the Graph Neural Networks on an Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz server with 72 CPUs and 692GB RAM. Networks were trained using NVIDIA GEFORCE GTX 1080 graphics cards with 12GB of memory. All the experiments related to our program repair tasks were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 64GB and a timeout of 60 seconds.

### *5.1* IPAS *Dataset*

To evaluate our work, we used C-PACK-IPAS [26], a benchmark of student programs developed during an introductory programming course in the C programming language for ten different IPAS, over two distinct academic years, at Instituto Superior Técnico. These IPAS are small imperative programs that deal with integers and input-output operations [30]. First, we selected a set of correct submissions, i.e., programs that compiled without any error and satisfied a set of input-output test cases for each IPA. We gathered 238 correct students' submissions from the first year and 78 submissions from the second year. We used the students' submissions from the first

**Table 1**: Validation mappings fully correct after 20 training epochs.

| | **Buggy Programs** | | | |
|---|---|---|---|---|
| | WCO Bug | VM Bug | ME Bug | All Bugs |
| Accuracy | 93.7% | 95.8% | 93.4% | 96.49% |

**Table 2**: The number of correct variable mappings generated by our GNN on the evaluation dataset and the average overlap coefficients between the real mappings and our GNN's variable mappings.

| | **Buggy Programs** | | | |
|---|---|---|---|---|
| **Evaluation Metric** | WCO Bug | VM Bug | ME Bug | All Bugs |
| # Correct Mappings | 87.38% | 81.87% | 79.95% | 82.77% |
| Avg Overlap Coefficient | 96.99% | 94.28% | 94.51% | 95.05% |
| # Programs | 1078 | 1936 | 1152 | 4166 |

year for training and for validating our GNN and the submissions from the second year for evaluating our work.

Since we need to know the real variable mappings between programs (ground truth) to evaluate our representation, we generated a dataset of pairs of correct/incorrect programs to train and evaluate our work with specific bugs. This is a common procedure to evaluate machine learning models in the field of program repair [3, 37, 4, 42, 29]. To generate this dataset, we used MULTIPAS [28], a program modifier capable of mutating C programs syntactically, generating semantically equivalent programs, i.e., changing the program's structure but keeping its semantics. There are several program mutations available in MULTIPAS: mirroring comparison expressions, swapping the if's then-block with the else-block and negating the test condition, increment/decrement operators mirroring, variable declarations reordering, translating for-loops into equivalent while-loops, and all possible combinations of these program mutations. Hence, MULTIPAS has thirty-one different configurations for mutating a program. All these program mutations generate semantically equivalent programs. Afterwards, we also used MULTIPAS, to introduce bugs into the programs, such as *wrong comparison operator* (WCO), *variable misuse* (VM), *missing expression* (ME). Hence, we gathered a dataset of pairs of programs and the mappings between their sets of variables [30]. Each pair corresponds to a real correct student's implementation, and the second program is the student's program after being mutated and with some bug introduced. Thus, this IPA dataset is generated, although based on real programs. The dataset is divided into three different sets: training set, validation set, and evaluation set. The programs generated from *first year* submissions are divided into a training and validation set based on which students' submissions they derive from. 80% of the students supply the training data, while 20% supply validation data. The evaluation set, which is not used during the machine learning optimization, is chronologically separate: it consists only of *second year* submissions, to simulate the real-world scenario of new, incoming students. The training set is composed of 3372, 5170, and 2908 pairs of programs from the first academic year for the WCO, VM, and ME bugs, respectively. The validation set, which was used during development to check the generalization of the prediction to unseen data, comprises 1457, 1457, and 1023 pairs of programs from the first year. Note that we subsample from the full spectrum of possible mutations, to keep the training data size small enough to train the network with reasonable time constraints. From each of the 31 combinations of mutations, we use one randomly created sample for each student per exercise. We found that this already introduced enough variation in the training dataset to generalize to unseen data. Finally, the evaluation set is composed of 4166 pairs of programs from the second year (see $3^{rd}$ row, Table 2). This dataset will be publicly available for reproducibility reasons.

