Detect Hidden Dependency to Untangle Commits

Mengdan Fan
Key Lab of High Confidence Software Technologies (Peking University)
Beijing, China
fanmengdan@pku.edu.cn

Wei Zhang*
Key Lab of High Confidence Software Technologies (Peking University)
Beijing, China
zhangw.sei@pku.edu.cn

Haiyan Zhao
Key Lab of High Confidence Software Technologies (Peking University)
Beijing, China
zhhy.sei@pku.edu.cn

Guangtai Liang
Huawei Cloud Computing Technology Co., Ltd.
Beijing, China
liangguangtai@huawei.com

Zhi Jin*
Key Lab of High Confidence Software Technologies (Peking University)
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

In collaborative software development, developers generally make code changes and commit the changes to the repositories. Among others, "making small, single-purpose commits" is considered the best practice for making commits, allowing the team to quickly understand the code changes. Rather than following best practices, developers often make *tangled commits*, which wrap code changes that implement different purposes. Such commits make it difficult for other developers to understand the code changes when conducting subsequent development. Early works on untangling code changes rely on human-specified heuristic rules or features, do not consider context, and are labor intensive. Recent works model the local context of code changes as a graph at the statement level, with statements as nodes and code dependencies as edges, and then cluster the changed statements. However, recent works ignore the hidden dependencies in the global context, e.g. a pair of tangled code changes may have no code dependency, and a pair of untangled code changes may have obvious code dependency. To solve this problem, we focus on detecting hidden dependencies among code changes. We model the global context of code changes as graphs at finer-grained, hierarchical levels, i.e., at both entity and statement levels. Then we propose a Heterogeneous Directed Graph Neural Network (*HD-GNN*) to detect hidden dependencies among code changes by aggregating the global context in both connected or disconnected entity-level subgraphs that intersected with the code changes. Evaluation of common C # and Java datasets with 1,612 and 14k *tangled commits* and manually validated datasets (MVD) with 600 commits shows that *HD-GNN* achieves an average enhancement of effectiveness of 25% and 19.2% compared to existing approaches and far superior to existing approaches in MVD, without sacrificing time efficiency.

*The corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3694996

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**.

KEYWORDS

tangled commit, graph neural network, concern

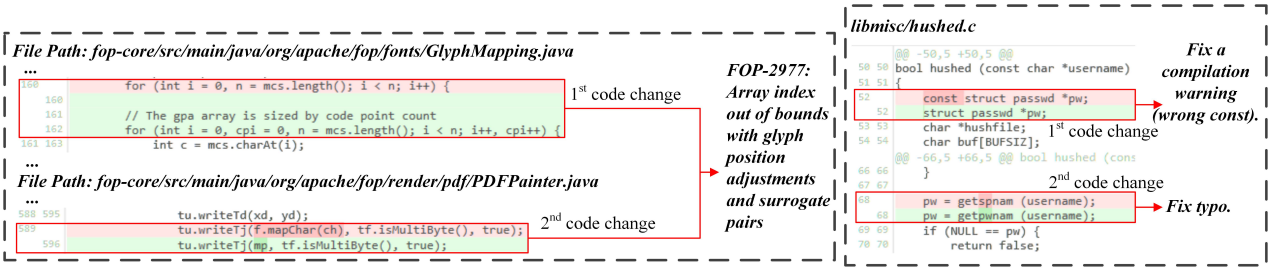
ACM Reference Format:
Mengdan Fan, Wei Zhang, Haiyan Zhao, Guangtai Liang, and Zhi Jin. 2024. Detect Hidden Dependency to Untangle Commits. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3694996

1 INTRODUCTION

In collaborative software development, developers often make code changes for different development concerns and then commit the changes to the repositories. Each concern represents a single purpose for changing the code, such as implementing new features, fixing bugs, refactoring, and so on. *Commit* is the basic unit to organize these code changes, which consists of a set of code changes and a *message*. Each code change consists of one or more consecutive changed statements, and the *message* summarizes the concern of the code changes [23]. 'Making small, single-purpose *commit*' is often considered the best practice to make a *commit* [22, 24, 27]. This practice makes it easier for other developers to understand and review code changes for subsequent development. A single purpose *commit* is called an *atomic commit* [5, 9, 10, 19, 23]. Many communities (e.g., Git official[1], and Angular[2]) and companies (e.g. Google[3]) encourage developers to follow best practice through policies and guidelines.

Unfortunately, instead of following the policies and guidelines of the best practice, developers often make *tangled commits* that wrap code changes that implement different development concerns because of time pressure, and the boundaries between concerns are often unclear[19, 21]. It is difficult for other developers to understand the *tangled commits* when conducting subsequent development [22, 23]. For example, [10] once again show that *tangled*

[1]https://git-scm.com/docs/gitworkflows
[2]https://github.com/angular/angular/blob/master/CONTRIBUTING.md
[3]https://google.github.io/eng-practices/



(a) An *atomic commit* (Commit ID:c2922dc) in repository *fop*.

(b) A *tangled commit* (Commit ID: 2ba18ea) in repository *shadow*.

Figure 1: Two commits in *fop* and *shadow* respectively. 1^{th} and 2^{nd} code changes in (a) belong to a concern but have no code dependency, heuristic rules and features. There are more similar cases described in Section II-B. 1^{st} and 2^{nd} code changes in (b) are tangled but have code dependency, heuristic rules and features.

commits have a strong impact on bug counting models. Many identified buggy files (17.4%) are not real buggy files. In a study of five open source Java repositories, [9] found that between 7% and 20% of all bug fixes consist of multiple *tangled commits*, resulting in an average of 16.6% of all source files incorrectly associated with bug reports. Previous empirical studies [9, 10, 20, 25] have confirmed that *tangled commits* are common in software repositories, i.e., on average 11% ∼ 40% of all commits are tangled.

Researchers have proposed several approaches to automatically untangle *commits*. These approaches have gradually reduced the reliance on human experience (human intuition being involved in deciding on the rules/metrics). Depending on the degree of automation of the approaches, they can be divided into three types: *heuristic rule-based*, *feature-based*, and *Graph clustering-based* approaches. *heuristic rule-based* approaches rely on human experience. These approaches according to human-specified heuristic rules (e.g. # code lines between two code changes; the frequency of two code changes occurring together) to classify the relations (indicates whether the code changes belong to a concern) among code changes [2, 10, 15, 27]. Such approaches treat the clustering problem for code changes as a classification problem of whether each pair of code changes belongs to a concern because the number of concerns is unknown. However, they only focus on the dependencies among code changes, but do not consider the context (unchanged code in the changed files of a *tangled commit*), which makes it difficult to detect some tangles caused by the dependencies among changed and unchanged code, for example, the cases in Fig. 1. The design of heuristic rules is labor intensive because it relies entirely on human experience. And it is labor-intensive to contact the developer who made the *tangled commit* to help understand the commit [23].

Feature-based approaches [5, 30] rely half on the human experience. These approaches define human-specified machine learning (ML) learnable features (e.g. a Boolean value indicating whether a pair of code changes occurs in the same package/class/method; a numeric value indicating the time or # changes between the pair of code changes), and then feed these features into ML algorithms to classify the relations among code changes. Such approaches also treat the clustering problem for code changes as a classification problem for the relations between code changes. These approaches have the same disadvantages as *heuristic rule-based* approaches because they do not consider context (e.g. Fig. 1), and the definition of the features relies on human experience.

Recently, *Graph clustering-based* approaches [4, 19, 21, 23] consider the context of code changes and do not rely on human experience, by modeling each statement as a node, and the code dependency [31] among nodes as edges, to model each tangled commit as a graph first, and then clustering statements. These approaches can solve some cases like Fig. 1(b), but model each code change at a relatively coarse-grained statement level, and focus on detecting obvious code dependencies among code changes (i.e., changed statements) in the local context (i.e., only the statement that have code dependency with code changes), which makes it difficult to detect some hidden dependencies among code changes in the global context (i.e., both of the statements have or have no code dependencies with code changes), e.g. the cases like Fig. 1(a), although two code changes belong to the same concern, there is no obvious dependency between the code changes. An empirical study in section 2 shows the proportion of commits that contain such cases in all *tangled commits* of a repository is about 25% ∼ 37%.

