LEARNING TO WALK OVER RELATIONAL GRAPHS OF SOURCE CODE

Pardis Pashakhanloo[†], **Aaditya Naik**[†], **Hanjun Dai**[‡], **Petros Maniatis**[‡], **Mayur Naik**[†]
[†]University of Pennsylvania, [‡]Google Research

ABSTRACT

Information-rich relational graphs have shown great potential in designing effective representations of code for program-understanding tasks. However, the wealth of structural and semantic information in such graphs can overwhelm models, because of their limited input size. A promising approach for overcoming this challenge is to gather presumed-relevant but smaller context from a larger graph, and random walks over graphs was one of the first such approaches discovered. We propose a deep-learning approach that improves upon random walks by learning task-specific walk policies that guide the traversal of the graph towards the most relevant context. In the setting of relational graphs representing programs and their semantic properties, we observe that models that employ learned policies for guiding walks are 6-36% points more accurate than models that employ uniform random walks, and 0.2-3.5% points more accurate than models that employ expert knowledge for guiding the walks.

1 INTRODUCTION

Deep learning techniques have made great strides in solving a wide range of program understanding tasks (Lu et al., 2021). The effectiveness of these techniques on a task depends heavily on the *program representation*. An effective program representation requires the knowledge of rich structural and semantic information in source code (Allamanis et al., 2018; Hellendoorn et al., 2020; Pashakhanloo et al., 2022). Program representation techniques benefit greatly from the abundance of information, but also pose a fundamental challenge. It is difficult to determine which kinds of program information are most relevant for a given task. There have been efforts in crafting features for programs. These approaches, however, require a lot of engineering and domain knowledge. AST- or control-flow-based models (Alon et al., 2019; Allamanis et al., 2018) require domain experts to select what information to include and how to incorporate it.

A recent approach (Pashakhanloo et al., 2022) alleviates this issue by modeling programs as relational graphs that uniformly store *all available* structural and semantic information. CodeTrek outperforms the state-of-the-art by representing programs as biased walks over rich relational graphs, thereby selecting some of the available information via the walk policy. This technique, however, does not provide an automated approach to discovering bias; a domain expert must determine which kinds of information (e.g., statements, successors) are more relevant for a given task. CodeTrek is limited by its reliance on domain knowledge when tackling a new task. Also, using uniform random walks (such as in Perozzi et al. (2014)) does not help because, given a limited budget, as the space of possible random walks increases, it becomes more difficult to select suitable random walks to fit the needs of a given task.

In this paper, we propose a deep learning approach to learn walk policies to address challenges that stem from the abundance of available program information in relational graphs. Our proposed policy-learning mechanism selects relevant relations in a task-specific manner by sampling a set of graph walks over the relations. It uses the properties of relational graphs (relation names, tuples, and key-foreign key relationships) to learn policies that guide the random-walk generation. In particular, it aggregates the attention weights that are computed while the model learns to embed each walk, to rank relations by their relevance to the task. Relations that are ranked higher are of higher relevance, and are more likely to be visited during random traversal. We call the mapping from relation names to their scores the *walk policy*.

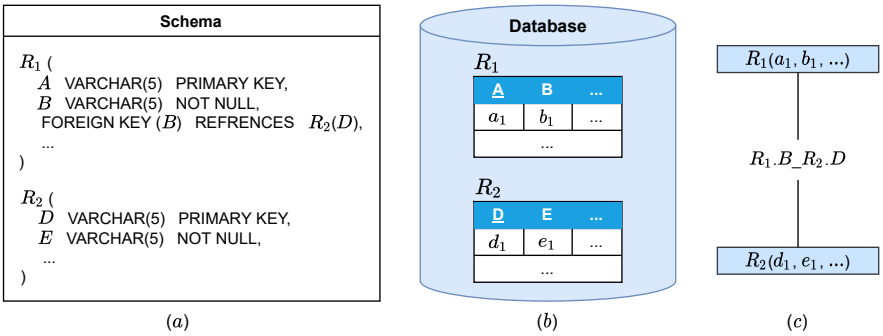

Figure 1: Illustration of (a) the schema of a relational database, (b) tables that correspond to relations that are defined in the schema, and (c) an example of a link in the relational graph.

