Science of Computer Programming 237 (2024) 103141

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

TerGEC: A graph enhanced contrastive approach for program termination analysis

Shuo Liu [a], Jacky Wai Keung [a], Zhen Yang [b,*], Yihan Liao [a], Yishu Li [a]

[a] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[b] School of Computer Science and Technology, Shandong University, Qingdao, China

ARTICLE INFO

Keywords:
Code representation learning
Graph neural networks
Contrastive learning
Termination analysis

ABSTRACT

Context: Programs with non-termination behavior induce various bugs, such as denial-of-service vulnerability and memory exhaustion. Hence the ability to detect non-termination programs before software deployment is crucial. Existing detection methods are either execution-based or deep learning-based. Despite great advances, their limitations are evident. The former requires complex sandbox environments for execution, while the latter lacks fine-grained analysis.
Objective: To overcome the above limitations, this paper proposes a graph-enhanced contrastive approach, namely TerGEC, which combines both inter-class and intra-class semantics to carry out a more fine-grained analysis and exempt execution during the detection process.
Methods: In detail, TerGEC analyzes behaviors of programs from Abstract Syntax Trees (ASTs), thereby capturing intra-class semantics both syntactically and lexically. Besides, it incorporates contrastive learning to learn the discrepancy between program behaviors of termination and non-termination, thereby acquiring inter-class semantics. In addition, graph augmentation is designed to improve the robustness. Weighted contrastive loss and focal loss are also equipped in TerGEC to alleviate the classes-imbalance problem during the non-termination detection. Consequently, the whole detection process can be handled more fine-grained, and the execution can also be exempted due to the nature of deep learning.
Results: We evaluate TerGEC on five datasets of both Python and C languages. Extensive experiments demonstrate TerGEC achieves the best performance overall. Among all experimented datasets, TerGEC outperforms state-of-the-art baselines by 8.20% in terms of mAP and by 17.07% in terms of AUC on average.
Conclusion: TerGEC is capable of detecting non-terminating programs with high precision, showing that the combination of inter-class and intra-class learning, along with our proposed classes-imbalance solutions, is significantly effective in practice.

1. Introduction

Termination analysis aims to discover non-terminating programs that are induced by improper code structures or faulty statements. These programs may execute infinitely and incur various bugs, such as denial-of-service vulnerability and memory exhaustion.

* Corresponding author.
E-mail addresses: sliu273-c@my.cityu.edu.hk (S. Liu), Jacky.Keung@cityu.edu.hk (J.W. Keung), zhenyang@sdu.edu.cn (Z. Yang), yihanliao4-c@my.cityu.edu.hk (Y. Liao), yishuli5-c@my.cityu.edu.hk (Y. Li).

https://doi.org/10.1016/j.scico.2024.103141
Received 7 August 2023; Received in revised form 11 May 2024; Accepted 11 May 2024

Available online 15 May 2024

Therefore, many researchers have devoted themselves to this field. Fig. 2 illustrates an example of the non-termination issue. It can be observed that if the initial value of a and b is not equivalent, then the program will not terminate and execute infinitely.

Traditional methods of termination analysis employ formal symbolic reasoning. For example, some studies developed various methods to prove the existence of ranking functions, thereby identifying terminating programs [7–9,28,46,49]. However, since these methods cannot discover non-terminating programs, their detection results always contain lots of false positives. Accordingly, subsequent studies proposed to adopt the recurrence sets to prove the existence of non-terminating programs [13,18,23,40]. Whereas recurrence sets usually need to be singletons, making it difficult to find a recurrence set for a program. Moreover, all of them can not conduct their detection without program execution. Hence, the detection process can be very time-consuming, and the configuration of the sandbox environments for execution is also cumbersome.

In recent years, machine learning-based methods gradually emerged [26,48,54], where they adopt support vector machine and neural networks to simulate ranking functions and recurrence sets during the termination detection, thus they still can not exempt the execution. Different from the above methods, Alon et al. [5] adopted Graph Neural Network (GNN) to directly analyze program semantics on their Abstract Syntax Trees (ASTs). Hence, they can detect non-terminating programs effectively without program execution. Although this approach overcame deficiencies of previous studies, their methodologies are coarse-grained. On the one hand, their modeling process is one-sided. Because they only consider intra-class semantics without learning inter-class semantics when termination detection, the discrepancy between terminating and non-terminating programs will be ignored. On the other hand, their approach also neglects the class-imbalance problem. As a kind of program defect, programs tend to have more terminating instances than non-terminating ones [20] in practice.

To address the aforementioned problems, we propose a graph-enhanced contrastive approach for program termination analysis (TerGEC) in this paper. First of all, TerGEC analyzes program semantics via ASTs, thereby ensuring the comprehension of code both syntactically and lexically. Besides, both intra-class and inter-class learning are considered in TerGEC, which is our main innovation and makes the analysis to be more fine-grained. Moreover, we also proposed a series of solutions in TerGEC to overcome the class-imbalance problem.

In detail, TerGEC first transforms programs into their corresponding ASTs. Subsequently, we propose two graph augmentation methods, namely feature masking and feature projection, which can increase the diversity of training samples, improve the robustness of TerGEC, and is the prerequisite for the following inter-class learning. Afterwards, both the original ASTs and the augmented ASTs are encoded by GNNs, where three GNN models are experimented in this paper, including GCN [39], GAT [50], and GIN [53]. Towards the intra-class semantics, we follow the processes that Alon et al. [5] used and introduce focal loss during the training. As for inter-class semantics, TerGEC utilizes contrastive learning to learn the discrepancy between terminating and non-terminating programs. Finally, we bridge the learning of both intra-class and inter-class perspectives by a hyperparameter and formulate the final optimization target. Thereby a more comprehensive analysis can be accomplished. In addition, during inter-class and intra-class learning, weighted contrastive loss and focal loss are also equipped in TerGEC to alleviate the classes-imbalance problem.