To solve this problem, we focus on capturing the hidden real dependencies among code changes. We first model the each code changes and their global context (i.e., all the code in changed files of a *tangled commit*) as a graph jointly at finer-grained hierarchical levels: The first level is a heterogeneous directed *entity reference graph* that considers entities as nodes and the *syntactic relations* among entities as edges. The second level is a homogeneous undirected *code change graph* that considers code changes as nodes and the hidden dependencies among code changes as edges. These two-level graphs not only capture the code dependencies among entities but also capture the inclusion relationship between a code change and its entities. In contrast to previous approaches, we consider the context that both have or have no code dependency with the code changes. That is, because not all entities have code dependencies with each other, so the *entity reference graph* could consist of multiple disconnected subgraphs, and we consider the hidden dependency between a pair of code changes whether the entity level subgraphs intersected with these two code changes respectively are connected or disconnected. For example, as the motivating cases

shown in Fig. 1(a), we found that although a pair of tangled code changes have no code dependency, i.e., the subgraphs of the *entity reference graph* intersected with these two code changes respectively are disconnected, but the pair of code changes may have hidden dependencies. It may be because there are high similarities in the entity-level subgraphs' structures or some entities separately in the subgraphs reflect high correlation in the context. And as the motivating case shown in Fig. 1(b), although a pair of tangled code changes have some obvious rules/features/code dependencies with each other, the code changes do not belong to a concern. It may be because there are significant differences between the structure of the entity-level subgraphs intersected with these two code changes, or some entities separately in the subgraphs reflect high decorrelation in the context. We second propose a Heterogeneous Directed Graph Neural Network (*HD-GNN*)[4] to untangle the code changes by detecting the hidden dependencies among the code changes. *HD-GNN* first learn the global context by learning the embeddings of nodes and edges on the *entity reference graph*. Then for each pair of code changes, *HD-GNN* aggregate the embeddings of the nodes and edges in both connected and disconnected subgraphs of *entity reference graphs* that intersected with the code changes respectively, and finally classifies the relation between the code changes to determine whether the code changes have hidden dependency/should not be untangled.

We evaluate the effectiveness and efficiency of the proposed approach. Experiments on the common C# and Java datasets with 1,612 and 14k *tangled commits*, and manually validated datasets(MVD) with 600 commits show that *HD-GNN* achieves better untangling results (i.e., an average of 25% and 19% enhancement of effectiveness for C# and Java, compared to existing approaches) and far superior to existing approaches on MVD, without sacrificing time efficiency. Overall, our work makes the following contributions:

- A fine-grained hierarchical graph model of commits. The graph model captures the global context of the code changes by considering the context both have or have no obvious (direct or indirect) code dependency with the code changes.
- A novel *HD-GNN* which can not only learn the embeddings of heterogeneous directed graphs, but also learn hidden dependencies among code changes by aggregating the embeddings of the nodes and edges associated with the code changes in both connected and disconnected entity-level subgraphs. It can be extended to many other graph learning applications.

2 MOTIVATION

2.1 Notation

The notations, meaning, and examples are in Table 1.

2.2 Motivating example

We found that in some complex cases, for a pair of code changes, obvious dependencies such as code dependencies between them can not determine whether they belong to a real concern. We consider this because concern is a vague concept that is difficult to define, and there is no existing work that defines all the categories of concerns. E.g., an *atomic commit* of fop[5] is shown in Fig. 1(a), black dotted box represents a *commit*, solid red boxs represent code changes. It has a independent concern. Although 1^{st} and 2^{nd} code changes belong to the same concern, they are predicted to not belong to a concern according to the heuristic rules or features (e.g. # code lines between two code changes is large). *Graph clustering-based* approaches also make misjudgments because there is no code dependency between the code changes. These approaches are based on the assumption that the existence of a code dependency between a pair of nodes is a prerequisite for the nodes belonging to a concern. While we analyse the concern is abstract and requires a true understanding of the global context to discover that the two changes belong to the same concern. There are more such examples. e.g., refactoring a complex class while optimizing the performance of some methods. Although these two changes focus on the same concern (improve the readability and maintainability of code), they have no code dependencies. Or similar structural optimisations for a number of different functional code blocks, and so on. Similarly, a *tangled commit* of shadow [6] is shown in Fig. 1(b). The concern of the 1^{st} code change is "fix a compilation warning (wrong const)", and the concern of the 2^{nd} code change is "Fix typo". Both 1^{st} and 2^{nd} code changes involve *pw* variable, so these two code changes have *use-use* heuristic rules/features according to feature-based approaches [2, 27]. And there is obvious direct code dependency between these two code changes according to *Graph clustering-based* approaches, so the code changes are clustered/classified into a concern. Thus, it is difficult to untangle commits without understand the global context.

2.3 Empirical study and analysis

How often do the cases in figure 1 occur in real repositories? We invite 20 developers (≥5 years of programming experience) from an international software development company to manually untangle the commits. Code dependency analysis tool is then used to detect code dependencies between each pair of code changes. The result is shown in table 2, each column indicates: name of repository, #commits sampled in all commits (randomly sample from a commit list sorted by time), #tangled commits in the samples, #commits containing the above cases, #code changes in the tangled commits, #cases(code change pairs) similar to Fig. 1(a), and #cases similar to Fig.1(b). The result shows the proportion of the commits containing the above cases in all tangled commits is about 25% ∼ 37%. Nearly 76% of such cases are atomic commits like Fig. 1(a) and 24% are tangled commits like Fig.1(b). So the above cases often appear.

Why obvious dependencies (i.e., heuristic rules/features/code dependencies) used in above three types of existing approaches does not correctly resolve these cases? From a problem perspective, We refer to the previous literature and interviewed 20 developers (randomly invited by email, coming from different departments and involved in development work in different directions, such as front-end development, back-end development, AI model design, theoretical algorithm research, etc.) and observed the cases. About 2-3 developers are recruited for each department.) and observed the cases, summarising the first reason: Concern is a ambiguous concept [30],

[4]The artifacts are available at: https://github.com/fanmengdan/HD-GNN

[5]https://github.com/apache/xmlgraphics-fop

[6]https://github.com/shadow-maint/shadow

Table 1: Notations with meaning and examples

Notations	Meaning	Examples in Fig. 1(a)
p	Commit p consists of a set of code changes and a message. It is one of a set of commits made by developers to modify a software repository. The software repository consists of a set of files and the code changes are scattered in the files.	Fig. 1(a) shows a commit p.
C	The set of code changes $C = \{c_1, c_2 \cdots, c_n\}$ of p.	$C = \{c_1, c_2\}$.
m	Message m of p.	m is "FOP-2977: Array index out of bounds with glyph position adjustments and surrogate pairs"
c_i	i^{th} code change c_i of C, in which c_i consists of a pair of consecutive removed and added statements.	c_1=($c_1^{'}$, $c_1^{''}$), c_2=($c_2^{'}$, $c_2^{''}$).
$c_i^{'}$	Consecutive removed statements $c_i^{'}$.	$c_1^{'}$ and $c_2^{'}$ are red removed statements 160 and 589 in the previous version.
$c_i^{''}$	Consecutive added statements $c_i^{''}$.	$c_1^{''}$ and $c_2^{''}$ are green added statements 160-162 and 596 in current version.
f_i	File f_i that contains code change c_i.	f_1 is *GlyphMapping.java*.

Table 2: Statistics of cases like Fig. 1 in 3 repositories

Repo	#Samp/All	#Tangled	#Contain	#Changes	#Case1	#Case2
fop	500/8443	96	36	10762	65603	11786
platform	200/3609	16	4	854	5476	2125
xerces	500/5527	127	42	5142	12704	4392

and the boundaries between concerns are often unclear[19, 21], thus a limited #obvious dependencies may be difficult to apply to continually increasing ambiguous types of concerns. No existing work defines the types of concerns due to the difficulty of summarizing the ambiguous types of concerns. We track the average #types of concerns for 5 open Java repositories familiar to the developers, in their most frequently modified years. We manually filter concerns based on commit messages and code changes, and consider concerns that have the same functionality but different descriptions to be the same concern to filter the real concerns. Result in Fig. 2 shows #types of concerns are continually increasing. It may be because human consciousness is complex and have inconsistent programming styles, so the granularity of concerns is sometimes fine (e.g. related to a certain entity) and coarse (e.g. abstract concern covering many modifications). So, limited human-defined #obvious dependencies may hard to fit the increasing types of concerns, automating the search for new hidden dependencies is necessary.

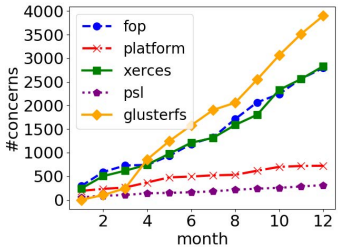


Figure 2: #Types of concerns of 5 repositories in the year they were most frequently modified.

From technical perspective, for manual untangling, there is a significant labor cost to understand code changes written by other people due to the uncertainty of the types of concern and the large size of some commits (e.g., in *drools-wb*[7], 28.68% commits can span dozens of files, thousands of code changes). For automated approaches, it is difficult to model the global context. *heuristic rule-based*, *feature-based* haven't consider the context outside the code changes. Although *Graph clustering-based* considers local context, code changes without code dependencies are difficult to model in the same graph, and only local context has code dependencies with code changes considered. The above reasons may have caused wrong untangled results like Fig.1.