We evaluate our proposed walk policy learning mechanism on various program-understanding tasks. We observe that the policies learned in this manner yield superior accuracy to even those hand-picked by a human expert.

In summary, this paper makes the following contributions:

1. We present a mechanism to learn walk policies that prunes large relational graphs by ranking the relations based on their relevance in a task-specific manner.
2. We evaluate our approach and show the effectiveness of the proposed mechanism on accuracy.
3. We discuss the limitations of the policy and identify future directions.

2 Overview

2.1 Relational Graphs

In a relational database, information is uniformly stored as relations according to a schema. A schema defines a set of relations containing information in the form of tuples with multiple columns representing different kinds of program elements, and the relationships between them in the form of key-foreign-key (KFK) relationships. Each KFK relationship has the form $R.A \rightarrow S.B$ and states that column A is a foreign key of referencing relation R, and B is a key of referenced relation S. For example, in Figure 1(a), R_1 and R_2 are two relations defined in the schema. The relationship between these relations is established by a FOREIGN KEY entry: R_1 references $R_2.D$ through foreign key B.

A relational graph is the abstraction of a relational database. Every tuple of a relation named R is mapped to a node whose type is R and its values are any value in the tuple other than primary and foreign keys. For example, in Figure 1(b), tuples are $R_1(a_1, b_1, ...)$ and $R_2(d_1, e_1, ...)$. The values a_1, b_1, d_1, and e_1 correspond to attributes $R_1.A$, $R_1.B$, $R_2.D$, and $R_2.E$, respectively. So, as illustrated in blue boxes in Figure 1(c), a tuple $R_1(a_1, b_1)$ whose relation name is R_1 is translated to a node whose type is R_1 and b_1 is its value. For each KFK relationship $R.A \rightarrow S.B$, an edge type $R.A\_S.B$ is defined, connecting the tuples of the two relations with the same value on the edge attributes $R.A$ and $S.B$. For example, in Figure 1(c), assuming that $b_1 = d_1$, the KFK relationship $R_1.B \rightarrow R_2.D$ is mapped to an edge with type $R_1.B\_R_2.D$ between the two nodes.

2.2 Learning a Walk Policy

Representing programs by walking relational graphs of code is challenging because the space of possible walks is combinatorially large. If done randomly, there will be many unrelated walks among the sampled walks that not only do not contribute to improving the program representation, but make it noisier. On the other hand, if walks are guided by a domain expert, the exploration space and noise decreases for the given task at the cost of human engineering effort; moreover, some signal in large graphs may not be obvious to a human expert, which might lead to a loss of opportunity to discover relevant and powerful relations. Learning policies to guide the walks removes this cost.

A natural way to automate walks is to learn another neural network policy that recommends the next node to visit based on the walk history. However, this presents another set of challenges. First, the space is discrete and combinatorial, thus no direct gradient update to the walk policy is available. It is possible to use reinforcement learning for this challenge, but the samples are complex, making training difficult. Regardless of how we approach the above challenges, it may still be difficult to interpret the learned walk policy.

The challenges of walk learning arise from the fact that walks are latent—they are unobserved and lack supervision. However, a learned policy has the potential to improve the overall predictive performance of the model, and thus walk learning can be viewed as an expectation-maximization algorithm (E-M). The E-step is to estimate the walk policy based on the current model, and the M-step is to improve the current model based on the estimated walk policy. The walk policy must now be designed in such a way that it can be estimated by using the current transformer model.

To design a walk policy, we can use some properties of relational graphs such as relation names, tuples, and KFK relationships to learn walk policies. The policy gives each relation name a score which is proportionate to the relevance of the relation to a specific task. These scores specify the next step in a random walk. For example, when the score of the stmt relation is 0.04 and the score of the expr relation is 0.02, it is twice as likely to step to a node of type stmt than a node of type expr, when there is a choice between the two at the next step of a walk. The policy is learned before training the actual model that solves the task. The policy learner updates the policy based on the aggregated relevance of relations in previous iterations of training.