In experiments, we construct four custom datasets with a total of 20k Python programs for termination analysis, which contain much more samples than the previous dataset and reflect the classes-imbalance problem that the previous dataset overlooked. To assess the generalization ability of TerGEC, we also use an existing dataset TPDB [27] with 206 C programs for extra experiments. Subsequently, we compare TerGEC with six state-of-the-art baselines on the aforementioned five datasets. Experimental results show that TerGEC outperforms baselines by 4.59% - 15.56% in terms of mAP on dataset 125M_HumanEval. And similarly, the increments of mAp can up to 2.56% - 17.79% on dataset 125M_MBPP, 0.18% - 11.38% on dataset 1_3B_HumanEval, 1.00% - 10.60% on dataset 1_3B_MBPP, and 3.17% - 15.18% on dataset TPDB. As for the AUC metric, TerGEC achieves the best on datasets 125M_MBPP, 1_3B_MBPP, and TPDB, where the improvements are 5.10% - 27.55%, 1.25% - 42.38%, and 2.69% - 34.87% respectively, although it underperforms a little on the other two datasets. Besides, we design a series of ablation experiments to analyze the effectiveness of the graph augmentation module and the balance factors used in the weighted contrastive loss and the overall loss. Extensive experiments demonstrate the graph-enhanced contrastive approach that combines intra-class and inter-class learning, together with our proposed classes-imbalance solutions, can significantly improve the performance of non-terminating program detection in practice.

The main contributions of this paper can be summarized as follows:

1. We propose a graph-enhanced contrastive approach that simultaneously captures the intra-class and inter-class semantics of programs, making the termination analysis to be more fine-grained.
2. We propose two graph augmentation strategies to supplement samples and increase its diversity, and introduce weighted contrastive loss and focal loss to alleviate the classes-imbalance problem.
3. We construct four custom termination detection datasets with totally 20K samples. At the time of writing, it is the first large-scale dataset for termination analysis. We also open source our replication package,[1] including the custom datasets and the source code of TerGEC.
4. We carry out extensive experiments to evaluate the performance of TerGEC on both Python and C language programs. The experimental results demonstrate the effectiveness and generalization ability of TerGEC.

The remainder of this paper is organized as follows. Section 2 presents the related work of termination analysis, GNN and graph contrastive learning. Section 3 indicates some background knowledge. Section 4 and section 5 elaborate on the proposed approach

[1] https://zenodo.org/records/10703451.

and data preparation. Section 6 discusses the evaluation results and corresponding analyses. Section 7 demonstrates the threats to validity. In the end, Section 8 concludes the overall work.

2. Background and related work

2.1. Termination analysis

Termination is associated with the liveness and safety properties of programs, which means termination is crucial to state the correctness of programs [4]. Proving termination is vital for specific domains like term rewrite systems [57] and GPU programming [36], and therefore research about it emerged constantly. To prove termination, many existing methods intend to search for ranking functions, which map program states to certain values [26]. In addition, these functions are monotonic decreasing after every loop iteration, and are lower bounded. Due to the existence of a ranking function is equivalent to the termination behavior of given programs, various approaches to synthesize ranking functions are proposed, such as linear [7,46], lexicographic linear [9,28], multiphase [8], and piecewise [49]. However, these ranking function-based methods mostly are incomplete. The denial of proving termination does not imply non-termination, and the program still may or may not terminate.

Correspondingly, proving non-termination is also a challenging task. The general approach is based on recurrence set [30], which is a set of non-terminal program states in which the program can stay indefinitely [12]. Chen et al. [13] further proposed a concept, i.e., closed recurrence set, simplified the non-termination proving problem, and solved it with safety provers. Cook et al. [18] introduced live abstractions to soundly disprove termination. Frohn et al. [23] designed a loop acceleration technique, which can generate a non-termination criterion to promote the proving process. Le et al. [40] utilized dynamic invariant inference to learn recurrent sets.

In recent years, some approaches combined with machine learning and deep learning become popular. Unlike previous methods that need to predefine a linear or polynomial ranking function template first, Tan et al. [48] proposed to learn ranking functions via deep neural network (DNN). A similar thought was implemented by Giacobbe et al. [26], and they developed their prototype tools for proving termination. Xu et al. [54] presented a series of data-driven techniques to infer the upper bound of loop iterations. Alon et al. [5] took the priority of GNN in representation learning, and designed a classifier for program termination detection. Based on their contributions, more potential of deep learning methods can be explored.

Our proposed method, as a deep learning-based termination analysis tool, can be used in scenarios in the areas of software development, use, and maintenance. The main application is to predict defective programs that can not be terminated. It is particularly crucial for software testing, as the bugs caused by non-terminating programs will lead to undesirable outputs, infinite executions, and even crashes. Our proposed method can enhance the quality of software testing and reduce the risk from the non-termination vulnerability.

2.2. GNN and graph contrastive learning

Inspired by the outstanding performance brought by DNN, GNN is presented to deal with graph-structured data and becomes a prosperous research domain [29]. Kipf et al. [39] extended the spectral convolution operation and proposed graph convolutional network (GCN). Velickovic et al. [50] designed graph attention network (GAT), introducing masked self-attentional layers to conduct the model to focus on the most relevant parts. Xu et al. [53] emphasized capturing the difference between graph structures, and developed graph isomorphism network (GIN). There are also diverse variants that have their own advantages for different graph types, like EdgeConv [52] for point clouds, GraphSAGE [31] for large graphs. These models can be used as encoders to transform graph-structured data into vectors.