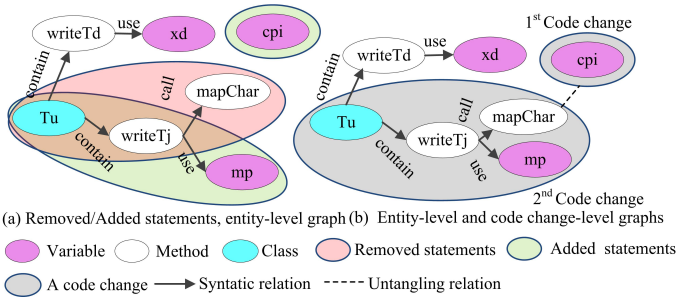


Figure 3: Preliminary local graph of the commit in Fig. 1(a).

2.4 Methodology

Based on the above studies, we focus on modeling and learning the global context to detect hidden dependencies. For each commit, we first model the global context (i.e., both the code that has or has no code dependencies with the code changes in all changed files) as an entity-level graph, where entities are nodes and their *syntactic relations* are edges. The graph is heterogeneous because there is more than one type of node or edge [28], and is incompletely connected because not all entities have *syntactic relations* with each other. As shown in Fig. 3, the entities like the method *writeTj* in both removed and added statements of the 2^{nd} code change, the variable *xd* in the context are nodes. The *syntactic relations* among the above entities are edges, e.g. *writeTj* call *mapChar*. Then the global context of code chnages (i.e., the entity-level graph) is learned by GNN. Related work on learning the graph of the program shows the advantages of *graph neural networks* (*GNN*). For example, *GNN* is used in various areas of software development, such as program understanding [1, 18], vulnerability identification [32], code generation [3], and so on. However, these approaches lack the ability to learn heterogeneous directed graphs. So we propose

[7]https://github.com/kiegroup/drools-wb

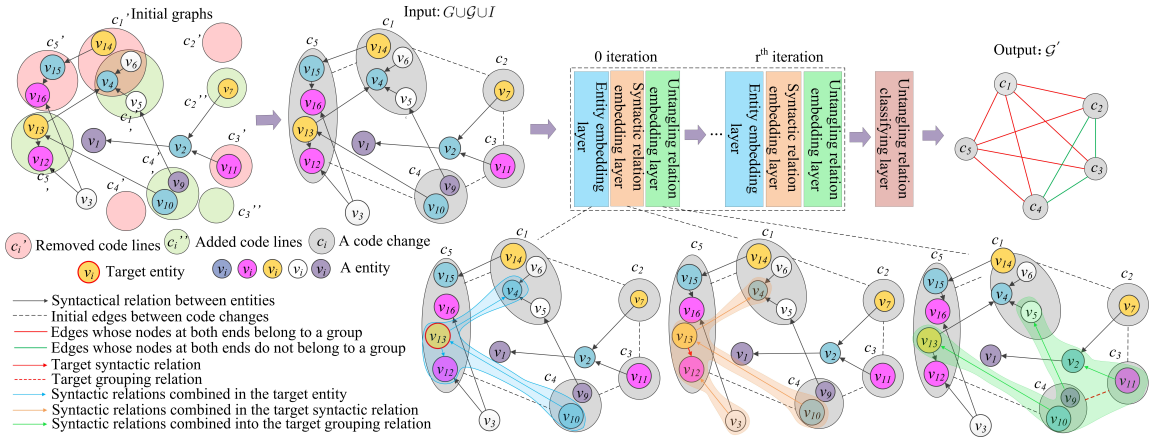


Figure 4: The overview of *HD-GNN*. Each iteration has three embedding layers.

a Heterogeneous Directed Graph Neural Network (HD-GNN) to learn the global context.

Secondly, since the types of concerns is uncertain, we following [2, 5, 10, 15, 27, 30] to transform the clustering problem for code changes into a classification problem for the untangling relations among code changes. We model all code changes with or without code dependencies as nodes, and set *untangling relation* as edges (each of which indicates whether a pair of code changes have a hidden dependency/belong to a concern), as a statement/code change level graph. Then we learn and classify the *untangling relation* between each pair of code changes by combining the global context of the entity-level graph. The graph is a fully-connected homogeneous undirected graph. E.g., as shown in Fig. 3(b), the 1^{st} and the 2^{nd} code changes are taken as nodes, and their *untangling relation* is taken as an edge.

3 HIERARCHICAL MODELING SCHEME AND PROBLEM FORMULATION

3.1 Hierarchical modeling scheme

Each *tangled commit* is initially modeled as graphs on two levels to capture the syntactic structure of *tangled commits*. The graph at the first level is *Entity reference graph*, which captures the global context of the code changes. *Entity reference graph* takes the entity in all changed files of a *tangled commit* such as class, method, and field as nodes, and the *syntactic relations* among entities (e.g. invocation relations between methods and inheritance relations between classes) as directed edges. We implement our proposed approach on Java, but the proposed approach can be extended to other programming languages. The common entities of the Java programming language are summarised according to the book *Core Java Volume I-Fundamentals* [11] and Java tutorials published by vogella company[8]. Then, for each *tangled commit*, the *Entity reference graph* of the previous version and the current version are merged into one new *Entity reference graph*. As shown in the Initial graphs and Input in Fig. 4, the entities v_{14} and v_{15} belong to the removed statements c_1' and c_5' of the previous version respectively; the entity v_6 belongs to both the removed statements c_1' of the previous version and the added statements c_1'' of the current version (i.e., the statement containing v_6 has changed, but v_6 hasn't really been removed); the entity v_3 doesn't belong to any removed or added statements (i.e., the statement containing v_3 hasn't changed); and the entity v_7 belongs to the added statements c_2'' of the current version. Then all the entities v_{14}, v_{15}, v_6, v_3, v_7 and the directed edges among these entities are reserved in the merged graph. The different colors of the entities indicate the different types of entities. In fact, not all entities have edges with other entities, so the *Entity reference graph* may consist of multiple disconnected subgraphs.

The graph on the second level is *code change graph*, which takes code changes as nodes and the *untangling relation* between each pair of code changes as undirected edges, where the edge indicates whether the pair of code changes has hidden dependency/belongs to a concern. For example, as shown in the Input and Output in Fig. 4, c_1 and c_5 are code changes, and the undirected edge between c_1 and c_5 is a *untangling relation* which indicates c_1 and c_5 belong or not belong to a concern. The black dotted line between c_1 and c_5 in the Input of Fig. 4 shows the initial edge that has not been learned by *HD-GNN*. And the solid line between code changes in the Output are the edges learned by *HD-GNN*, where the red/green color means that c_1 and c_5 are predicted to belong/not belong to a concern. *code change graph* is a fully-connected graph.

We use the class *ASTVisitor* of *Java Development Tools (JDT)*[9] to extract the entities of java and then use *JGraphT* [10] to help to construct the input graphs of each *tangled commit*. As a result, the entities and code changes in files of each *tangled commit* are jointly modeled to an input *Entity reference graph* G and input *Code change relation graph* $\mathcal{G}$.

Entity reference graph is a heterogeneous directed graph $G(V, E, X, Y)$, which takes the entities of a *tangled commit* such as class, method, and field as nodes, and the *syntactic relations* among entities (e.g. invocation relations between methods and inheritance relations between classes) as directed edges. V is the set of N entities in the

[8]https://www.vogella.com/tutorials/JavaIntroduction/article.html

[9]https://www.eclipse.org/jdt/

[10]https://jgrapht.org/

previous and the current versions that are involved in a *tangled commit*. $v_i \in V$ $(1 \le i \le N)$ is the i^{th} node of V. E is the set of n edges. Each node has no edge connected to itself. If $v_i \in V$ $(1 \le i \le N)$ and $v_j \in V$ $(1 \le j \le N)$ have a *syntactic relation*, there is an edge $e_{i,j} \in E$ between the nodes v_i and v_j. $X \in \mathbb{R}^{N \times d}$ is the node embedding matrix, $x_i \in \mathbb{R}^{1 \times d}$ $(1 \le i \le N)$ represents the node embedding of entity v_i, and d is the dimension of the node embedding. Entities of different types have different initial node embeddings (i.e., we mark the types of nodes as numbers and then embed the number by one-hot encoding). $Y \in \mathbb{R}^n$ is the edge embedding tensor, where $y_{i,j} \in \mathbb{R}^{1 \times D}$ represents the edge embedding of the edge $e_{i,j}$, and D is the dimension of the edge embedding. n is the # edges. Edges have the same initial embedding.