## 5.2 Training

At training time, since the incorrect program is generated, the mapping between the variables of both programs is known. The network is trained by minimizing the cross entropy loss between the labels (which are categorical integer values indicating the correct mapping) and the values in each corresponding row of the matrix $\mathcal{P}$. As an optimizer, we used the Adam algorithm with its default settings in PyTorch [19]. The batch size was 1. As there are many different programs generated by the mutation procedures, we took one sample from each mutation for each student. Each network was trained for 20 full passes (epochs) over this dataset while shuffling the order of the training data before each pass. For validation purposes, data corresponding to 20% of the students from the first year of the dataset was kept separate and not trained on.

Table 1 shows the percentage of validation data mappings that were exactly correct (accuracy) after 20 epochs of training, using four different GNN models. Each GNN model was trained on programs with the bugs of wrong comparison operator (WCO), variable misuse (VM), missing expression (ME) or all of them (All). Furthermore, each GNN model has its own validation set with programs with a specific type of bug. The GNN model trained on All Bugs was validated using a mix of problems from each bug type. In the following sections, we focus only on this last GNN model (All Bugs).

## 5.3 Evaluation

Our GNN model was trained on programs with bugs of wrong comparison operator (WCO), variable misuse (VM), and missing expression (ME). We used two evaluation metrics to evaluate the variable mappings produced by the GNN. First, we counted the number of totally correct mappings our GNN was able to generate. We consider a variable mapping totally correct if it correctly maps all the variables between two programs. Secondly, we computed the overlap coefficient between the original variable mappings and the variable mappings generated by our GNN. The overlap coefficient is a similarity metric given by the intersection between the two mappings divided by the length of the variable mapping.

The first row in Table 2 shows the number of totally correct variable mappings computed by our GNN model. One can see that the GNN maps correctly around 83% of the evaluation dataset. We have

**Table 3**: Percentage of variable mappings fully correct on the validation set for different sets of edges used. Each type of edge is represented by an index using the mapping: {0: AST; 1: sibling; 2: write; 3: read; 4: chronological}.

| **Edges Used** | All | (1,2,3,4) | (0,2,3,4) | (0,1,3,4) | (0,1,2,4) | (0,1,2,3) | (0,1) |
|---|---|---|---|---|---|---|---|
| **Accuracy** | **96.49%** | 52.53% | 73.76% | 95.45% | 94.87% | 96.06% | 94.74% |

also looked into the number of variables in the mappings we were not getting entirely correct. The results showed that programs with more variables (e.g., six or seven variables) are the most difficult for our GNN to map their variables correctly (see [30]). For this reason, we have also computed the overlap coefficient between the GNN's variables mappings and the original mappings (ground truth). The second row in Table 2 shows the average of the overlap coefficients between the original variable mappings and the mappings generated by our GNN model. The overlap coefficient [38] measures the intersection (overlap) between two mappings. If the coefficient is 100%, both sets are equal. One set cannot be a subset of the other since both sets have the same number of variables in our case. The opposite is 0% overlap, meaning there is no intersection between the two mappings. The GNN achieved at least 94% of overlap coefficients, i.e., even if the mappings are not always fully correct, almost 94% of the variables are correctly mapped by the GNN.

**Ablation Study.** To study the effect of each type of edge in our program representation, we have performed an ablation study on the set of edges. Prior works have done similar ablation studies [3]. Table 3 presents the accuracy of our GNN (i.e., number of correct mappings) on the evaluation dataset after 20 epochs. We can see that the accuracy of our GNN drops from 96% to 53% if we remove the AST edges (index 0), which was expected since these edges provide syntactic information about the program. Removing the sibling edges (index 1) also causes a great impact on the GNN's performance, dropping to 74%. The other edges are also important, and if we remove them, there is a negative impact on the GNN's performance. Lastly, since the AST and sibling edges caused the greatest impact, we evaluated using only these edges on our GNN and got an accuracy of 94.7%. However, the model using all the proposed edges has the highest accuracy of 96.49%.

### 5.4 *Program Repair*

This section presents the results of using variable mappings on the three use-cases described in Section 4, i.e., the tasks of repairing bugs of: *wrong comparison operator* (WCO), *variable misuse* (VM) and *missing expression* (ME). For this evaluation, we have also used the two current publicly available program repair tools for fixing introductory programming assignments (IPAs): CLARA [10] and VERIFIX [1]. Furthermore, we have tried to fix each pair of incorrect/correct programs in the evaluation dataset by passing each one of these pairs of programs to every repair method: VERIFIX, CLARA, and our repair approach based on the GNN's variable mappings.