Based on graph neural networks, researchers proposed graph contrastive learning to improve the performance of graph tasks. Graph contrastive learning is an approach that aims to utilize those label-invariant properties to gather samples with the same label and separate samples with different labels. The properties are usually learned from the original data and the augmented auxiliary data. You et al. [62] proposed four data augmentation methods from the perspectives of node, edge, attribute, and subgraph. Zhang et al. [65] presented three feature augmentation methods that directly perturb hidden feature matrices. Zhu et al. [67] demonstrated that augmentation methods should adjust by input graphs. Different from the above self-supervised contrastive approach, Khosla et al. [37] and Zhang et al. [64] extended it to a supervised setting and applied it in image classification.

As for the field of program analysis, to represent programs as graphs, there are various representation methods [2,51,63]. These methods can be applied to downstream tasks like type inference [1], bug detection [3], code summarization [21,59], etc. The significant improvement obtained by these GNN-based methods shows the superiority of graphs in analyzing programs.

3. Preliminary

To formally illustrate our graph-enhanced contrastive approach, we first present some background knowledge in this section. It contains the notations of graphs and the calculation of some GNN layers that are used in TerGEC.

3.1. Notations

In this paper, we denote a graph as $G = (\mathcal{V}, \mathcal{E}, X)$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. $N = |\mathcal{V}|$ is the number of vertices. The graph structural and attributive features are represented as an adjacency matrix $A \in \{0,1\}^{N \times N}$ and a feature matrix $X \in \mathbb{R}^{N \times F}$, where $A_{i,j} = 1$ iff $e_{i,j} \in \mathcal{E}$. F is the dimension of attribute vector, and $x_i \in \mathbb{R}^F$ is the feature of v_i.

3.2. GNN layers

Graph Convolutional Network (GCN), which is one of the most representative methods of GNN. Set H^l is the feature matrix in the l^{th} layer, that is $H^0 = X$ for original graphs and $H^0 = \tilde{X}$ for augmented graphs, the layer-wise propagation rule is:

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

Here, $\hat{A} = A + I_N$ is the adjacency matrix A with the addition of the identity matrix I_N. This is actually adding self-connections on graphs to consider the influence from nodes themselves. $\tilde{D}$ is the degree matrix of $\hat{A}$, which is a diagonal matrix and $\tilde{D}_{ii} = \sum_j \hat{A}_{ij}$. Adding the degree matrix $\tilde{D}$ can normalize $\hat{A}$ that can avoid neglecting the information from those nodes with few degrees. W^l is a layer-specific trainable weight matrix. $\sigma(\cdot)$ is an activation function, such as ELU [17], ReLU [45], PReLU [32].

Graph Attention Network (GAT), which implements the self-attention mechanism in the graph domain, emphasizing concentrating on the most relevant parts. In calculation, the preference is realized by attention coefficients, which is defined as:

$$e_{ij} = a\left(\left[W^l h_i^l; W^l h_j^l\right]\right) \tag{2}$$

Where e_{ij} is the attention coefficient between node i and node j, indicating the importance of node j's feature to node i. h_i^l is the input feature vector of node i in the l^{th} layer, and is also the i^{th} row of feature matrix H^l. h_j^l has a similar meaning. W^l is a trainable weight matrix for l^{th} layer. $[\cdot;\cdot]$ denotes the concatenation operator. The function $a(\cdot)$ maps the concatenation matrix to a real number. In the implementation, $a(\cdot)$ is a single-layer feedforward neural network, which is parameterized by a weight vector $\vec{\mathbf{a}}$, and followed by an activation function LeakyReLU [44] to provide nonlinearity. Thus attention coefficients e_{ij} can be represented as:

$$e_{ij} = \text{LeakyReLU}\left(\vec{\mathbf{a}}^T\left(\left[W^l h_i^l; W^l h_j^l\right]\right)\right) \tag{3}$$

Following the graph topology, only adjacency nodes are computed in the attention coefficient. It is necessary to normalize the coefficients so that makes them easily comparable. After normalization, the coefficients are expressed as:

$$\alpha_{ij} = \text{softmax}\left(e_{ij}\right) = \frac{\exp\left(e_{ij}\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(e_{ik}\right)} \tag{4}$$

Where $\mathcal{N}_i$ is the neighborhood of node i.

To calculate the output feature vector of node i in the l^{th} layer, namely $\hat{h}_i^l$, multi-head attention is applied that can concatenate attention coefficients from diverse representation spaces. Formally, with K independent attention mechanisms, $\hat{h}_i^l$ is defined as:

$$\hat{h}_i^l = \text{Concat}_{k=1}^{K}\left(\sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k W^{lk} h_j^l\right)\right) \tag{5}$$

Here, $\sigma(\cdot)$ is an activation function. W^{lk} is the corresponding weight matrix W^l in the k^{th} attention mechanism.

Graph Isomorphism Network (GIN), which achieves maximum discriminative power for graph topology. It redesigns the aggregation process and combination process, and updates node representations by:

$$h_i^l = \text{MLP}^l\left(\left(1 + \epsilon^l\right) \cdot h_i^{l-1} + \sum_{j \in \mathcal{N}_i} h_j^{l-1}\right) \tag{6}$$

Here, h_i^l denotes the feature of node i in the l^{th} layer. MLP^l is one multi-layer perceptron that is used in the l^{th} layer, to transform features to the output dimension. ϵ^l can be a trainable parameter or a fixed scalar, which can introduce a tiny difference in different GIN layers.

4. Research methods

Fig. 1 demonstrates the overall framework of TerGEC, which contains four components, i.e., graph augmentation, graph encoder, intra-class learning, and inter-class learning. Programs are first transformed into graphs, and then graphs are fed into the graph augmentation module. Graph augmentation is used to generate different views for input graphs, which is the foundation for the following inter-class learning and improves the robustness of the overall framework. Graph Encoder maps input graphs and augmented graphs to a feature space. Subsequently, a weighted contrastive loss and a focal loss are introduced in TerGEC to guide intra-class learning and inter-class learning, respectively. The above pipeline consequently forms our enhanced contrastive strategy.