Code change relation graph takes code changes of a *tangled commit* as nodes and the *untangling relations* (each of which indicates whether the code changes at either end of the relation should be untangled) among the code changes as undirected edges, which is a homogenous undirected graph $\mathcal{G}(C, S, O)$. C is the set of M nodes. $c_i \in C$ $(1 \le i \le M)$ is i^{th} node of C. S is the set of edges. Each pair of nodes has an edge, each node has no edge connected to itself. $s_{i,j} \in S$ is the edge between $c_i \in C$ and $c_j \in C$. $O \in \mathbb{R}^{(M \times M - M) \times L}$ is the edge embedding tensor, $o_{i,j} \in \mathbb{R}^{1 \times L}$ represents the edge embedding of edge $s_{i,j}$. Edges have same initial embedding.

Note that there is no node embedding matrix in *Code change relation graph*. This is because the number of concerns is unknown before a *tangled commit* is untangled, but we do know that there are two types of relation between each pair of code changes: the pair of code changes should be untangled or they should not be untangled. So we follow the previous work [9, 23, 27] to transform the clustering problem for code changes into a classification problem for the relations among code changes and do not need to learn the embeddings of the code change nodes. In addition, if the proposed *GNN*-based model may produce inconsistent results e.g. c_1, c_2, and c_2, c_3 should not be untangled, but c_1, c_3 should be. This means that the model misclassifies the *untangling relations*, so the model needs to be further trained (e.g. training hyperparameters need to be adjusted, training time needs to be increased, and so on).

3.2 Problem formulation

The goal of this paper is to propose a *GNN*-based graph representation learning algorithm to learn the hidden dependencies among the code changes and classify whether each pair of code changes should be untangled or not. That is, this paper expects the following prediction process: The input of the *GNN*-based algorithm is $G \cup \mathcal{G} \cup I$, as shown in the Input in Fig. 4, where I is the index information indicating each entity in which code changes. And the output of the *GNN*-based algorithm is an output *Code change relation graph* $\mathcal{G}^{'}(C, S, O^{'})$, as shown in the Output in Fig.4, where $O^{'} \in \mathbb{R}^{(M \times M - M) \times L}$ is the output edge embedding tensor, $o^{'}_{i,j} \in \mathbb{R}^{1 \times L}$ is the output edge embedding of edge $s_{i,j}$. The edges are of two types, each indicating whether the nodes at either end should be untangled. The graph model of a untangling *tangled commit* is shown as the Initial graphs, Input, and Output in Fig. 4. Initial graphs show the consecutive removed statements and consecutive added statements of each code change, and the *Entity reference graph*. Inputs are the input *Entity reference graph*, *Code change relation graph*, and the index information indicating each entity in which code changes. Output is the *Code change relation graph*.

In the training phase, we minimize the loss between each pair of Output and Target/ground truth. The Target is a target *Code change relation graph* $\mathcal{G}^{''}(C, S, O^{''})$, where $O^{''} \in \mathbb{R}^{(M \times M - M) \times L}$ is the target edge embedding tensor, $o^{''}_{i,j} \in \mathbb{R}^{1 \times L}$ is the edge embedding of edge $s_{i,j}$. The value $o^{''}_{i,j}$ has 2 types, each of which indicates whether the nodes at either end should be untangled. Finally, the problem of untangling *tangled commit* is transformed into learning a function F: $F(G \cup \mathcal{G} \cup I) \rightarrow \mathcal{G}^{'}$ by reducing the *GNN*-based algorithm's loss between $\mathcal{G}^{'}$ and $\mathcal{G}^{''}$.

4 HETEROGENEOUS DIRECTED GRAPH NEURAL NETWORK

In addition to nodes, edges also have structural features, e.g. in a molecular graph, the bond between a pair of atoms may have features describing the type of the bond. Therefore, in previous graph learning tasks, some studies have encoded edge features together with node features by *GNN* [6, 7, 12, 14, 17, 29].

In this section, inspired by the previous work on graph learning mentioned above, we proposed a hierarchical *Heterogeneous Directed Graph Neural Network* (*HD-GNN*). *HD-GNN* could learn the features of nodes and edges in heterogeneous directed *Entity reference graph* and homogenous undirected *Code change relation graph*, and then classify the *untangling relation* between each pair of code changes that indicate whether the pair of code changes has hidden dependency, i.e., whether the code changes belong to a concern.

4.1 Overview

Overview is shown in Fig 4. To learn a function F: $F(G \cup \mathcal{G} \cup I) \rightarrow \mathcal{G}^{'}$, *HD-GNN* iteratively updates the embeddings of entities, and the *syntactic relations* among entities to learn the global context, and then sends the context to the *untangling relations* among code changes, and finally classifies the *untangling relations*. Each iteration has three layers: *Entity embedding layer*, *Syntactic relation embedding layer* and *Untangling relation embedding layer*. And after the last iteration there is a *Untangling relation classifying layer*.

4.2 Entity embedding layer

Entity embedding layer is used to update the embedding of each entity by aggregating the structural information around the entity. Specifically each entity updates its embedding by aggregating the embedding of the *syntactic relations* associated with it and the embedding of its first-order neighbours. *Entity embedding layer* of the l^{th} iteration is shown in Fig. 5. We denote x_i^l as the embedding of v_i at the l^{th} iteration and define $x_i^0 = x_i$ (x_i represents the node embedding of the entity v_i in the input G described in Section 3.1). x_i^l could be define as:

$$x_i^l = Aggregator^e(\{[x_i^{l-1}, y_{i,j}^{l-1}, x_j^{l-1}] | v_j \in \mathcal{N}(v_i)\}) \quad (1)$$

Where $\mathcal{N}(v_i)$ is the set of neighbours of v_i. $[\cdot]$ function concatenates its variables sequentially. $Aggregator^e(\cdot)$ aggregates its

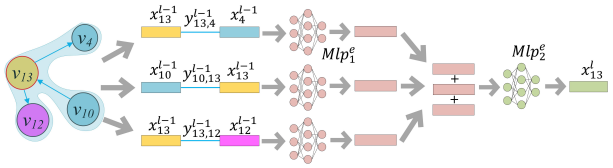


Figure 5: Entity embedding layer of l^{th} iteration

variables (a set of vectors), which is defined as:

$$Aggregator^e(\cdot) = Mlp_2^e(Sum(Mlp_1^e(\cdot))) \tag{2}$$

Where Mlp_1^e and Mlp_2^e are different *Multi-Layer Perceptions* functions. And *Sum* function sums the output (a set of vectors) of Mlp_1^e. For each entity v_i, we do not use the *Combinator* like other GNNs to integrate its embedding of the last iteration [8, 13, 14, 18, 26], because its embedding of the last iteration x_i^{l-1} has been integrated into its embedding of current iteration $x_i^{(l)}$ by $[\cdot]$ function.

4.3 Syntactic relation embedding layer

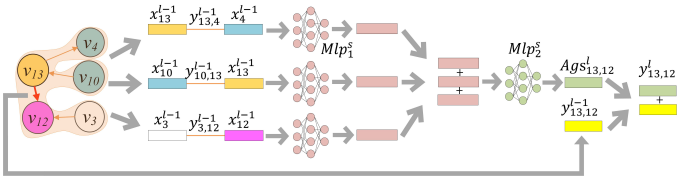


Figure 6: Syntactic relation embedding layer of l^{th} iteration

Syntactic relation embedding layer is used to update the embedding of each *syntactic relation* by aggregating the structural information around *syntactic relation*. In this layer, each *syntactic relation* updates its embedding by aggregating: 1) the embedding of the entities at its two ends, 2) its first-order neighboring *syntactic relations*, and 3) the embedding of the entities connected to the first-order neighbouring *syntactic relations*. Then the result of aggregation is combined with the *syntactic relation* embedding of the last iteration. *Syntactic relation embedding layer* of l^{th} iteration is shown in Fig. 6. We denote $y_{i,j}^l$ as the embedding of $e_{i,j}$ in l^{th} iteration, and define $y_{i,j}^0 = y_{i,j}$ ($y_{i,j}$ represents the embedding of the edge of edge $e_{i,j}$ in the input G, which is described in Section 3.1). The update process of *syntactic relation* embedding of $y_{i,j}$ in l^{th} iteration could be defined as:

$$\begin{aligned} Ags_{i,j}^l = Aggregator^s(\{[x_i^{l-1}, y_{i,z}^{l-1}, x_z^{l-1}], [x_j^{l-1}, y_{j,k}^{l-1}, x_k^{l-1}] \\ |v_z \in \mathcal{N}(v_i), v_k \in \mathcal{N}(v_j)\}) \end{aligned} \tag{3}$$

$$y_{i,j}^l = Combinator^s(y_{i,j}^{l-1}, Ags_{i,j}^l) \tag{4}$$

where $Aggregator^s(\cdot)$ aggregates its variables. $Ags_{i,j}^l$ has the same dimension as $y_{i,j}^{l-1}$. The function $[\cdot]$ concatenates its variables sequentially. For each *syntactic relation* we use *Combinator* as other GNNs to combine the vector $y_{i,j}^{l-1}$ into the vector $y_{i,j}^l$. The $Aggregator^s(\cdot)$ and $Combinator^s(\cdot)$ are defined respectively as:

$$Aggregator^s(\cdot) = Mlp_2^s(Sum(Mlp_1^s(\cdot))) \tag{5}$$

$$Combinator^s(\cdot) = Sum(\cdot) \tag{6}$$

where Mlp_1^s and Mlp_2^s are different *Multi-Layer Perceptions* functions. And the *Sum* function sums its variables.