If our repair procedure cannot fix the incorrect program using the most likely variable mapping according to the GNN model, then it generates the next most likely mapping based on the variables' distributions computed by the GNN. Therefore, the repair method iterates over all variable mappings based on the GNN's predictions. Lastly, we have also run the repair approach using as baseline variable mappings generated based on uniform distributions. This case simulates most repair techniques that compute all possible mappings between both programs' variables (e.g., SEARCHREPAIR [18]).

Table 4 presents the number of programs repaired by each different repair method. The first row presents the results for the baseline, which was only able to fix around 50% of the evaluation dataset. In the second row, the interested reader can see that VERIFIX can only repair about 62% of all programs. CLARA, presented in the third row, outperforms VERIFIX, being able to repair around 72% of the whole dataset. The last row presents the GNN model. This model is the best one repairing 88.5% of the dataset.

The number of executions that resulted in a timeout (60 seconds) is relatively small for VERIFIX and CLARA. Regarding our repair procedure, it either fixes the incorrect program or iterates over all variable mappings until it finds one that fixes the program. Thus, the baseline and the GNN present no failed executions and considerably high rates of executions that end up in timeouts, almost 50% for the baseline and 11.5% in the case of the GNN model. Additionally, Table 4 also presents the failure rate of each technique, i.e., all the computations that ended within 60 seconds and did not succeed in fixing the given incorrect program. VERIFIX has the highest failure rate, around 35% of the entire evaluation set. CLARA also presents a significant failure rate, about 28%. As explained previously, this is the main drawback of these tools. Hence, these results support our claim that it is possible to repair these three realistic bugs solely based on the variable mappings' information without matching the structure of the incorrect/correct programs.

Furthermore, considering all executions, the average number of variable mappings used within 60 seconds is 1.24 variable mappings for the GNN model and 5.6 variable mappings when considering the baseline. The minimum number of mappings generated by both approaches is 1, i.e., both techniques were able to fix at least one incorrect program using the first generated variable mapping. The maximum number of variable mappings generated was 32 (resp. 48) for the GNN (resp. baseline). The maximum number of variable mappings used is high because the repair procedure iterates over all the variable mappings until the program is fixed or the time runs out. Moreover, even if we would only consider using the first variable mapping generated by the GNN model to repair the incorrect programs, we would be able to fix 3377 programs in 60 seconds, corresponding to 81% of the evaluation dataset.

Regarding the time performance of each technique, Figure 3 shows a cactus plot that presents the CPU time spent, in seconds, on repairing each program ($y$-axis) against the number of repaired programs ($x$-axis) using different repairing techniques. One can clearly see a gap between the different repair methods' time performances. For example, in 10 seconds, the baseline can only repair around 1150 programs, VERIFIX repairs around 2300, CLARA repairs around 2850 programs while using the GNN's variable mappings, we can repair around 3350 programs, i.e., around 17% more. We are considering the time the GNN takes to generate the variable mappings and the time spent on the repair procedure. However, the time spent by the GNN to generate one variable mapping is almost insignificant. The average time the GNN takes to produce a variable mapping is 0.025 seconds. The minimum (resp. maximum) time spent by the GNN, considering all the executions is 0.015s (resp. 0.183s).

**Table 4**: The number of programs repaired by each different repair technique: VERIFIX, CLARA, and our repair approach based on our GNN's variable mappings. The first row shows the results of repairing the programs using variable mappings generated based on uniform distributions (baseline).