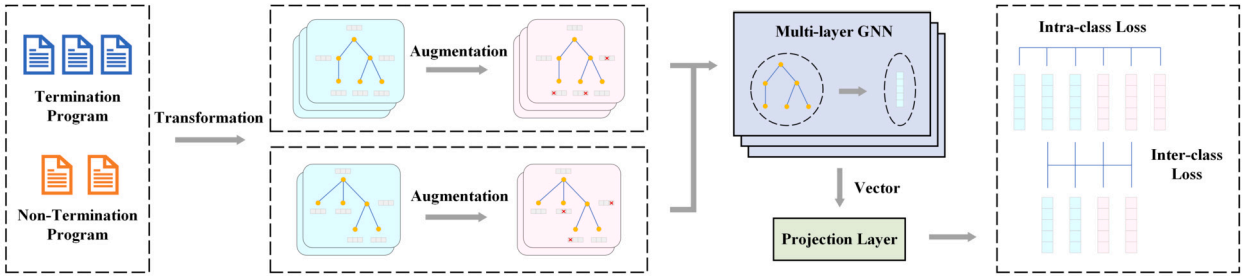

Fig. 1. The overall framework of TerGEC.

4.1. Graph augmentation

As for input programs, we first transform them into graph representations. The conversion process is a part of data preprocessing and is elaborated in section 5. Subsequently, samples are fed into the graph augmentation module. The objective of the augmentation module is to improve the sufficiency and diversity of samples [58], and is also the prerequisite for the following inter-class learning. In addition, augmentation has been shown to be effective in preventing deep networks from over-fitting and improving robustness [25,56,66]. To create rational data, graph augmentation usually follows certain methods to add perturbation on nodes, edges, or attributes.

In TerGEC, for input AST graphs, the structural information of programs is represented via the graph topology. To keep the topology and make ASTs meaningful, we propose two methods operating on attributive features (i.e., node representations) to augment graphs, i.e., feature masking and feature projection. It is flexible to use either of them in our framework.

Feature masking, which uses a randomly generated masking template to substitute a fraction of dimensions with zeros in attributive features. The masking template $M \in \{\mathbf{0}, \mathbf{1}\}^F$, where $\mathbf{0} \in \{0\}^{N\times 1}$ and $\mathbf{1} \in \{1\}^{N\times 1}$. The distribution for each column vector in M is a Bernoulli distribution, namely the column vector is $\mathbf{0}$ with probability p_m, and is $\mathbf{1}$ with probability $1 - p_m$. Therefore, the augmented feature matrix $\tilde{X}$ is defined as:

$$\tilde{X} = M \circ X \in \mathbb{R}^{N\times F} \tag{7}$$

Where $\circ$ denotes the Hadamard product.

Feature projection, which aims to project the input attributive features to an approximate feature matrix. It has been proved that can facilitate learning mixtures of Gaussians [19]. Let $P \in \mathbb{R}^{N\times N}$ be the projection matrix, whose element p_{ij} is randomly sampled and has a standard normal distribution, namely $p_{ij} \sim \mathcal{N}(0, 1)$. By multiplying P and X, the augmentation of feature matrix is computed by:

$$\tilde{X} = PX \in \mathbb{R}^{N\times F} \tag{8}$$

After graph augmentation, there are batches of augmented graphs that have the same topology as input graphs, and only attributive features are transformed. They all are sent to the same graph encoder for the following processes.

4.2. Graph encoder

Graph encoder targets extracting graph-level representations, consisting of an embedding layer and a multi-layer GNN. The embedding layer converts features to an embedding space, where the dimension of features is F_e, and $F_e \ll F$. By embedding, sparse vectors are converted to dense vectors, which preserve the semantic information and reduce the dimensionality. It also decreases the number of parameters that can accelerate the calculation of TerGEC.

The multi-layer GNN is the backbone of encoding. Each GNN layer updates nodes' representations by aggregating the information from neighbors of nodes. It can be represented formally as:

$$a_i^k = \text{AGGREGATE}^k\left(\left\{h_j^{k-1} : j \in \mathcal{N}_i\right\}\right) \tag{9}$$

$$h_i^k = \text{COMBINE}^k\left(h_i^{k-1}, a_i^k\right) \tag{10}$$

Here, a_i^k denotes the aggregation representation from the neighbors of node i in the k^{th} layer. h_i^k is the updated representation of node i in the k^{th} layer.

The number of layers indicates how many hops the aggregation can be executed. With the hop increasing, the central node can interact information with nodes from a wider scope. Fig. 2 shows the interaction scope of the root node with 1 hop and 2 hops respectively. Referring to the models utilized broadly in the graph classification tasks, we propose three models that can be used in the multi-layer GNN, i.e., GCN [39], GAT [50], GIN [53]. They use differentiated methods to implement the $\text{AGGREGATE}^k(\cdot)$ and $\text{COMBINE}^k(\cdot)$ functions. The details of corresponding calculations are demonstrated in section 3.2.