4.4 Untangling relation embedding layer

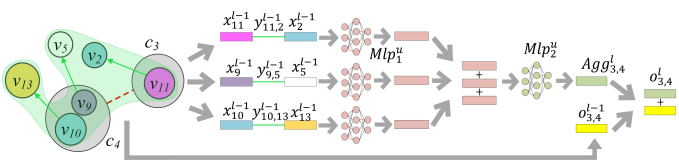


Figure 7: Untangling relation embedding layer of l^{th} iteration

Untangling relation embedding layer is used to update the embedding of each *untangling relation* by aggregating the structural information around *untangling relation*. *Untangling relation embedding layer* of l^{th} iteration is shown in Fig. 7. In this layer each *untangling relation* updates its embedding by aggregating: 1) the entities and the *syntactic relations* in the code change at its two ends, 2) the *syntactic relations* connected with the entities in the code changes at its two ends, and 3) the first-order neighbouring entities of the entities in the code changes at its two ends. And then the result of aggregation are combined with the *untangling relation* embedding of the last iteration. We denote $o_{i,j}^l$ as the embedding of $s_{i,j}$ in l^{th} iteration, and define $o_{i,j}^0 = o_{i,j}$ ($o_{i,j}$ represents the embedding of the edge of edge $s_{i,j}$ in the input $\mathcal{G}(C, S, O)$, which is described in Section 3.1). The process of updating the *untangling relation* embedding of $o_{i,j}$ in iteration l^{th} could be defined as:

$$\begin{aligned} Agg_{i,j}^l = Aggregator^u(\{[x_a^{l-1}, y_{a,z}^{l-1}, x_z^{l-1}], [x_b^{l-1}, y_{b,k}^{l-1}, x_k^{l-1}] \\ |v_a \prec c_i, v_b \prec c_j, v_z \in \mathcal{N}(v_a), v_k \in \mathcal{N}(v_b)\}) \end{aligned} \tag{7}$$

$$o_{i,j}^l = Combinator^u(o_{i,j}^{l-1}, Agg_{i,j}^l) \tag{8}$$

Where $Aggregator^u(\cdot)$ aggregates its variables. $[\cdot]$ concatenates its variables sequentially. The element to $\prec$'s left is inside the element to its right, e.g. the entity v_a is inside the code change c_i. $Agg_{i,j}^l$ has the same dimension as $o_{i,j}^{l-1}$. $\mathcal{N}(v_a)$ and $\mathcal{N}(v_b)$ is the set of neighbors of v_a and v_b. For each *untangling relation*, we use the *Combinator* like other GNNs to combine $o_{i,j}^{l-1}$ into $o_{i,j}^l$. The $Aggregator^u(\cdot)$ and $Combinator^u(\cdot)$ are respectively defined as:

$$Aggregator^u(\cdot) = Mlp_2^u(Sum(Mlp_1^u(\cdot))) \tag{9}$$

$$Combinator^u(\cdot) = Sum(\cdot) \tag{10}$$

where Mlp_1^u and Mlp_2^u functions are a *Multi-Layer Perceptions*. And *Sum* function sums its variables.

In particular, there is no obvious code dependency between c_3 and c_4. Because the subgraph of the *entity reference graph* intersected with c_3 (consists of v_{11}, v_2 and their edges) and the subgraph intersected with c_4 (consists of v_9, v_{10}, v_{13} and their edges) are disconnected. That is, entities respectively in c_3 and c_4 have no connected edges. But *HD-GNN* could also learn the embedding of their *untangling relation* which may indicate c_3 and c_4 belong to the same concern/have hidden dependency (marked as red dotted line). Symmetrically, if a pair of code changes have obvious code dependency, *HD-GNN* could also learn the embedding of their *untangling relation* that may indicate that these two code changes do not belong to the same concern/have no hidden dependency.

4.5 Untangling relation classifying layer

This layer has two neurons. For each *untangling relation*, the neurons calculate two scores for the two possible types: whether the nodes at either end belong to a concern/should be untangled. The score of each neuron is then fed into the softmax function to calculate a probability. The type with the maximum probability is the final type of the *untangling relation*.

5 EXPERIMENTS

5.1 Research Questions

The goal of this paper is to solve cases that were difficult to solve by previous approaches, thus improving the accuracy. The following questions are set (on Inter(R)Core(TM)i7-8700CPU@3.2GHZ and 32GB RAM): **RQ1. How effective is *HD-GNN* compared to SOTA approaches on C# datasets? RQ2. How effective is *HD-GNN* compared to SOTA approaches in Java datasets? RQ3. Whether heterogeneous graph networks will increase time consumption?** Since our approach learns the global context, we are concerned that it will lead to a decrease in efficiency.

5.2 Datasets

Common datasets We first use two datasets that have been widely used in previous work: 1)A C# dataset used in [4, 19, 21], which contains 1,612 tangled commits (each of which has ≤ 3 concerns) across 9 Github projects. Although our approach was initially limited to Java, we modified the tools for constructing graphs based on the C#, as described in the experimental procedure in *Settings for RQ1*. 2) A Java dataset used in [19]. This dataset is collected in the same way as [23]. The Java dataset contains +14K tangled commits in 10 GitHub repositories, where # concerns in a commit ranges from 2-32. **Manually validated datasets (MVD)** *common datasets* are imperfect because they are synthetic and are not manually inspected. So we refer to [9, 10] to collect a C# and a Java dataset to simulate real-world application scenarios. Each dataset comprises 300 tangled commits, each involving 2∼3 concerns. The Java MVD is from 10 repositories of an international software company to make them understandable to the reviewers of that company. However, due to the lack of sufficient C# repositories in that company, the C# MVD is from the common C# dataset. The datasets are inspected by 10 reviewers each of whom has ≥ 5 years of programming experience, and 50% are complex samples with hidden dependencies. The ratio 50% of complex cases to all cases is the average ratio of Java repositories that the company currently develops and maintains. However, due to the influence of developers' commit habits, the ratio may vary in other companies' repositories.

To split the training and testing sets on each dataset, we follow the procedure of the first trainable model *UTango* [19]: 1) We sort all commits in chronological order, based on when they were created; 2) For trainable *GNN*-based approaches, we use 80% of the oldest commits for training, 10% of the next oldest for tuning, and the latest 10% of the commits for testing; 3) The other baselines do not require training, so we evaluate them using the 10% test data. We set the training parameters for trainable *GNN*-based approaches as follows: 1) We use the hyperparameters from *UTango*'s paper [19] to obtain the best performance of *UTango*. 2) For *HD-GNN*, we adopt the following key hyperparameters to obtain the best performance: Epoch size (50, 100, 200); Batch size (50, 50, 100); Learning rate (0.001, 0.003, 0.005).

5.3 Experimental Setup

5.3.1 Settings for RQ1. **Baselines:** *Barnett et al.* [2]: A *heuristic rule-based* approach considering *def-use*, *use-use*, and *same-enclosing-method* relations among code changes. *Herzig et al.* [9]: A *heuristic rule-based* approach combine various *Confidence Voters* and build triangle partition matrix to untangle the commits. Although *Herzig et al.* only applied their work to Java, we reproduce this work on the C# by following the reproduction process of the previous work [19, 21]. *Flexeme* [21]: A *Graph clustering-based* approach builds a δ-NFG (proposed in *Flexeme*) of commits and then applies agglomerative clustering on this δ-NFG to untangle the commits. *δ-PDG+CV* [21]: A variant of *Flexeme* by applying *Herzig et al.*'s confidence voters directly to δ-PDG (proposed in *Flexeme*). *UTango* [19]: A *Graph clustering-based* approach builds a δ-PDG from commits and then applies GNN and agglomerative clustering on this δ-PDG to untangle the commits. **Evaluation Metrics:** *Accuracy (Acc):* We used three different accuracies proposed in *Flexeme*, *UTango* and our approach respectively. The first one proposed by *Flexeme* is defined as the percentage of the statements that are predicted with a correct cluster/concern in all the statements of a commit: $Accuracy^a = \frac{\#Correctly\ predicted\ nodes}{\#All\ nodes\ in\ graph}$. The second one proposed by *UTango* is similar to $Accuracy^a$ except that it considers only changed statements: $Accuracy^c = \frac{\#Correctly\ predicted\ changed\ nodes}{\#Changed\ nodes\ in\ graph}$. The third one proposed by our approach is defined as the proportion of *untangling relations* that are correctly classified because we transform the clustering problem for code changes into a classification problem for the edges among code changes: $Accuracy^e = \frac{\#Correctly\ predicted\ edges}{\#Edges\ in\ graph}$. Note that the *graph* in the three accuracies above are different; they are δ-NFG, δ-PDG, *Code change relation graph* in *Flexeme*, *UTango* and our approach respectively. Where each node in δ-NFG and δ-PDG is a single statement or conditional expression. While each node in the *Code change relationship graph* is a changed code-fragment/hunk (a pair of consecutive *removed* and consecutive *added* statements) that follows the definition of code change of previous works [2, 9, 23], i.e., *Flexeme* and *UTango* untangle commits at single-statement/code-line granularity, but other approaches consider untangling tangled commits at code-fragment/hunk/diff-region granularity sufficient. In addition, for $Accuracy^e$, indirectly related nodes in *Code change relation graph* should not be untangled. E.g., if nodes A and B are tangled, B and C are tangled, then A and C should also be tangled. If A and C are untangle, the model is not robust.