| | Buggy Programs | | | | Not Succeeded | |
|---|---|---|---|---|---|---|
| **Repair Method** | WCO Bug | VM Bug | ME Bug | All Bugs | **% Failed** | **% Timeouts (60s)** |
| **Baseline** | 618 (57.33%) | 1187 (61.31%) | 287 (24.91%) | 2092 (50.22%) | 0 (0.0%) | **2074 (49.78%)** |
| **VERIFIX** | 555 (51.48%) | 1292 (66.74%) | 741 (64.32%) | 2588 (62.12%) | **1471 (35.31%)** | 107 (2.57%) |
| **CLARA** | 722 (66.98%) | 1517 (78.36%) | 764 (66.32%) | 3003 (72.08%) | 1153 (27.68%) | 10 (0.24%) |
| **GNN** | **992 (92.02%)** | **1714 (88.53%)** | **981 (85.16%)** | **3687 (88.5%)** | 0 (0.0%) | 479 (11.5%) |

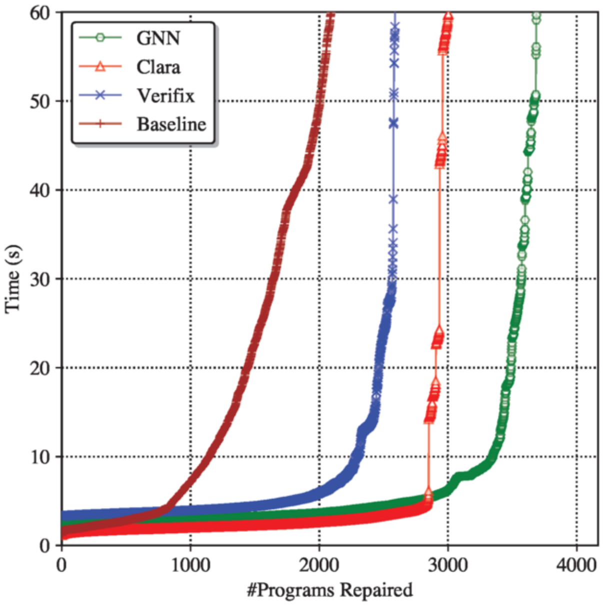


**Figure 3**: Cactus plot - The time spent by each method repairing each program of the evaluation dataset, using a timeout of 60 seconds.

## 6 Related Work

*Automated program repair* [1, 24, 10, 12, 41] has become crucial to provide feedback to novice programmers by checking their introductory programming assignments (IPAS) submissions using a test suite. In order to repair an incorrect program with a correct reference implementation, CLARA [10] requires a perfect match between both programs' control flow graphs and a bijective relation between both programs' variables. Otherwise, CLARA returns a structural mismatch error. VERIFIX [1] aligns the control flow graph (CFG) of an incorrect program with the reference solution's CFG. Then, using that alignment relation and MAXSMT solving, VERIFIX proposes fixes to the incorrect program. VERIFIX also requires a compatible control flow graph between the incorrect and the correct program. BUGLAB [4] is a Python program repair tool that learns how to detect and fix minor semantic bugs. To train BUGLAB, [4] applied four program mutations and introduced four different bugs to augment their benchmark of Python programs. DEEPBUGS [31] uses rule-based mutations to build a dataset of programs from scratch to train its ML-based program repair tool. Given a program, this tool classifies if the program is buggy or not.

*Mapping variables* can also be helpful for the task of *code adaption*, where the repair framework tries to adapt all the variable names in a pasted snippet of code, copied from another program or a Stack Overflow post to the surrounding preexisting code [25]. ADAPTIVEPASTE [25] focused on a similar task to *variable misuse* (VM) repair, it uses a sequence-to-sequence with multi-decoder transformer training to learn programming language semantics to adapt variables in the pasted snippet of code. Recently, several systems were proposed to tackle the VM bug with ML models [3, 13, 40]. These tools classify the variable locations as faulty or correct and then replace the faulty ones through an enumerative prediction of each buggy location [3]. However, none of these methods takes program semantics into account, especially the long-range dependencies of variable usages [25].

## 7 Conclusions

This paper tackles the highly challenging problem of mapping variables between programs. We propose the usage of graph neural networks (GNNS) to map the set of variables between two programs using our novel graph representation that is based on both programs' abstract syntax trees. In a dataset of 4166 pairs of incorrect/correct programs, experiments show that our GNN correctly maps 83% of the evaluation dataset. Furthermore, we leverage the variable mappings to perform automatic program repair. While the current state-of-the-art on program repair can only repair about 72% of the evaluation dataset due to structural mismatch errors, our approach, based on variable mappings, is able to fix 88.5%.