Note that, although we present our work on learning walk policies in the context of relational graphs, there is nothing in our approach that makes it inapplicable to other graphs (e.g., data-flow and control-flow graphs constructed out of syntax trees, such as those used by Allamanis et al. (2018), or even basic-block graphs such as those used by Vasudevan et al. (2021)). Our approach works under the following conditions:

- A walk policy can be expressed in terms of bias over vertex types.
- The program-representation model architecture that uses the walks computes some notion of vertex importance, such as attention scores, that can be used to learn an aggregate measure of importance about vertex types.

We present our results in the context of CodeTrek, but it would be applicable to other architectures such as Transformer-based systems (e.g., Hellendoorn et al. (2020)), or graph-attention networks (Veličković et al., 2018).

3 Learning and Inference

3.1 Walk Policy Learning

We define a *walk policy* over a relational graph as a mapping from every relation name to a score, indicating the relevance of a relation to the task for which the neural network is being trained. These scores are computed prior to training a model for the task. They are then used—while training a model for the given task—to guide sampling a set of walks W over a relational graph.

We revisit here the model architecture from Pashakhanloo et al. (2022). Consider a sequence of node type embeddings $\bar{\mathbf{e}}^{(n)} = [\mathbf{e}_1^{(n)}, ..., \mathbf{e}_N^{(n)}]$, edge type embeddings $\bar{\mathbf{e}}^{(e)} = [\mathbf{e}_1^{(e)}, ..., \mathbf{e}_{N-1}^{(e)}]$, and node value embeddings $\bar{\mathbf{e}}^{(v)} = [\mathbf{e}_1^{(v)}, ..., \mathbf{e}_N^{(v)}]$ that correspond to a walk w with a maximum of N nodes and $N-1$ edges, sampled from a relational graph G. The walk embedding $\mathbf{x}_w$ is computed as follows:

$$\mathbf{e}_w = \bar{\mathbf{e}}^{(n)} \parallel \bar{\mathbf{e}}^{(e)} \parallel \bar{\mathbf{e}}^{(v)} \in \mathbb{R}^{(3N-1)\times d}$$

where $\parallel$ is the concatenation operator. We elaborate on the details of the embedding in subsection 3.2. We compute the self-attention weights for $\mathbf{e}_w$ as follows (Vaswani et al., 2017):

$$\text{Self-Attention}(\mathbf{e}_w) = \text{Attention}(\mathbf{e}_w W^Q, \mathbf{e}_w W^K, \mathbf{e}_w W^V)$$

where W^Q, W^K, and W^V are learned matrices, and $\text{Attention}(Q, K, V) = \text{Softmax}\big(QK^T/\sqrt{d}\big)V$. Intuitively, these weights are the results of comparing each element (node type, edge type, or value) in the walk to all the other elements in it.

For our approach, we compute scores for relation names as follows. We discard all but the first N rows of Self-Attention($\mathbf{e}_w$) that correspond to the node types in walk w. We refer to this tensor as $A_w^{(n)} \in \mathbb{R}^{N \times d}$. We then expand each $A_w^{(n)}$ to a tensor with dimension R (the total number of relation names) so that the weights that correspond to each relation name have their own column, by aggregating the attention weights of all nodes with the same relation name. We then compute a raw score vector S:

$$S = \text{Softmax}\big(\sum_{w \in W} A_w^{(n)}\big) \in \mathbb{R}^R$$

Each row in S corresponds to a relation name. At each training step, we update the scores:

$$S \leftarrow \gamma S + (1 - \gamma) S_{previous}$$

where $\gamma \in [0, 1]$ is a hyperparameter. Updates are repeated until the node types reach a fixed relative ranking. S is the list of scores that we use for guiding the random walks during the training process of a model for a given task. For example, if S['stmt'] is 0.06 and S['expr'] is 0.03, the learned walk policy enforces that the neighbors with node type stmt are 2x more likely to be visited by the random walk generator than neighbors with node type expr.

Stability of Ranking. We empirically observe that the relative rankings converge to a fixed point for all our tasks. Also, the rankings do not change across different training instances of the same task.