To apply $Accuracy^a$ and $Accuracy^c$ to our approach, we extend *HD-GNN* to δ-NFG and δ-PDG. That is, for each tangled commit, we construct *Entity Reference Graph* and δ-NFG or δ-PDG, where δ-NFG or δ-PDG is regraded as *Code change relationship graph* and nodes of δ-NFG or δ-PDG is regraded as code changes (i.e, instead of considering each code fragment as a code change, we consider a statement or conditional expression as a code change). And then we evaluate the *HD-GNN*'s $Accuracy^a$ on δ-NFG and *HD-GNN*'s $Accuracy^c$ on δ-PDG. The entity types of *Entity Reference Graph*

in Java are replaced by the corresponding entity types in C#. In addition, for $Accuracy^a$ and $Accuracy^c$, an approach might label the nodes with a different permutation of cluster labels than the ground truth. For example, we have five nodes and three concerns. A model may predict [1,2,2,3,3] and the ground truth has [3,1,1,2,2]. A naive score would give an accuracy of 0.0. However, a permutation of the labels for the clusters would actually give an accuracy of 1.0. So we adopt the solution of *Flexeme* and *UTango*: use the Hungarian algorithm [16] to find the permutation that gives the maximum accuracy. We apply this solution to both $Accuracy^a$ and $Accuracy^c$. To apply $Accuracy^e$ to *Flexeme*, *δ-PDG+CV* and *UTango*, because these approaches untangle commits at code-line granularity, after getting the results of untangling the nodes in *δ*-NFG or *δ*-PDG, we cover the nodes in *δ*-NFG or *δ*-PDG (i.e., a statement or conditional expression) with nodes in *Code change relation graph* (i.e., code fragments). Then cluster each pair of code fragments that contains more pairs of statements that belong to a concern than pairs of statements that do not belong to a concern. Then we evaluate these works' $Accuracy^e$ on *Code change relation graph*.

5.3.2 Settings for RQ2. **Baselines:** *Barnett et al.*, *Herzig et al.* and *UTango* mentioned in *Setting for RQ1*. Although *Barnett et al.* restricted their work to C#, *Shen et al.* [23] reproduce this work on their Java dataset, considering only def-use, use-use, and same-enclosing-method relations on their proposed *Diff Hunk Graph*. So we adopt this reproduced work, also called *Base-3* in [23]. *SmartCommit* [23]: A *Graph clustering-based* interactive approach builds a *Diff Hunk Graph* of commits and then proposes graph partitioning algorithm on this graph to untangle the commits. And the baselines used in [23]: *Base-1* (a rule-based approach that puts all changes into one concern), *Base-2* (a rule-based approach that puts the changes in each file into one concern). **Evaluation Metrics:** In this RQ, since *Flexeme* only works on C# and is therefore not in the baselines for this RQ, we do not use the $Accuracy^a$ proposed in their paper. We only use the $Accuracy^c$ and $Accuracy^e$ introduced in *Settings for RQ1*. And the specific comparison process is the same as in *Settings for RQ1*.

5.3.3 Settings for RQ3. **Baselines:** For both the C# and Java datasets, the above *Barnett et al.*, *Herzig et al.* and *UTango* baselines are used. In addition, *Flexeme* and *δ-PDG+CV* only work on C#, so they are used as baselines on the C# dataset. And *SmartCommit* only works on Java, so it is used as a baseline on the Java dataset. **Evaluation Metrics:** The time (second) taken to untangle a tangled commit.

5.4 Experiment Results

Answer to RQ1: Table 3 shows the comparison of metrics on the common C# dataset, where CL:Commandline, CM:CommonMark, HF:Hangfire, HU:Humanizer, LE:Lean, NA:Nancy, NJ:Newtonsoft.Json, NI:Ninject, RS:RestSharp, Avg: Average of the accuracies of #Cs=2 and 3, OA: Overall accuracy of #Cs=2 and 3, *: No avail data point. More specifically, #Cs means the number of concerns in a commit. Avg means the average of the accuracies of the commits with #Cs=2 and #Cs=3, and OA means the overall accuracy of the commits with #Cs=2 and 3. As seen, *HD-GNN* achieves best performances compared with SOTA approaches. Specifically, *HD-GNN* improves *Barnett et al.*, *Herzig et al.*, *δ-PDG+CV* *Flexeme* and *Utango* by 43%, 22%, 16%, 18%, 6% for OA $Accuracy^c$; by 78%, 24%, 8%, 10%, 2% for OA $Accuracy^a$; by 79%, 34%, 14%, 19%, 3% for OA $Accuracy^e$. Some data points are unavailable since those commits are older in the chronological order and appear only in the training data, but not in the testing data. Fig. 8 shows the results of *HD-GNN* on C# MVD are far superior to other approaches, which may because there are 50% complex samples containing hidden dependencies in MVD.

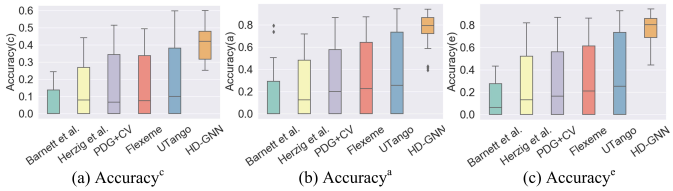


Figure 8: Performance on the C# dataset of MVD.

Answer to RQ2: Similarly, table 4 shows the comparison on the common Java dataset. where SB: spring-boot, ES: elasticsearch, RJ: RxJava, GU:guava, RE: retrofit, DU: dubbo, GH: ghidra, ZX: zxing, DR: druid, EB: EventBus, Avg: Average of the accuracies of #Cs=2 and 3, OA: Overall accuracy of #Cs=2 and 3. As seen, *HD-GNN* achieves best performances compared with SOTA approaches. Specifically, *HD-GNN* improves *Barnett et al.*, *Herzig et al.*, *Base-1*, *Base-2*, *SmartCommit* and *Utango* by 20%, 13%, 20%, 18%, 14%, 3% for OA $Accuracy^c$; by 27%, 25%, 42%, 19%, 23%, 6% for OA $Accuracy^e$. And Fig. 8 shows the results of *HD-GNN* on Java MVD are far superior to other approaches.

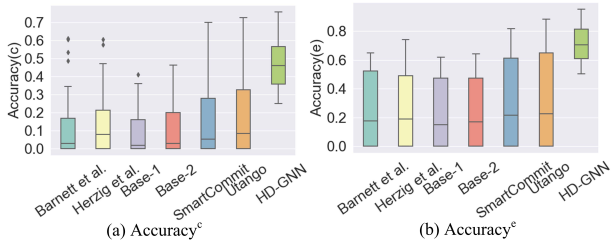


Figure 9: Performance on the Java dataset of MVD.

Answer to RQ3: Figure 10 shows that the def-use chain technique of [2] is by far the fastest, due to its lightweight, heuristic, rule-based approach. *Base-1* and *Base-2* do not use any algorithms, only indexing information is needed to derive the result. Therefore, these two baselines consume very little time. In addition, although our approach uses a deep learning model, the testing time does not significantly exceed that of previous *heuristic rule-based* approaches due to the efficient representation capabilities of GNNs. And *HD-GNN* is 28 times faster than Herzig et al. because the Herzig et al. technique requires n^2 shortest path computations when n are #code changes (analysed by the authors of *Flexeme*).