In future work, we propose to integrate our variable mappings into other program repair tools to evaluate the impact of using these mappings to repair other types of bugs. Additionally, we will analyze using our mappings to fix an incorrect program using several correct programs.

## Acknowledgements

This work was supported by Portuguese national funds through FCT under projects UIDB/50021/2020, PTDC/CCI-COM/2156/2021, 2022.03537.PTDC and grant SFRH/BD/07724/2020. This work was also supported by European funds through COST Action CA2011; by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466 (JP), Amazon Research Awards (JP), and by the Ministry of Education, Youth, and Sports within the program ERC CZ under the project POSTMAN no. LL1902. This article is part of the RICAIP project that has received funding from the EU's Horizon 2020 research and innovation program under grant agreement No 857306.

## References

[1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury, 'Verifix: Verified repair of programming assignments', *ACM Trans. Softw. Eng. Methodol.*, (jan 2022).

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley series in computer science / World student series edition, Addison-Wesley, 1986.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi, 'Learning to represent programs with graphs', in *6th International Conference on Learning Representations, ICLR 2018*, (2018).

[4] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt, 'Self-supervised bug detection and repair', in *NeurIPS*, (2021).

[5] Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton, 'Layer normalization', *CoRR*, **http://arxiv.org/abs/1607.06450**, (2016).

[6] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alex Aiken, 'Semantic program alignment for equivalence checking', in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pp. 1027–1040. ACM, (2019).

[7] Daniel Cummings and Marcel Nassar, 'Structured citation trend prediction using graph neural networks', in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3897–3901, (2020).

[8] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller, 'Overcode: Visualizing variation in student solutions to programming problems at scale', *ACM Trans. CHI.*, (2015).

[9] Zarathustra Amadeus Goertzel, Jan Jakubuv, Cezary Kaliszyk, Miroslav Olsák, Jelle Piepenbrock, and Josef Urban, 'The isabelle ENIGMA', in *13th International Conference on Interactive Theorem Proving, ITP 2022*, volume 237 of *LIPIcs*, pp. 16:1–16:21, (2022).

[10] Sumit Gulwani, Ivan Radicek, and Florian Zuleger, 'Automated clustering and program repair for introductory programming assignments', in *PLDI 2018*, pp. 465–480. ACM, (2018).

[11] Rahul Gupta, Aditya Kanade, and Shirish K. Shevade, 'Deep reinforcement learning for syntactic error repair in student programs', in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pp. 930–937. AAAI Press, (2019).

[12] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade, 'Deepfix: Fixing common C language errors by deep learning', in *AAAI 2017*, pp. 1345–1351. AAAI Press, (2017).

[13] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber, 'Global relational models of source code', in *8th International Conference on Learning Representations, ICLR*, (2020).

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to automata theory, languages, and computation, 3rd Edition*, Pearson international edition, Addison-Wesley, 2007.

[15] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu, 'DECKARD: scalable and accurate tree-based detection of code clones', in *29th International Conference on Software Engineering (ICSE 2007)*, pp. 96–105. IEEE Computer Society, (2007).

[16] Manu Jose and Rupak Majumdar, 'Cause clue clauses: error localization using maximum satisfiability', in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pp. 437–446. ACM, (2011).

[17] Rafael-Michael Karampatsis and Charles Sutton, 'How often do single-statement bugs occur?: The manysstubs4j dataset', in *MSR 2020*, pp. 573–577. ACM, (2020).

[18] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun, 'Repairing programs with semantic code search (T)', in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pp. 295–306. IEEE Computer Society, (2015).

[19] Diederik P. Kingma and Jimmy Ba, 'Adam: A method for stochastic optimization', in *3rd International Conference on Learning Representations, ICLR 2015*, (2015).

[20] Thomas N. Kipf and Max Welling, 'Semi-supervised classification with graph convolutional networks', in *5th International Conference on Learning Representations, ICLR 2017*, (2017).

[21] Robert Könighofer and Roderick Bloem, 'Automated error localization and correction for imperative programs', in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, eds., Per Bjesse and Anna Slobodová, pp. 91–100. FMCAD Inc., (2011).

