A Mocktail of Source Code Representations

Dheeraj Vagavolu*, Karthik Chandra Swarna* and Sridhar Chimalakonda
Research in Intelligent Software & Human Analytics (RISHA) Lab
Dept. of Computer Science & Engineering
Indian Institute of Technology Tirupati
Tirupati, India
{cs17b028, cs17b026, ch}@iitpt.ac.in

Abstract—Efficient representation of source code is essential for various software engineering tasks such as code classification and code clone detection. Most recent approaches for representing source code still use AST and do not leverage semantic graphs such as CFG and PDG. One effective technique for representing source code involves extracting paths from the AST and using a learning model to capture program properties. Code2vec is one such path-based approach that uses an attention-based neural network to learn code embeddings which can then be used for various downstream tasks. However, this approach uses only AST and does not leverage CFG and PDG. Even though an integrated graph approach (Code Property Graph) exists for representing source code, it has only been explored in the domain of software security. Moreover, it does not leverage the paths from the individual graphs. Our idea is to extend the path-based approach code2vec to include the semantic graphs CFG and PDG with AST, which is largely unexplored in software engineering. We evaluate our approach on the task of METHODNAMING using a C dataset of 730K methods collected from GitHub. In comparison to code2vec, our approach improves the F1 score by 11% on the full dataset and up to 100% with individual projects. We show that semantic features from the CFG and PDG paths drastically improve the performance of the software engineering tasks. We envision that looking at a mocktail of source code representations for various software engineering tasks can lay the foundation for a new line of research and a rehaul of existing research.

I. INTRODUCTION

Due to the drastic rise and availability of enormous volumes of source code in open-source projects and tools to extract and analyze them [1]–[3], researchers have explored many ways of solving software engineering problems such as code classification [4], [5], code clone detection [6]–[8], code summarization [9], [10], method name prediction [11], and source code retrieval [12], [13]. Representing source code is central to all of these software engineering tasks [14] and is influential in determining the performance of the approaches.

Many of the existing approaches utilize tree and graph-based representations such as AST [6], [7], [14]–[19] and CFG [20], [21] to represent source code for diverse tasks. In their work, Zhang et al. [14] use sub-trees extracted from the AST with tree-based CNN to generate code vectors. Li et al. [22] use the global program dependencies of source code and the local ASTs for predicting bugs. Some works also use Data Flow Graphs to represent source code [23], [24].

Alon et al. [25] proposed representing a code snippet as a set of paths extracted from its AST. The advantage of using this path-based representation is that program features need not be manually designed for a specific task or a specific language. Later, code2vec [26] used these AST paths with an attention-based neural network. This neural network encodes the code’s structural information from individual paths by generating method-level code embeddings, which can then be used for various software engineering tasks. Like most existing approaches [14], [15], the path-based approach places less emphasis on semantic graph structures such as CFG and PDG and instead focuses solely on the AST.

Few approaches have used different kinds of graph structures together in a combined way for representing source code [27], [28]. The Code Property Graph (CPG) by Yamaguchi et al. [27] is one such approach where they have combined CFG, PDG, and AST into a joint static data structure to model and detect software vulnerabilities. The CPG helps express common code vulnerabilities for specific applications, which can be queried by graph traversal queries [29]. Similarly, Allamanis et al. [28] have combined AST, data flow, and control dependency edges in a single graph; however, the graph edges are hand-crafted for a specific language and task [26]. Although both of these works use different graph structures, they do not leverage the paths within each graph, which can flexibly represent code for diverse tasks. Also, the CPG is modeled explicitly for the domain of software security [27].

Can we leverage paths from various source code representations to support diverse software engineering tasks? With this line of thought, this paper is the first attempt to integrate paths from both structural and semantic code representations and demonstrate its efficacy. We extend the code2vec model [26] to include paths extracted from various graph representations, namely AST, CFG, and PDG. We choose code2vec as it is still used for various tasks and many of the recent approaches are built upon it [30]–[33]. Extending the code2vec model with CFG and PDG can better capture the code semantics, which is not possible with AST alone. Similar to code2vec, the proposed approach gives optimal weights to the individual paths extracted from CFG and PDG along with AST.