6 DISCUSSION OF ERROR CASES

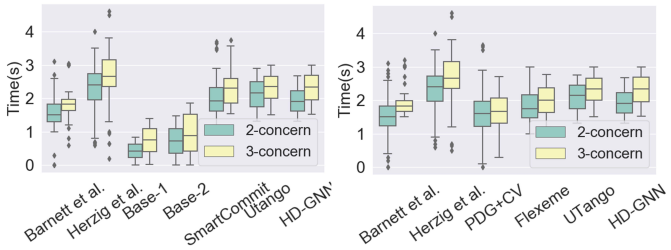
In the experiments of RQ1 and RQ2, the proposed approach still misclassifies some cases. To improve the proposed approach in

Table 3: Performance on the common C# Dataset ($Accuracy^c$%/$Accuracy^a$%/$Accuracy^e$%)

	Barnett et al.		Herzig et al.		δ-PDG + CV		Flexeme		UTango		**HD-GNN**	
#Cs	Avg	OA	Avg	OA	Avg	OA	Avg	OA	Avg	OA	**Avg**	**OA**
CL	14/ 20/ 18	14/ 19/ 16	28/ 58/ 47	28/ 64/ 57	34/ 81/ 76	34/ 80/ 74	34/ 87/ 73	34/ 82/ 68	46/ 90/ 87	46/ 90/ 87	**52/ 91/ 89**	**52/ 91/ 89**
CM	*/ 20/ 18	*/ 20/ 18	*/ 65/ 57	*/ 65/ 57	*/ 90/ 83	*/ 90/ 83	*/ 70/ 58	*/ 70/ 58	*/ */ *	*/ */ *	*/ */ *	*/ */ *
HF	12/ 15/ 13	11/ 15/ 12	28/ 62/ 53	28/ 64/ 58	36/ 86/ 80	35/ 87/ 82	33/ 77/ 64	31/ 79/ 66	46/ 88/ 85	45/ 86/ 83	**52/ 90/ 88**	**48/ 89/ 87**
HU	13/ 25/ 22	13/ 18/ 16	27/ 53/ 46	27/ 62/ 55	30/ 69/ 58	30/ 69/ 58	33/ 70/ 61	33/ 81/ 72	44/ 89/ 86	44/ 89/ 86	**50/ 90/ 89**	**50/ 90/ 89**
LE	7/ 16/ 14	8/ 18/ 16	27/ 66/ 57	29/ 69/ 58	35/ 78/ 65	35/ 84/ 77	35/ 80/ 72	33/ 80/ 73	46/ 89/ 87	45/ 88/ 85	**50/ 91/ 90**	**48/ 90/ 88**
NA	*/ 9/ 7	*/ 9/ 7	*/ 63/ 54	*/ 67/ 58	*/ 83/ 72	*/ 86/ 75	*/ 87/ 73	*/ 84/ 69	*/ */ *	*/ */ *	*/ */ *	*/ */ *
NJ	7/ 13/ 11	7/ 15/ 12	28/ 64/ 56	28/ 71/ 66	34/ 78/ 69	34/ 82/ 78	27/ 62/ 53	27/ 71/ 64	41/ 83/ 80	41/ 83/ 80	**45/ 86/ 83**	**45/ 86/ 83**
NI	10/ 14/ 12	10/ 14/ 12	26/ 57/ 47	26/ 57/ 47	37/ 94/ 89	37/ 94/ 89	32/ 80/ 68	32/ 80/ 68	46/ 91/ 89	46/ 91/ 89	**54/ 93/ 91**	**54/ 93/ 91**
RS	10/ 13/ 11	9/ 12/ 9	31/ 70/ 60	31/ 70/ 59	33/ 64/ 57	31/ 70/ 61	34/ 86/ 77	33/ 82/ 73	46/ 88/ 86	43/ 87/ 85	**47/ 90/ 88**	**48/ 89/ 87**
OA	7/ 13/ 11	8/ 13/ 11	28/ 66/ 55	29/ 67/ 56	35/ 84/ 78	35/ 83/ 76	34/ 83/ 75	33/ 81/ 71	46/ 90/ 88	45/ 89/ 87	**50/ 90/ 89**	**51/ 91/ 90**

Table 4: Performance on the common Java Dataset ($Accuracy^c$%/$Accuracy^e$%)

	Barnett et al.		Herzig et al.		Base-1		Base-2		SmartCommit		UTango		**HD-GNN**	
#Cs	Avg	OA	Avg	OA	Avg	OA	Avg	OA	Avg	OA	Avg	OA	**Avg**	**OA**
SB	21/ 62	19/ 61	25/ 67	23/ 65	14/ 35	12/ 32	23/ 65	21/ 62	29/ 71	28/ 70	32/ 80	30/ 77	**37/ 83**	**35/ 81**
ES	27/ 64	24/ 63	32/ 70	30/ 69	19/ 46	16/ 45	22/ 68	21/ 63	34/ 78	33/ 75	36/ 86	35/ 75	**40/ 90**	**38/ 87**
RJ	24/ 65	22/ 63	27/ 67	26/ 65	18/ 37	17/ 35	28/ 73	27/ 72	31/ 76	30/ 74	33/ 81	32/ 77	**38/ 87**	**36/ 86**
GU	25/ 66	24/ 64	27/ 68	26/ 66	21/ 51	20/ 50	31/ 75	29/ 73	33/ 77	31/ 76	37/ 82	35/ 78	**42/ 89**	**40/ 87**
RE	19/ 59	17/ 57	25/ 65	24/ 63	22/ 52	21/ 51	24/ 68	22/ 65	29/ 72	29/ 70	34/ 83	33/ 78	**36/ 85**	**35/ 84**
DU	18/ 55	18/ 58	20/ 62	19/ 61	15/ 37	14/ 33	19/ 60	17/ 58	24/ 70	23/ 68	30/ 79	29/ 75	**32/ 81**	**30/ 80**
GH	24/ 72	22/ 70	27/ 68	25/ 63	22/ 52	21/ 50	27/ 70	26/ 65	33/ 79	32/ 75	35/ 82	33/ 76	**40/ 89**	**36/ 87**
ZX	20/ 64	17/ 62	26/ 65	24/ 63	21/ 50	21/ 47	29/ 71	27/ 68	34/ 80	33/ 74	38/ 85	35/ 82	**42/ 90**	**34/ 89**
DR	18/ 54	15/ 61	29/ 69	27/ 65	17/ 44	14/ 43	29/ 70	26/ 67	32/ 74	30/ 73	34/ 84	33/ 79	**39/ 88**	**39/ 90**
EB	21/ 60	19/ 57	24/ 64	23/ 63	13/ 36	12/ 35	22/ 69	20/ 64	27/ 72	26/ 70	31/ 79	29/ 75	**36/ 86**	**35/ 85**
OA	17/ 65	16/ 62	23/ 65	23/ 64	17/ 50	16/ 47	24/ 68	22/ 66	30/ 75	29/ 74	34/ 81	33/ 83	**38/ 90**	**36/ 89**



(a) Time on the C# dataset (b) Time on the Java dataset

Figure 10: Time (s) on the common datasets.

the future, we randomly selected 50 cases of all error cases for manual analysis. Representative error cases are shown in figure 11, where the code repositories *spring4-sandbox* and cybergarage-upnp are datasets from RQ2, and *commandline* is a dataset from RQ1. 22 cases are related to formatting adjustments. As shown in both Figures 11(a) and (b), the proposed approach ignores the 1^{st} code change, which is related to the 2^{nd} code change. We speculate that the changes in (a) and the 1^{st} change in (b) are related to minor formatting adjustments. The code changes in both (a) and (b) have significant differences in context and internal code structure. 16 cases are related to code clean, as shown in 11(c). 12 cases are related to manual conflict resolution, as shown in 11(d). The code changes in (c) and (d) have significant internal and contextual differences, making it difficult for the proposed approach to detect the hidden dependencies between them. If these trivial code changes are not properly classified, they can affect the accuracy of downstream tasks such as bug fixes and refactoring detection. Overall, our approach essentially detects hidden dependencies between code changes by learning the syntax and structural information in commits, but without incorporating many external details such as commit messages, human observable heuristic rules, etc. to improve the accuracy of the final results. In future work, we will combine various external information to improve the effectiveness of the proposed approach.