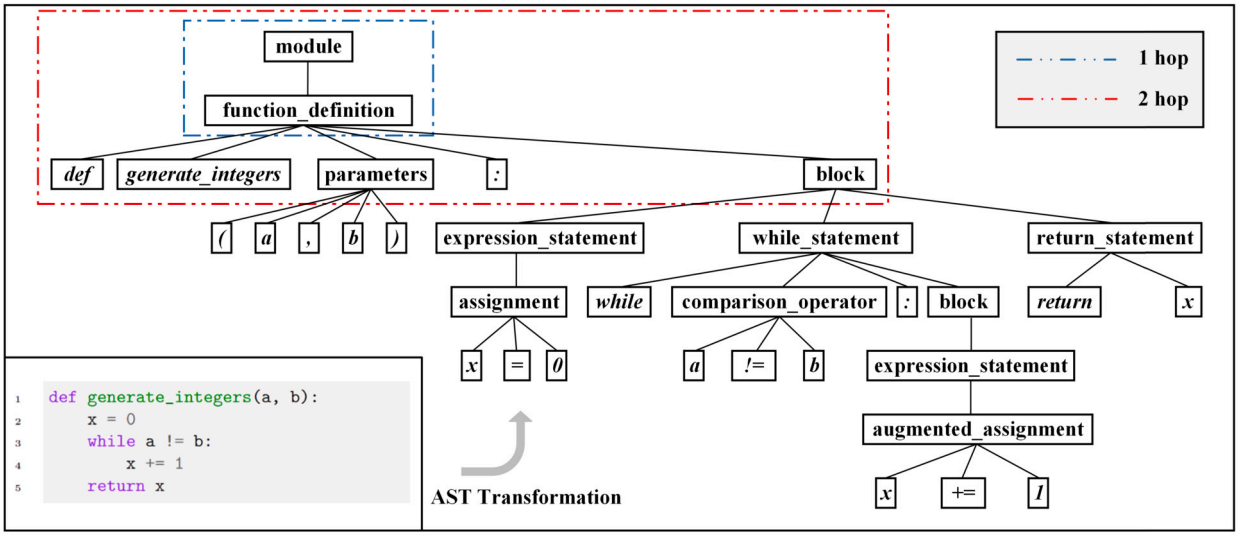

Fig. 2. An example of AST for Python.

After multi-layer GNN, nodes' feature vectors are updated. As for ASTs, it means the syntactical and lexical information has been captured by the graph encoder. Moreover, to predict whether a program is terminated or not, we adopt a READOUT$(\cdot)$ function to aggregate nodes' feature vectors from each graph in the final GNN layer. It can be denoted as:

$$h_G = \text{READOUT}\left(\left\{h_i^k | i \in G\right\}\right) \tag{11}$$

Where h_G is the aggregated feature vector. Some usual methods for the READOUT$(\cdot)$ function are sum, mean, and max aggregators. In this paper, we use the mean aggregators to avoid influences from the number of nodes.

4.3. Intra-class learning

In the intra-class learning module, we concentrate on original graphs G and their representations U that are extracted from the graph encoder. Therefore, we optimize TerGEC directly based on their respective labels. Similarly, an MLP as the projection layer $\theta^*(\cdot)$ is used for U firstly. To deal with the imbalance problem, instead of traditional cross-entropy loss, we use focal loss [42] to optimize TerGEC during intra-class learning. Focal loss can be formally defined below:

$$L_{intra} = FL\left(p_t\right) = -\alpha_t\left(1-p_t\right)^{\gamma}\log\left(p_t\right) \tag{12}$$

Here, p_t is the estimated probability of each class. If we use $y \in \{0,1\}$ indicates labels, $p_t = p$ if $y = 1$, and $p_t = 1 - p$ otherwise. α_t has an analogous definition, which is a weighting factor. $\left(1-p_t\right)^{\gamma}$ is a modulating factor, where γ is a focusing parameter. It is observed that focal loss is an extension of cross-entropy loss, which emphasizes adding weights for samples that are in the minority class and reducing the loss contribution from easy samples.

4.4. Inter-class learning

Apart from learning from intra-classes, acquiring the discrepancies between programs of terminating and non-terminating from inter-classes is also crucial and conducive for termination detection. From the perspective of feature space, samples of different classes should be separated as much as possible, while samples of the same class should be as similar as possible, which can be seen as inter-class information. In TerGEC, following this motivation, we propose the inter-class learning module.

For a batch of graphs G and their augmentations $\tilde{G}$, their graph representations $U \in \mathbb{R}^{S\times D}$ and $\tilde{U} \in \mathbb{R}^{S\times D}$ are obtained after graph encoder, where S is the batch size and D is the dimension of graph feature vectors. Before calculating the difference, a nonlinear projection layer $\theta(\cdot)$ is added to transform graph representations to another hidden space that is used to compute the inter-class loss. It has been proved that a nonlinear projection can help improve the representation quality [15]. In the implementation, the projection layer is an MLP with a nonlinear activation function.

After projection, U and $\tilde{U}$ are transformed to U' and $\tilde{U}'$. We concatenate the two graph feature matrices and get $\hat{U} = \left[U'; \tilde{U}'\right]$, computing inter-class loss based on $\hat{U}$. We set labels of augmented graphs as same as the corresponding original graphs. Thus in comparison to U', $\hat{U}$ contains more samples, whose quantity is $2S$, and introduces augmented positive samples that maintain the graph topology feature. In $\hat{U}$, each sample has its label, namely termination or non-termination, and samples can be separated according to their labels. What we target at is to enforce the classifier to maximize the difference between samples of different classes. For each pair of samples in $\hat{U}$, i.e., $\hat{u}_i$ and $\hat{u}_j$, we utilize a normalized temperature-scaled cross-entropy loss function to

capture their similarity, which is denoted as L_{inter}^{ij}. The difference is implicitly expressed in the other pairs of samples, i.e., $\hat{u}_i$ and $\hat{u}_k$, where $i \neq k$. Only pairs of samples that have the same label are accumulated for the inter-class loss L_{inter}, which can be seen as a weighted contrastive loss. It can be formally represented as:

$$L_{inter}^{ij} = -\log \frac{\exp\left(\varphi\left(\hat{u}_i, \hat{u}_j\right)/\tau\right)}{\sum_{k=1}^{N} \mathbf{1}_{i\neq k} \exp\left(\varphi\left(\hat{u}_i, \hat{u}_k\right)/\tau\right)} \tag{13}$$

$$L_{inter} = \sum_{i=1}^{N} \frac{\alpha}{N_{y_i} - 1} \sum_{j=1}^{N} \mathbf{1}_{i\neq j} \mathbf{1}_{y_i = y_j} L_{inter}^{ij} \tag{14}$$