The core argument of this paper is to demonstrate that integrating semantic structures (such as CFG, PDG) with syntactic structures (AST) can predominantly improve the performance of software engineering tasks.

*Authors have contributed equally
As a first step, we evaluate our approach on the task of METHOD NAMING using a custom C language dataset of around 730K methods. The comparative evaluation of both code2vec and our model shows an increase of 11% in F1 score on the whole dataset. However, when evaluated on different projects in our dataset, we see an increase of F1 score up to 100%. Moreover, our results show that diverse approaches built upon code2vec for various downstream tasks [30]–[33] can be enhanced by including paths from CFG and PDG. Thus, we believe considering a mocktail of the syntactic and semantic structures can lead to a new direction in representing source code while also improving existing works that rely solely on AST, CFG, or PDG.

II. METHODOLOGY

A. Background

In their work, Alon et al. [25] proposed representing a code snippet as a set of paths in its AST. They have used the AST paths with Conditional Random Fields (CRF) and evaluated the approach on METHOD NAMING. Later, code2vec [26] used these AST paths with an attention-based neural network and gained better results than the CRF approach. We briefly explain the concepts of AST path and path contexts:

**Definition 1 (AST Path).** Let $n$ represent a node in an AST, and it has two attributes - the type of node $d$ and the code token $t$. A path between nodes in the AST that begins at a terminal node $n_1$, goes through a series of intermediate non-terminal nodes $n_2, \ldots, n_k$, and ends at another terminal $n_{k+1}$ is called an AST path of length $k$. The path $p$ is represented by a sequence of the form: $d_1 a_1 d_2 a_2 \ldots d_k a_k d_{k+1}$, where $d_1, d_2, \ldots, d_{k+1}$ are the types of the nodes $n_1, n_2, \ldots, n_{k+1}$ respectively, and $a_1, a_2, \ldots, a_k$ are the directions of movements between path nodes in the AST. Here, $a_i \in \{↑, ↓\}$.

**Definition 2 (AST Path Context).** It is a tuple $\langle t_1, p, t_{k+1} \rangle$, where $t_1$ and $t_{k+1}$ are the tokens associated with the AST nodes $n_1$ and $n_{k+1}$ respectively, i.e., terminal nodes of $p$.

B. Defining semantic path contexts

Some researchers have used semantic features such as control flows and data dependencies to a specific problem [28], [34], but not many approaches have attempted to combine syntax and code semantics. The AST paths do not capture some semantic aspects like control flow and program dependencies. To overcome this issue, we have extended the concept of paths to CFG and PDG at the method level:

**Definition 3 (CFG Path).** Let $n$ represent a node in a CFG, and it has two attributes - the type of node $d$ and the code token $t$. A path between nodes in the CFG that begins at the METHOD node $n_1$, goes through a series of intermediate nodes $n_2, \ldots, n_k$, and ends at a node $n_{k+1}$ is called a CFG path of length $k$. The last node $n_{k+1}$ can be of two types:

- The METHOD\_RETURN node,
- A previously visited intermediate node that represents a loop control structure (i.e. $n_{k+1} \in n_2, \ldots, n_k$.)

The intuition behind this approach is that each AST path captures a unique structural template for a set of statements. For the code snippet in Fig. 1, we show the AST and an example AST path in Fig. 2. This example AST path represents the statement $flag = true$. Further, similar assignment statements in other code snippets (say $a = b$) have identical tree structures regardless of the variable names and hence, can be represented by the same AST path. Thus, we can identify structurally similar code snippets by using a set of AST paths.

![Fig. 2. Extracting AST, CFG, and PDG paths from the code snippet in Fig. 1](image-url)
The path \( p \) is represented by a sequence of the form: \( d_1 \downarrow d_2 \downarrow \ldots \downarrow d_k a_k d_{k+1} \), where \( d_1, d_2, \ldots, d_{k+1} \) are the types of the nodes \( n_1, n_2, \ldots, n_{k+1} \), and the direction \( a_k \) depends upon the last node \( n_{k+1} \). \( a_k \) is \( \downarrow \) if \( n_{k+1} \) is METHOD\_RETURN node, \( \uparrow \) otherwise.