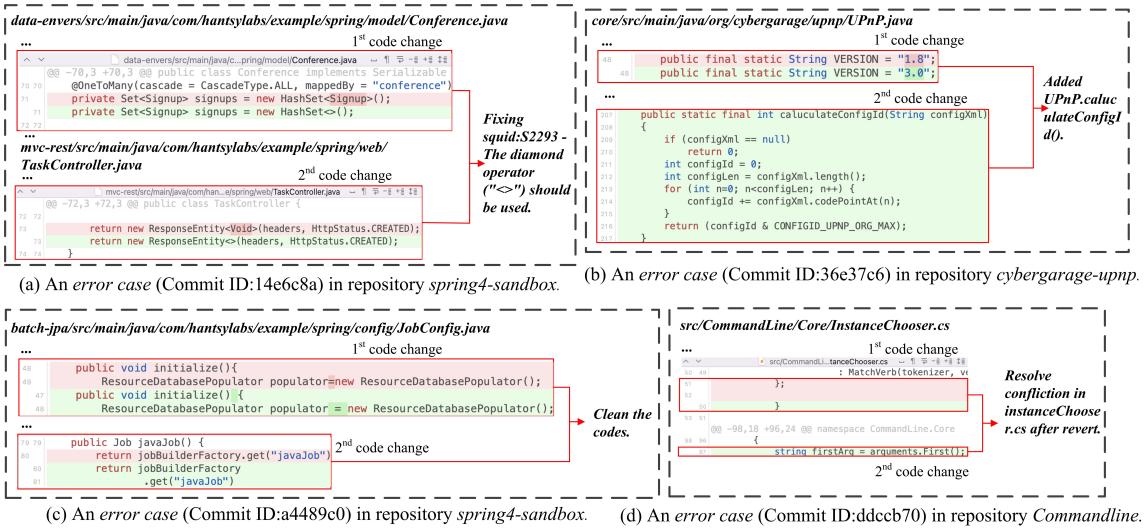
7 THREAT TO VALIDITY

Internal validity. The labels of C# MVD may contain inaccurate labels because of the difficulty of understanding the code not written by our own team. The entity reference graph requires static analysis tools to model the code. However, the accuracy and reliability of static analysis tools can be inaccurate, especially when using complex reflections, resulting in inaccurate analysis results.
External validity. The *common datasets* is not inspected. So, we created MVDs to further validate the effectiveness of the proposed approach. As mentioned in the second section, the types of concerns are ambiguous. Although we have identified the shortcomings of existing approaches and our approach enhances the possibility of capturing all hidden dependencies from a technical perspective, it is also difficult for us to summarize all the patterns of these hidden dependencies.

8 RELATED WORKS

8.1 Heuristic rule-based approaches

Herzig and Zeller [10] split each code change into a set of individual change operations that added or deleted method calls or method definitions. And they using multiple confidence voters to expects the relation between each pair of change operations. Kirinuki et al. [15] propose a technique that extracts code changes between every



(a) An *error case* (Commit ID:14e6c8a) in repository *spring4-sandbox*.
(b) An *error case* (Commit ID:36e37c6) in repository *cybergarage-upnp*.
(c) An *error case* (Commit ID:a4489c0) in repository *spring4-sandbox*.
(d) An *error case* (Commit ID:ddccb70) in repository *Commandline*.

Figure 11: Examples of error cases in the experiments of RQ1 and RQ2. Among them, (a) and (b) omitted minor symbols or numerical changes. (c) Related to code cleaning. (d) Related to manually resolving code conflicts.

two consecutive revisions and stores them into a database at first, and then suggests whether a developer should split a given commit and how to split it when he/she is about to commit changes based on heuristic rules. Similarly, [27] adopted CoRA (Code Review Assistant) based on code dependency analysis, tree-based similar code detection, and identify the most important code changes in each part based on the PageRank algorithm to untangle *tangled commits* for code review. [2] proposed ClusterChanges, which uses def-use and use-use relations as the primary organizing principle for clustering changes.

8.2 Feature-based approaches

[5] developed EpiceaUntangler to untangle tangles by collecting fine-grained features (that is, whether two changes occur in the same package, class and / or method, whether variables accessed happen between the two changes) of code changes and applying a clustering algorithm using the defined features. Recently, Yamashita et al. followed the study of Dias et al, and [30] introduced ChangeBeadsThreader (CBT), an interactive environment [5]. The initial clustering in CBT follows a simplified variant of the technique in [5] , and then allows the user to split/merge clusters interactively by developers.

8.3 Graph clustering-based approaches

Flexeme [21] build a multi-version Program Dependency Graph augmented with name flows for each tangled commit, and annotates data flow edges with the names/lexemes that flow across them. Then applies Agglomerative Clustering using Graph Similarity to that graph to untangle the corresponding tangled commit. SmartCommit [23] is a graph partitioning-based interactive approach to untangle tangled commits. It adopts an extensible graph representation, named Diff Hunk Graph, to capture comprehensive semantic and heuristic relations among at first. An edge shrinking algorithm is then used to partition the diff hunk graph into a set of subgraphs, each representing a concern of closely related diff hunks, as the initial untangling results. Developers can review the untangling results to decide whether or not to adjust them. [19] present *UTango*. They develop a graph-based code change representation learning model at first, and use Label, Graph-based Convolution Network to produce the embeddings for code changes secondly, finally cluster the code changes by agglomerative clustering algorithm.

CONCLUSION

In collaborative software development, instead of following the best practice of making commits, i.e. making small, single-purpose commits, developers often make *tangled commits* that implement multiple development concerns. *Tangled commits* often obfuscate the change history of software repositories and bring challenges to efficient collaboration among developers and software maintenance. Previous approaches have evolved from human-specified *Heuristic rule-based*/*Feature-based* approaches to automatic *graph-clustering-based* approaches. However, in these works, it is difficult to detect hidden dependencies among code changes in global context. To solve this problem, we model the *tangled commit* as a graph at finer-grained hierarchical levels. The graph captures the code dependencies among the entities and the inclusion relationship between a statement and its entities in the global context. And then we propose *HD-GNN*, which could learn the relation among code changes and determine whether a pair of code changes has hidden dependencies. The evaluation results show that the performance of *HD-GNN* is superior to the SOTA approaches without sacrificing efficiency in time.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant Nos. 62192731, 62192730 and 61690200.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[2] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 134–144.

[3] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[4] Siyu Chen, Shengbin Xu, Yuan Yao, and Feng Xu. 2022. Untangling Composite Commits by Attributed Graph Clustering. In *Internetware 2022: 13th Asia-Pacific Symposium on Internetware, Hohhot, China, June 11 - 12, 2022*. ACM, 117–126.

[5] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. IEEE Computer Society, 341–350.

[6] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1263–1272.

[7] Xiaojie Guo, Liang Zhao, Cameron Nowzari, Setareh Rafatirad, Houman Homayoun, and Sai Manoj Pudukotai Dinakarrao. 2019. Deep Multi-attributed Graph Translation with Node-Edge Co-Evolution. In *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*. IEEE, 250–259.

[8] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 1024–1034.

[9] Kim Herzig, Sascha Just, and Andreas Zeller. 2016. The impact of tangled code changes on defect prediction models. *Empir. Softw. Eng.* 21, 2 (2016), 303–336.

[10] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*. IEEE Computer Society, 121–130.

[11] C. Horstmann and G. Cornell. 2004. *Core Java 2, Volume 1: Fundamentals (7th Edition)*.

[12] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

[13] Thomas N. Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *CoRR* abs/1611.07308 (2016).

[14] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[15] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Splitting Commits via Past Code Changes. In *23rd Asia-Pacific Software Engineering Conference, APSEC 2016, Hamilton, New Zealand, December 6-9, 2016*. IEEE Computer Society, 129–136.

[16] Harold W. Kuhn. 1955. The Hungarian Method for the Assignment Problem. In *Naval Research Logistics Quarterly 2, 1-2*. 83–97.

[17] Guohao Li, Chenxin Xiong, Ali K. Thabet, and Bernard Ghanem. 2020. DeeperGCN: All You Need to Train Deeper GCNs. *CoRR* abs/2006.07739 (2020).

[18] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated Graph Sequence Neural Networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[19] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. UTANGO: untangling commits with context-aware, graph-based, code change clustering learning model. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 221–232.

[20] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. 2013. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*. IEEE Computer Society, 138–147.

[21] Profir-Petru Pârtachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: untangling commits using lexical flows. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 63–74.

[22] Peter C. Rigby, Brendan Cleary, Frédéric Painchaud, Margaret-Anne D. Storey, and Daniel M. Germán. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Softw.* 29, 6 (2012), 56–61.

[23] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: a graph-based interactive assistant for activity-oriented commits. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 379–390.

[24] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. ACM, 51.

[25] Yida Tao and Sunghun Kim. 2015. Partitioning Composite Code Changes to Facilitate Code Review. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. IEEE Computer Society, 180–190.

[26] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

[27] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. CoRA: Decomposing and Describing Tangled Code Changes for Reviewer. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1050–1061.

[28] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Peng Cui, Philip S. Yu, and Yanfang Ye. 2019. Heterogeneous Graph Attention Network. *CoRR* abs/1903.07293 (2019).

[29] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[30] Satoshi Yamashita, Shinpei Hayashi, and Motoshi Saeki. 2020. ChangeBeadsThreader: An Interactive Environment for Tailoring Automatically Untangled Changes. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 657–661.

[31] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 783–794.

[32] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 10197–10207.