3.2 Embedding Graphs

To represent program graphs, we follow the neural representation proposed by Pashakhanloo et al. (2022). It was shown to outperform GGNN, CuBERT, GREAT, and Code2Seq on bug-finding tasks. We embed each walk by treating it as a token sequence of relation names of its nodes, the edge types traversed, and their attribute values, in the order of traversal. Given each walk $w = [n_0, e_0, e_1, \ldots, n_{N-1}]$ consisting of N nodes and $N - 1$ edges, we produce an initial embedding X_w:

$$X_w = X_w^{(n)} \parallel X_w^{(e)} \parallel X_w^{(v)} \in \mathbb{R}^{(3N-1) \times d}$$

where d is the embedding dimension and $\parallel$ represents concatenation, using three corresponding learnable embedding matrices E_n, E_e, E_v. The first segment, $X_w^{(n)}$, represents the N node types (relation names), using an embedding lookup in $E_n \in \mathbb{R}^{R \times d}$, where R is the number of relations. The second segment, $X_w^{(e)}$, represents the $N - 1$ edge types, using an embedding lookup in $E_e \in \mathbb{R}^{I \times d}$, where I is the number of KFK relationships in the database. Finally, the last segment, $X_w^{(v)}$, represents the attribute values of the N nodes; we subtokenize attribute values (using a V-sized WordPiece vocabulary for attribute values), embed each subtoken using $E_v \in \mathbb{R}^{V \times d}$, and pool the subtoken embeddings into each node's attribute embedding. A sinusoidal positional encoding is computed for every segment of X_w. All encodings are concatenated to create the positional encoding of walk w:

$$\mathrm{P}_w = \mathrm{P}_w^{(n)} \parallel \mathrm{P}_w^{(e)} \parallel \mathrm{P}_w^{(v)} \in \mathbb{R}^{(3N-1) \times d}$$

Similar to Pashakhanloo et al. (2022), we use a Transformer encoder Vaswani et al. (2017) to encode walk w as follows:

$$Z_w = \text{Transformer}(X_w + \mathrm{P}_w)$$

Over all the $3N - 1$ elements of the walk on the last layer of Transformer, we perform attention pooling $f : \mathbb{R}^{(3N-1) \times d} \mapsto \mathbb{R}^d$ to obtain a d-dimensional walk embedding tensor $\mathbf{e}_w$.

3.3 Training

We train two models for each task. The first model takes a set of labeled graphs, each of which are represented as a set of uniform random walks and a label, (W, y). It learns a task-specific walk policy while learning to embed walks (see subsection 3.1). The second model takes a set of labeled graph, each of which are represented as a set of walks that are sampled using the learned walk policy, and a label. The hidden layers of the second model are similar to the first model, with an additional fully-connected network to train for predicting labels. Both models also embed each graph as set of walk embeddings $\{\mathbf{e}_w\}_{w \in W}$ using a permutation-invariant operation (CodeTrek uses DeepSet Zaheer et al. (2017)). We use the same cross entropy loss function for both models and optimize using the Adam optimizer.

Task	Input	Description	Type
VARMISUSE	Function and a Variable Access	Is the variable used correctly in the function?	Binary
VARMISUSE⋆	Function	Are all the variables used correctly?	Binary
VARSHADOW	Module	Is any of the global variables shadowed by any of local variables?	Binary
DEFUSE	Function and a Variable Definition	Is the variable definition used in the function?	Binary
DEFUSE⋆	Function	Are all the variables defined in the function used after their definitions?	Binary
EXCEPTION	Module and a Masked Exception Type	What is the exception that the except statement should catch?	Multiclass
EXCEPTION⋆	Function and a Masked Exception Type	What is the exception that the except statement should catch?	Multiclass
OPMISUSE	Function	Are all the binary operators used correctly?	Binary

Table 1: Bug-finding Task Descriptions. The variation of each task is marked by "⋆".

4 EVALUATION

4.1 RELATIONAL SCHEMA

We use the relational schema defined for Python by Semmle to construct a database for each program, and extract the tuples of each relation. Semmle is a code analysis platform which is adapted by GitHub and uses logic queries to help find bugs in codebases. This platform constructs databases from source code and uses CodeQL to query them. To better understand CodeQL, we explain its main componenets: 1) CodeQL databases and 2) CodeQL queries.