Each of the CFG paths represents a control flow during program execution. To represent loops in CFG, we extract three different paths from it - the first one ignores the loop and proceeds to the next node, the second path goes through the loop only once and proceeds to the next node, the last one goes back to the visited loop node and ends there. These three paths together represent possible executions of a loop. Hence, for this reason, we have two types of last node \( n_{k+1} \) in our definition of the CFG path.

**Definition 4 (PDG Path).** Let \( n \) represent a node in a PDG, and it has two attributes - the type of node \( d \) and the code token \( t \). An edge \( e \) in a PDG has a label \( l \) associated with it. A PDG path \( p \) is a sequence of nodes \( n_1, n_2, \ldots, n_{k+1} \) where all of the edges along the path have the same label \( l_p \). The path \( p \) is represented by a sequence of the form: \( d_1 a_1 d_2 a_2 \ldots d_k a_k d_{k+1} \), where \( d_1, d_2, \ldots, d_{k+1} \) are the types of the nodes \( n_1, n_2, \ldots, n_{k+1} \), and \( a_1, a_2, \ldots, a_k \) are the directions of movements between path nodes in the PDG. Here, \( a_i \in \{ \uparrow, \downarrow \} \).

Since PDG is a combination of CDG and DDG, each PDG path can either represent a control dependence of statements or a data dependence. The labels of edges in a path decide the type of PDG path. Since PDG is a graph (and not a tree), moving up and down does not make sense in PDG. However, we chose to keep all the directions of movements as \( \downarrow \). We do this to have a consistent model input format. The concept of CFG and PDG path contexts are similar to AST path contexts. Fig. 2 depicts the CFG and PDG (in blue and green edges) for the code in Fig. 1 and an example path for each representation.

**C. Approach**

1) **Dataset Collection:** Although some Java datasets created by [9] and [26] exist, it is not straightforward to understand the complex control flows and program dependencies in an object-oriented language like Java. Therefore, we chose an imperative language (C) for our dataset as a first step to evaluate our idea. This approach could later be extended to other languages such as Java. To ensure the quality of the dataset, we use the popularity of the repository as a metric, inline with the existing literature [9], [26]. We determine the most popular projects based on each project’s number of stars and forks and select the top projects with at least 10K methods. We finally collect 16 open-source C projects from GitHub, totaling 730K methods, which is inline with the existing works [9], [26].

2) **Pre-Processing:** This is the path extraction phase. We have developed and used a python-based tool to extract different path contexts from code snippets. Firstly, we divide all the C source files in the dataset into individual methods. Then we use a platform called Joern [27] to generate AST, CFG, and PDG for each method. Finally, we extract paths from these graphs as per the definitions provided above. To account for the high variation in lengths of AST paths, we follow code2vec’s policy and extract only those with a maximum length of 8 and width of 2. An AST path’s width refers to the difference in leaf node indices when all the leaf nodes are indexed sequentially. Further, to avoid high variation in the number of path contexts across methods, we limit the number of AST, CFG, and PDG path contexts to 200, 10, and 150 per method, respectively.

3) **Parallel Attention Pipeline:** To combine AST, CFG, and PDG paths, we extend the code2vec pipeline [26], which takes only AST path contexts as input. Our extended-code2vec model takes multiple types of path contexts as its input in a...
parallel pipeline. The model applies a dense layer on path contexts to create a context vector for each path context. It then aggregates the context vectors by performing a weighted average over them. An Attention Layer in the neural network learns these weights. Each context vector and the weighted average vector are trained and learned concurrently. This process is done for each type of paths (AST, CFG, and PDG) parallelly. Then, a concatenation layer combines the final AST, CFG, and PDG vectors to generate the final code vector. Finally, we use a softmax layer for predicting method names.

Fig. 3 depicts the complete parallel pipeline for our approach.