Where $\varphi\left(\hat{u}_i, \hat{u}_j\right) = \hat{u}_i \cdot \hat{u}_j / \|\hat{u}_i\| \|\hat{u}_j\|$ is the cosine similarity function. τ is a temperature parameter. y_i is the label of sample $\hat{u}_i$, and N_{y_i} is the number of samples whose labels are as same as y_i in the batch. $\mathbf{1}_{i\neq j} \in \{0, 1\}$ and $\mathbf{1}_{y_i = y_j} \in \{0, 1\}$ are two indication functions. They equal to 1 iff $i \neq j$ and $y_i = y_j$ respectively. α is a weighted coefficient we set to alleviate the imbalance between termination samples and non-termination samples. According to their proportion in the batch, a higher α is assigned to samples that are in the minority class, namely non-termination. With the decreasing of L_{inter}, samples of the same class are gathered and samples of different classes are separated, which brings a brand-new optimization objective for high-quality termination analysis.

Finally, to combine the intra-class and inter-class semantic information together, we use a hyperparameter λ to balance their weights. Therefore, the overall loss is defined as:

$$L = \lambda L_{inter} + (1 - \lambda) L_{intra} \tag{15}$$

Different from the training stage where the loss is produced by a weighted sum of intra-class and inter-class modules, during the inference stage, given a piece of code snippet, the predictive results of termination or not are produced by the intra-class learning module of TerGEC only.

In summary, the graph-enhanced contrastive approach is manifested in Algorithm 1.

Algorithm 1 Graph Enhanced Contrastive Approach.

Input: Batches of graphs G, which are transformed from programs, with their corresponding labels y that indicate termination or non-termination.
Output: Prediction results y'.

for epoch = 1, 2, … **do**
  Generate the augmented graphs $\tilde{G} = (\mathcal{V}, \mathcal{E}, \tilde{X})$.
  Obtain graph representations U and $\tilde{U}$ of input graphs G and augmented graphs $\tilde{G}$ after the graph encoder.
  Project U that is obtained in step 3 by $\theta^*(\cdot)$.
  Calculate the intra-class loss L_{intra}.
  Project U and $\tilde{U}$ by $\theta(\cdot)$ to obtain U' and $\tilde{U}'$.
  $\hat{U} = \left[U'; \tilde{U}'\right]$.
  Calculate the inter-class loss L_{inter}.
  Combine the two optimization objectives and calculate L.
  Update parameters by gradient descent.
end for
Predict labels using the trained model.
return y'

5. Data preparation

Existing termination detection datasets face some common problems. On the one hand, the numbers of data samples are relatively limited, usually several hundred or so, and some labels are missing. On the other hand, the proportion of termination samples and non-termination samples in these datasets is approximately balanced, which violates the practical situation that only a minority of programs are non-terminal. To address these limitations, we construct four custom datasets for termination detection.

5.1. Preprocessing

The raw datasets are derived from Large language models (LLMs)-generated programs. To be specific, it is based on two widely utilized code generation datasets, i.e., HumanEval [14] and MBPP [6]. The two datasets are widely acknowledged in the field of code generation, and they are equipped with test cases, which can be used to judge whether each program can terminate. To construct our datasets, we adopt the code generation results of both GPT-Neo 125M and GPT-Neo 1.3B [35], producing four distinct raw datasets, namely 125M_HumanEval, 125M_MBPP, 1_3B_HumanEval, and 1_3B_MBPP, respectively. GPT-Neo is an open-sourced model and is pre-trained on Pile dataset [24]. It has two versions whose parameters are 125M and 1.3B. We chose GPT-Neo because there are existing results on HumanEval and MBPP, which has been published by [35]. Thus, it is convenient to construct the raw datasets. In addition, two code-generation models are involved which can guarantee the diversity of our four constructed datasets.

The raw constructed datasets include plentiful programs. HumanEval is a set of 164 hand-written programming problems, and MBPP contains 500 test problems. For each problem, there are about 100 generated programs. The raw number of each dataset is shown in Table 1. However, as are automatically generated, many of them may have grammatical errors. Therefore, we filter them

Table 1
Statistics for datasets.

Datasets	Raw Programs	Graphs	Features	Nodes	Edges	Termination (Negative)	Non-Termination (Positive)
125M_HumanEval	16400	3338	850	~70.1	~69.1	3250	88
125M_MBPP	47964	1336	921	~65.1	~64.1	1308	28
1_3B_HumanEval	15744	4562	886	~62.5	~61.5	4485	77
1_3B_MBPP	47978	10995	1695	~50.6	~49.6	10856	139
TPDB	206	206	220	~126.3	~125.3	158	48

and subsequently label the raw datasets with termination or not. Specifically, we use the test cases given by HumanEval and MBPP to evaluate each generated code. Programs without loops are first filtered out because a loop is a prerequisite for non-termination. When executing a program with equipped test cases, if it exceeds a predefined time limit, which is set as 10 s in this work, it means the program is a "non-termination". For the remaining programs, if they can pass all test cases, they will be annotated as "passed". Or if they raise some error, they will be labeled as "failed". There are many reasons that may cause a program failure. If the raised error is a logic error, which means the program can not pass all the test cases, but the program can still execute and terminate. To supplement our datasets, among all programs with logic errors, we pick up those containing loop iterations, and combine them with "passed" programs to constitute "termination". Subsequently, programs labeled with termination and non-termination will be transformed into graph representations.