CodeQL Database. A CodeQL database stores all the program information including but not limited to AST, def-use, call-graph, and more. Similar to any relational database, CodeQL databases have key-foreign-key relationships, are optimized for program data, and can be queried using a SQL-like query language. Each programming language requires its own extractor because it can be different in terms of the capabilities and paradigm from other languages. For the same reasons, each programming language has its own schema for the database. The Python schema, for instance, specifies control-flow information, dataflow-information, AST, and some meta-information. In contrast, the underlying query evaluater that carries out the analyses is language-agnostic.

CodeQL Queries. The Semmle platform uses CodeQL as its query language. These queries can be used to perform code analysis. There is a pool of CodeQL queries that are used for analyzing programs, and, in our work, to construct feature-rich program graphs.

4.2 EXPERIMENTAL SETUP

We evaluate the proposed approach to learn walk policies using five bug-finding tasks and their variants: VARMISUSE, VARSHADOW, DEFUSE, EXCEPTION, and OPMISUSE. We use the program graphs generated by Pashakhanloo et al. (2022) for training the first four tasks, and ETH Py150 Open for the OPMISUSE task. The number of programs that are used to train, validation, and test these tasks are reported in Table 2. The description of the tasks and their variants are summarized in Table 1. We use accuracy to measure the performance of all the tasks other than DEFUSE. We measure the performance of DEFUSE using AUC score to account for its imbalanced dataset.

4.3 EFFECT OF LEARNING WALK POLICY

We compare the effectiveness of learned walk policy on performance by training three models with the same neural architecture but a different approach to scoring relations. In the first scenario, all relations are equally likely to be visited. In the second scenario, a human *expert* identifies a few relations as the most relevant relations to the task. Particularly, nodes with types stmt, expr, and variable are biased to be visited five times more often than others. In addition, in the EXCEPTION task, the score assigned to type module is 0 to avoid traveling from one function to another through their common module node. As a result, the walks will only go to other functions by taking the call

Task	# Training	# Validation	# Testing
VARMISUSE	700,683	75,468	378,401
VARMISUSE⋆	700,683	75,468	378,401
VARSHADOW	70,183	21,794	39,845
DEFUSE	217,591	52,598	104,111
DEFUSE⋆	33,182	8,149	16,296
EXCEPTION	18,456	2,086	10,334
EXCEPTION⋆	18,456	2,086	10,334
OPMISUSE	457,400	49,800	251,531

Table 2: The number of samples used for training, validation, and testing.

γ	0.0	0.2	0.4	0.5	0.6	0.8	1.0
Accuracy	0.52	0.60	0.61	0.65	0.57	0.56	0.36

Table 3: Sensitivity to γ variations.

graph edges between them. In the final scenario, we use the proposed policy learning mechanism to obtain the relevance of relations without prior knowledge. The results of the models trained under these scenarios are reported in Figure 2. Models with learned policies consistently outperform models without policies by 6% to 36%. Interestingly, in all the tasks, models that use learned policies outperform the models for which a human expert scored the relations, by 0.2–3.5% points.

The improved performance of models can be explained by considering the space of possible walks in each scenario. Choosing scores for walks reduces the space by prioritizing relations that are more relevant to solving a specific task, thus reducing the noise caused by irrelevant relations. To illustrate the usefulness of learning walk policies, consider the example in Listing 1. In this example, the model predicts the type of exception to catch on line 247. It must consider the exceptions raised by the summarize method on an instance of class DiskQueue. Hence, the definition of summarize (line 2) must be considered. In line 3, it calls get_disk_cap which raises an exception. This chain of function calls can be made explicit via a call graph, making it possible to walk along them.