[22] Si-Mohamed Lamraoui and Shin Nakajima, 'A formula-based approach for automatic fault localization of imperative programs', in *International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, eds., Stephan Merz and Jun Pang. Springer, (2014).

[23] Si-Mohamed Lamraoui and Shin Nakajima, 'A formula-based approach for automatic fault localization of multi-fault programs', *JIP*, (2016).

[24] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu, 'Automatic grading of programming assignments: an approach based on formal semantics', in *ICSE (SEET) 2019*, pp. 126–137. IEEE / ACM, (2019).

[25] Xiaoyu Liu, Jinu Jang, Neel Sundaresan, Miltiadis Allamanis, and Alexey Svyatkovskiy, 'Adaptivepaste: Code adaptation through learning semantics-aware variable usage representations', *CoRR*, (2022).

[26] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho, 'C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments', *arXiv preprint arXiv:2206.08768*, **https://doi.org/10.48550/arXiv.2206.08768**, (2022).

[27] Pedro Orvalho, Mikolás Janota, and Vasco Manquinho, 'InvAAST-Cluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments', *arXiv preprint arXiv:2206.14175*, **https://doi.org/10.48550/arXiv.2206.14175**, (2022).

[28] Pedro Orvalho, Mikolás Janota, and Vasco Manquinho, 'MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation', in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022*, pp. 1657–1661. ACM, (2022).

[29] Pedro Orvalho, Jelle Piepenbrock, Mikoláš Janota, and Vasco Manquinho, 'Project Proposal: Learning Variable Mappings to Repair Programs', in *7th Conference on Artificial Intelligence and Theorem Proving, AITP*, (2022).

[30] Pedro Orvalho, Jelle Piepenbrock, Mikolás Janota, and Vasco Manquinho, 'Graph Neural Networks For Mapping Variables Between Programs – Extended Version', *arXiv preprint arXiv:2307.13014*, **https://doi.org/10.48550/arXiv.2307.13014**, (2023).

[31] Michael Pradel and Koushik Sen, 'Deepbugs: a learning approach to name-based bug detection', *ACM Program. Lang.*, **2**(OOPSLA), (2018).

[32] PyTorchGeometric. Documentation. https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.RGCNConv, 2022. Accessed 2022-08-12.

[33] Henry Gordon Rice, 'Classes of recursively enumerable sets and their decision problems', *Transactions of the American Mathematical Society*, **74**(2), 358–366, (1953).

[34] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling, 'Modeling relational data with graph convolutional networks', in *The Semantic Web - 15th International Conference, ESWC 2018*, volume 10843 of *Lecture Notes in Computer Science*, pp. 593–607. Springer, (2018).

[35] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury, 'Codeflaws: a programming competition benchmark for evaluating automated program repair tools', in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*, eds., Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, pp. 180–182. IEEE Computer Society, (2017).

[36] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian, 'Learning to fix build errors with graph2diff neural networks', in *ICSE '20: 42nd International Conference on Software Engineering*, pp. 19–20. ACM, (2020).

[37] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh, 'Neural program repair by jointly learning to localize and repair', in *7th International Conference on Learning Representations, ICLR 2019*. OpenReview.net, (2019).

[38] MK Vijaymeena and K Kavitha, 'A survey on similarity measures in text mining', *Machine Learning and Applications: An International Journal*, **3**(2), 19–28, (2016).

[39] Ke Wang, Rishabh Singh, and Zhendong Su, 'Dynamic neural program embeddings for program repair', in *6th International Conference on Learning Representations, ICLR 2018*. OpenReview.net, (2018).

[40] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang, 'Learning semantic program embeddings with graph interval neural network', *Proc. ACM Program. Lang.*, **4**(OOPSLA), 137:1–137:27, (2020).

[41] Michihiro Yasunaga and Percy Liang, 'Graph-based, self-supervised program repair from diagnostic feedback', in *ICML 2020*, volume 119 of *Proceedings of Machine Learning Research*, pp. 10799–10808. PMLR, (2020).

[42] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann, 'Language-agnostic representation learning of source code from structure and context', in *9th International Conference on Learning Representations, ICLR 2021,*, (2021).