5.2. Data transformation

To represent programs both syntactically and lexically, we transform them into AST graphs. AST is a homogeneous and undirected graph presenting program syntax via its hierarchy and non-leaf nodes while conserving the program lexical via leaf nodes, which is widely applied in compilers for static analysis. In this paper, those programs that have been preprocessed are converted to corresponding ASTs by an open-sourced parser generator tool, tree-sitter [11]. Fig. 2 illustrates an example in 125M_HumanEval dataset. The values of leaf nodes in AST are shown in italics, such as "def" and "generate_integers". Non-leaf nodes express different types in syntax, such as "module" and "parameters", and they are organized hierarchically. Therefore, the content and structure of the original program can be fully represented in AST.

As for the initialization of attributive features, we maintain a dictionary for each dataset. The dictionary contains all types of non-leaf nodes and all values of leaf nodes that appear in the same dataset, and it maps each node to a one-hot vector. It is noted that we do not embed original node features in data preparation, but combine the embedding process with our methods. Table 1 manifests the statistics of our four custom datasets, including the number of graphs, the dimension of features, the average number of nodes and edges, as well as the number of different classes. As shown in the table, the positive samples merely take a small proportion.

6. Evaluation

To investigate the performance of TerGEC, we conduct plentiful experiments, including comparing with multiple state-of-the-art baselines and analyzing the improvement contributed by the components of TerGEC. Specifically, our evaluations follow four research questions below:

RQ1. **Quantitative Evaluation**. How does TerGec perform compared with the state-of-the-art baselines?

RQ2. **Role of Graph Augmentation**. Can our proposed augmentation strategies improve the performance in termination analysis?

RQ3. **Backbone Analysis**. Embedded with various backbone models, such as GCN, GAT, and GIN, how do they influence the performance of TerGEC?

RQ4. **Effectiveness of Balance Factors**. To what extent that the balance factors in TerGEC can improve performance in an imbalanced setting?

6.1. Experiment setting

Baseline models. There are totally eight state-of-the-art baselines that TerGEC compares with. Besides three fundamental methods, namely GCN [39], GAT [50], and GIN [53], we also compare TerGEC with some recent representative methods concluding GGNN [41], GNN-FiLM [10], and graph transformer [47]. GGNN [41] extends the usage of gated recurrent units (GRU) to graph-structured problems so that it can update node and graph representation vectors. GNN-FiLM [10] introduces the graph hypernetworks and a feature-wise linear modulation to modify the message passing process that numerous GNNs used. Graph transformer [47] is inspired by the power of transformer in NLP, reforming it into graph learning tasks. In addition, two recurrent networks are also used to compare, namely Long Short-Term Memory (LSTM) [33] and Gated Recurrent Unit (GRU) [16]. It is noticed that there is also a branch of methods [26,54] that require satisfiability modulo theories (SMT) solving to execute programs and check whether programs can terminate, but they have noteworthy drawbacks like needing complex configuration for complete sandbox environment and taking too much time in execution, leading to their unrealistic application in practice. Therefore, we exclude these methods from baselines and choose methods that do not need the execution process to compare.

Table 2
Experimental results for the baselines and TerGEC.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)
LSTM	54.64	68.27	62.30	64.77	50.00	50.06	50.01	49.95	53.17	56.71
GRU	54.49	62.52	56.31	71.50	51.09	75.26	50.32	67.28	57.14	58.33
GCN	60.08	81.56	71.49	85.17	58.51	88.59	58.84	90.29	61.02	84.18
GAT	64.51	83.54	71.54	87.22	59.33	91.23	59.61	92.08	63.50	88.89
GIN	63.18	86.26	66.06	85.10	60.90	90.52	59.31	91.87	65.18	87.21
GGNN	64.14	86.15	68.03	86.69	61.18	**91.25**	55.54	90.36	62.57	83.16
GNN-FiLM	64.92	85.21	65.90	87.22	58.67	89.56	54.71	88.74	63.38	82.49
Graph Transformer	65.46	**87.91**	67.57	86.24	61.20	89.33	58.33	88.96	63.20	83.16
TerGEC	**70.05**	84.11	**74.10**	**92.32**	**61.38**	90.20	**60.61**	**93.33**	**68.35**	**91.58**

Metrics. To evaluate the performance of TerGEC, we use mean Average Precision (mAP) and the Area Under the receiver operating characteristic Curve (AUC) as metrics. Average Precision (AP) summarizes the Precision-Recall curve to one scalar value, which can be seen as the weighted mean of precisions when setting different thresholds. Mean average precision is the average of APs for termination and non-termination instances. Receiver Operating characteristic Curve (ROC) shows the true positive rate and the false positive rate at each classification threshold, and AUC uses a value between 0 to 1 to manifest ROC. A higher AUC indicates better performance.

Datasets. We conduct extensive experiments on programs with Python and C language, which can evaluate the generalization ability of TerGEC. As for Python programs, we use four custom datasets, and their construction processes are introduced in section 5. As for C programs, we use an existing dataset, the Termination Problem DataBase (TPDB), which is from the TermComp termination competition [27]. Although it has been published by [27] as claimed in their paper, we cannot find the complete version. Therefore, we tried our best to collect a subset, which contains 206 C programs, with 158 termination programs and 48 non-termination programs. Its detailed statistics are shown in Table 1. All datasets are split into the training set, validation set, and testing set by the ratio of 7/1/2 for experiments.

Setup. The experiments were conducted on a Ubuntu GPU server with four RTX 3090 24 GB GPUs. TerGec was implemented by the deep learning framework PyTorch[2] and a GNN package PyG [22]. Specifically, we set 4 layers for the GNN encoder and use Adam optimizer [38] whose learning rate is adjusted in $\{5e-4, 1e-4, 5e-5\}$ according to the datasets. Moreover, batch size is adjusted in $\{64, 128\}$ and the dimension of hidden vectors comes within $\{128, 256\}$. A maximum training epoch is set to 200. As for the hyperparameters in the intra-class and inter-class learning module, we set $\alpha_t = 0.95$ and $\gamma = 2$ for the focal loss, which is a default setting in many previous studies [5,42] and has been proved the effectiveness. We adjust the value of λ in the range of (0, 0.5] and choose the value that performs best, which is $\lambda = 0.01$. An elaborated illustration of the role of λ has been discussed in section 6.5. In addition, we adjust the specific method used in the graph augmentation and the backbone. We consistently selected the best combination in each experimental dataset and recorded their performance. A detailed analysis of the two components is illustrated in section 6.3 and 6.4.