Despite this, the walk space is still too large to be traversed at random. For instance, in Listing 1, there are millions of walks that start from the node that corresponds to the except statement. It is difficult to find walks that connect the except node to raise node under a purely random scenario. We can reduce the exploration space by limiting the exploration to walks that connect the two nodes through the body of the body of the try block since the statements before and after that do not affect the caught exception. The challenge, however, is that only a domain expert can tune relation scores for a new task. Learning such scores eliminates the need for a domain expert for tuning. In Listing 1, the learned policy guides the walk generator to use the except_successor relation that connects the statements in a try block (e.g., line 245) to their corresponding except statement (e.g., line 247) more frequently. Having the walks visit except_successor during the traversal, reduces the number of 16-hop walks by over 95% compared to the unbiased case. Interestingly, this relation was overlooked by the human expert who scored the relations for the EXCEPTION task, justifying why learning the policy results in performing 2% points better than that of the human expert.

4.4 EFFECT OF SCORE UPDATE PARAMETER γ

We examine the effect of changing γ during walk learning on the accuracy of the models that are trained to understand code. As described in subsection 3.1, this value represents the influence of the newly computed versus the previously computed scores during walk learning. We learn the walk policy for the EXCEPTION task using varying values of γ and use them to train different models for the same task. As Table 3 shows, with γ set to 1, the model achieves the same accuracy as with uniform random walks. This outcome is expected because $\gamma = 1$ results in completely discarding the score values computed in the previous training iteration. At the other extreme, $\gamma = 0$ assigns an initially random but fixed score vector to relations throughout the policy learning. Regardless, it performs better than uniform random walks. Based on the empirical results, 0.5 is a reasonable value for γ in our selected tasks.

Listing 1: Example EXCEPTION task.

```
  1:  class DiskQueue:
  2:    def summarize(self):
  3:      dc = self.get_disk_cap()
  4:      ...

  5:    def get_disk_cap(self):
  6:      raise NotImplementedError
        ...

240:  class DiskQueueTest(QueueTest):
241:    chunksize = 100000
242:    self.q = DiskQueue()

243:    def test(self):
244:      try:
245:        self.queue.summarize()
246:        self.q.push('a')
247:      except [MASK]:
248:        ...
```

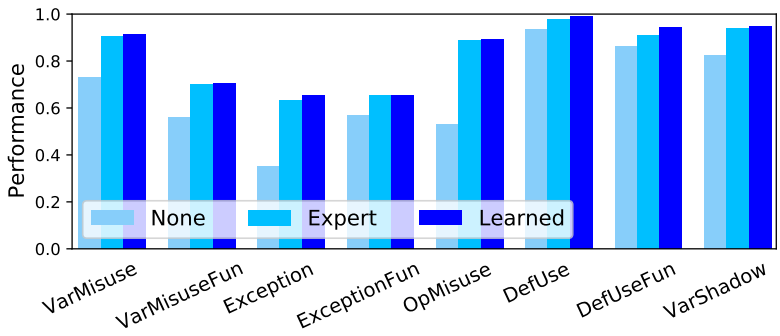

Figure 2: The effect of scoring relations on performance.

4.5 INTERPRETING LEARNED POLICIES

In this section, we take a closer look at the relations that have high scores. We qualitatively analyze some of the relations that are found relevant during the walk policy learning. The top-4 relations with the highest scores are summarized in Table 4 for each task.

For DEFUSE, we compare the relations with the highest learned scores with a CodeQL query crafted for the same purpose. flow_bb_node, successor, and ssa_use are the highest-ranking relations for this task, and all three are crucial for the CodeQL query. The first two relations establish control-flow relationships between basic blocks and statements, and the third relation computes dataflow information. VARSHADOW is another example–a global task which requires inspecting inner scopes (possibly multiple classes) for variables that shadow global variables. Interestingly, the learned policy detects that class is one of the relations with the highest scores.