III. EVALUATION AND RESULTS

We trained and evaluated the model on the task of METHOD NAMING. We shuffle methods from all 16 projects and divide them into Train-Val-Test sets to evaluate the model. Also, we randomly select some projects from our dataset and evaluate the model by treating them as individual datasets. We do this to find how well the model can generalize within a project. On each dataset, we test the model using different combinations of representations. We use the F1 score as the metric as previously done by researchers [9], [26]. The precision and recall are calculated over predicted subwords. The intuition is that the quality of a predicted method name is primarily determined by the subwords used to construct it. When a prediction has a high recall, we can infer that model can predict most of the subwords of the true label. When a prediction has high precision, we can say that most of the subwords in the predicted label are also in the true label. We summarize our results on the METHOD NAMING task in Table I. We can see that by adding additional representations like CFG and PDG, the F1 score increased in all of the cases.

| Dataset/Project | Number of methods | code2vec (AST) | AST + CFG | AST + CFG + PDG |
|-----------------|------------------|----------------|-----------|-----------------|
| Full C dataset  | 729,218          | 47.1           | 48.9      | 52.3            |
| FFmpeg          | 15,790           | 37.9           | 45.4      | 47.3            |
| SumatraPDF      | 16,356           | 13.2           | 28.4      | 33.8            |
| KBEengine       | 21,949           | 23.8           | 36.6      | 41.2            |
| QEMU            | 39,881           | 26             | 32.4      | 35.5            |
| CatBoost        | 54,365           | 37.4           | 46.9      | 49.6            |

IV. DISCUSSION AND LIMITATIONS

Our approach gives a performance gain of 11% over code2vec on the full C dataset. Furthermore, we observe that the model can capture program properties very well within a project, i.e., when trained and evaluated on a single project. For example, for the SumatraPDF project, the performance boost is more than 100%. Thus, the CFG and PDG paths help the model perform well within a project rather than the combined dataset. This behaviour is expected as the combined dataset has functions from different projects which might have diverse programming styles.

Moreover, even after limiting the number of AST paths based on the length and width, the average number of AST paths per method is 74 in our full dataset. In contrast, though they are not limited based on size, the average number of CFG and PDG path contexts are only 2.4 and 10.5, respectively. This introduces a huge problem of data sparsity on the CFG and PDG pipelines. However, in individual projects which are smaller in size, the variation between the average number of AST, CFG, and PDG paths is relatively less, and hence the model can perform well. One possible solution to address this sparsity is that we can use additional semantic features like data flow in addition to control flows (CFG) and program dependencies (PDG). One downside of this approach is the increased effort for pre-processing and training, which can be decided as a trade-off for a respective downstream task.

A drawback to our approach is that since we are not manually designing program features to represent code, some unusual cases like inline assembly may not be appropriately handled while extracting paths. This is a trade-off we should consider while choosing to extract features automatically.

V. CONCLUSION

Moving away from the predominantly common approach of using either syntactic (AST) or semantic (CFG / PDG) code representations, in this work, we proposed the idea of integrating them for efficient representation of source code. To support this idea, we extended code2vec, which currently supports AST paths, and adapted it to include both CFG and PDG path contexts. We evaluated our approach on the METHOD NAMING task with different sets of data, first with the full C dataset of 16 projects and then with some individual projects. We demonstrate that by including CFG and PDG path contexts, the model outperforms code2vec by 11% on the full dataset and up to 100% on individual projects. We observed that the performance boost observed for individual projects is significant than for the entire dataset, potentially owing to the higher variation in the number of AST, CFG, and PDG paths.

Even though the proposed approach can be leveraged to improve the performance of several works that are built upon code2vec, an in-depth evaluation is required for generalization. Further investigation needs to be done to find the adaptability of this approach to various tasks such as bug localization and code generation. Moreover, the proposed approach itself could be modified by experimenting with different kinds and combinations of code representations, datasets, and models for various downstream tasks. We anticipate that the mocktail of source code representations can motivate researchers to replicate existing approaches by integrating syntactic and semantic representations. In addition, we see that the proposed approach can lay the groundwork to spin-off multiple novel source code representations for various sub-domains of software engineering while efficiently leveraging advances in Artificial Intelligence and Natural Language Processing.