5 DISCUSSION

One could argue that a walk policy that only scores relations might not be sufficient for some tasks. Consider the following example. In a code database schema, there are KFK relationships between func and both of flow_bb_node and call relations. This means that there is an edge between a node of type func and a node of type flow_bb_node. There is also an edge between a node of type func and a node of type call. Traversing the call-func edges can be interpreted as visiting other functions that

Task	Most Relevant Relations
VARMISUSE, VARMISUSE⋆	stmt, scope, expr_context, expr
VARSHADOW	location, scope, variable, class
DEFUSE, DEFUSE⋆	flow_bb_node, ssa_use, successor, scope
EXCEPTION, EXCEPTION⋆	stmt, expr, variable, except_successor
OPMISUSE	expr, cmpop_list, unnary_op, str

Table 4: Relations with the highest learned scores.

a function calls. The flow_bb_node-func edges, on the other hand, connect function nodes to the root nodes of control-flow graphs of functions' bodies. Depending on whether the task requires inter-procedural (global) or intra-procedural (local) analysis, the relevance of the func relation varies. For such scenarios, we conjecture that scores for edge types should be learned rather than relations types. In future work, we plan to incorporate other factors such as edge names, values, and the length of walks into the learned walk policy. The current mechanism for learning walk policies also lacks memory, which might hinder its ability to capture complex patterns.

Another piece of information that should be specified by an expert is the anchor node. In a practical approach, every walk could start from the root (i.e., the top-level node in the program graph such as a module node) and learn better anchor points from there. There are also open questions regarding anchors; how many nodes should be selected as anchors? Should all the anchor nodes for a particular task have the same relation name?

6 RELATED WORK

Sampling large graphs. Large graphs have been examined in several studies. Some of them perform various sampling schemas such as mini-batching (Zeng et al., 2019; Kipf & Welling, 2016; Hamilton et al., 2017; Chen et al., 2018) or attention mechanism (Velickovic et al., 2017) to enable training on giant graphs. Various efforts have also been made to improve the efficiency of managing and training large graphs, including (Xie et al., 2021; Gandhi & Iyer, 2021; Zhu et al., 2019; Chiang et al., 2019). While these approaches are typically used for recommendation tasks that involve one huge graph, our technique takes into account many program graphs when tackling code understanding problems.

Walk-based embedding. A number of representation learning techniques on graphs use random walks to learn node embeddings. DeepWalk (Perozzi et al., 2014) uses simple unbiased random walks to measure node similarity. node2vec (Grover & Leskovec, 2016) builds on DeepWalk by introducing two hyperparameters to control the tendency for random walks to traverse more depth or breadth. Further extensions include Walklets (Perozzi et al., 2017), which skip over steps in random walks to allow short walks to capture multiple levels of relationships, as well as neural embedding approaches that employ hyperbolic rather than Euclidean spaces to define node similarities (Chamberlain et al., 2017). Unlike these approaches, our proposed technique aims to learn and guide traversal of graphs, rather than depending on simple unbiased random walks.

7 CONCLUSION

We proposed a deep learning approach for learning walk policies over relational graphs. We showed that sampling walks using a learned walk policy can result in models with better performance than that of a policy designed by a human expert. Among the opportunities for future research are: 1) incorporating edge types, values, and other characteristics of walks into the learning of walk policies, 2) expanding the application of the proposed policy learning mechanism beyond code understanding, and 3) providing walk policies with memory (state) to capture more complex patterns.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2019.

Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. Neural embeddings of graphs in hyperbolic space. *arXiv preprint arXiv:1705.10359*, 2017.

Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 257–266, 2019.

Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 551–568, 2021.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 855–864, 2016.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2020.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational representation. 2022.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710, 2014.

Bryan Perozzi, Vivek Kulkarni, Haochen Chen, and Steven Skiena. Don't walk, skip! online learning of multi-scale network embeddings. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pp. 258–265, 2017.

Shobha Vasudevan, Wenjie Jiang, David Bieber, Rishabh Singh, HAMID SHOJAEI, C. Richard Ho, and Charles Sutton. Learning semantic representations to verify hardware designs. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL `https://openreview.net/forum?id=oIhzg4GJeOf`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *stat*, 1050:20, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Anze Xie, Anders Carlsson, Jason Mohoney, Roger Waleffe, Shanan Peters, Theodoros Rekatsinas, and Shivaram Venkataraman. Demo of marius: a system for large-scale graph embeddings. *Proceedings of the VLDB Endowment*, 14(12):2759–2762, 2021.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. *arXiv preprint arXiv:1703.06114*, 2017.

Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.