6.2. RQ1: quantitative evaluation

Comparing TerGEC with state-of-the-art baselines can directly indicate the superiority that TerGEC has. To achieve the best performance of TerGEC, we adjust some components like the graph augmentation method and the backbone used, reporting the best results. The effects of choosing different graph augmentation strategies and backbones will be further discussed in section 6.3 and 6.4. For the eight baselines, which follow the same setting of TerGEC, we implemented them by PyTorch and PyG as well. Table 2 shows the detailed results for the baselines and TerGEC.

As shown in the table, TerGEC achieves the best performance on almost all datasets and all evaluation metrics. It can be observed that the performance of TerGEC surpasses LSTM and GRU a lot, which indicates that the graph representation of programs benefits the termination analysis. Specifically, for the mAP metric, TerGEC outperforms baselines in all five datasets. In terms of 125M_HumanEval, there is an increment on mAP about 4.59% - 15.56%. In addition, the increments are 2.56% - 17.79% in terms of 125M_MBPP, 0.18% - 11.38% in terms of 1_3B_HumanEval, 1.00% - 10.60% in terms of 1_3B_MBPP, and 3.17% - 15.18% in terms of TPDB. The improvements indicate that TerGEC can accurately predict non-terminating programs among a mass of termination samples with a lower error rate. As for the AUC metric, TerGEC achieves the best on datasets 125M_MBPP, 1_3B_MBPP, and TPDB, where the improvements are 5.10% - 27.55%, 1.25% - 42.38%, and 2.69% - 34.87% respectively. Although on the other two datasets, 125M_HumanEval and 1_3B_HumanEval, TerGEC does not perform best. We investigate that because TerGEC is equipped with classes-imbalance solutions, more non-terminating programs were found, but this also resulted in some misclassified termination samples, which leads to the decline of AUC metric in the two datasets. Furthermore, besides Python program datasets, TerGEC outperforms all baselines in C program dataset TPDB, which manifests TerGEC has the generalization ability that is regardless of the programming language.

[2] https://pytorch.org/.

Table 3
Experimental results about graph augmentation.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)
With FM	**70.05**	**84.11**	74.03	91.10	**61.38**	**90.20**	**60.61**	93.33	**68.35**	**91.58**
With FP	67.19	83.32	**74.10**	**92.32**	57.90	89.67	60.42	**94.23**	65.15	83.50
w/o Augmentation	62.80	83.08	73.90	90.87	57.80	88.23	57.09	92.95	63.33	83.50

Table 4
Experimental results with different backbones.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)
TerGEC-GCN	66.57	83.60	71.13	90.04	59.03	88.34	58.29	89.66	62.03	85.52
TerGEC-GAT	**70.05**	84.11	72.40	89.96	59.63	87.70	58.21	90.48	68.35	**91.58**
TerGEC-GIN	65.04	**85.22**	**74.03**	**91.10**	**61.38**	**90.20**	**60.61**	**93.33**	**68.44**	87.88

Answer to RQ1: When comparing with the state-of-the-art baselines, our proposed TerGEC can achieve better/comparable performance on both Python program datasets and C program datasets, which indicates that TerGEC can improve the prediction performance and has the generalization ability.

6.3. RQ2: role of graph augmentation

The graph augmentation module is a key component of TerGEC that can increase the diversity of samples and improve the robustness of TerGEC. To investigate the impact of graph augmentation, we compare the performance of TerGEC using different graph augmentation strategies with a special version of TerGEC that removes the graph augmentation module. The different graph augmentation strategies are feature masking (FM) and feature projection (FP), as elaborated in section 4.1. They can yield a batch of new graphs to supplement the original graphs. When removing the graph augmentation module, no more new graphs are produced, and only the original graphs are sent to the following graph encoder to calculate the intra-class and inter-class semantic information. Table 3 illustrates the experimental results of graph augmentation. For the selection of backbones, to achieve the best results as that in RQ1, we choose GAT for dataset 125M_HumanEval and TPDB, and GIN for dataset 125M_MBPP, 1_3B_HumanEval, and 1_3B_MBPP.

It can be seen from Table 3 that, regardless of FM or FP, using graph augmentation can bring a better performance on all five datasets. The maximum increment can be 7.25% for mAP and 8.08% for AUC. The good performance not only verifies that graph augmentation is beneficial to non-termination prediction but also proves that FM and FP are two effective strategies that have the potential to be used in program analysis tasks. As for the performance of TerGEC with FM and FP respectively, FP outperforms FM in dataset 125M_MBPP and 1_3B_MBPP, while FM has a higher score in the other datasets. We suspect that it is because the programs generated by the MBPP dataset are relatively short, so simple feature masking may lose more semantic information. The experimental results show that FM and FP have their own merits and demerits, and choosing a proper graph augmentation strategy for different datasets is vital. FM is simple in calculation but has the probability of losing the crucial dimension. FP is realized by a randomly sampled projection matrix, which has a higher possibility of introducing noise.

Answer to RQ2: Graph augmentation is conducive to the prediction of non-termination and can improve the performance in termination analysis tasks. Feature masking and feature projection are two effective graph augmentation strategies in practice.

6.4. RQ3: backbone analysis

Graph encoder in TerGEC aims to capture the semantic information from ASTs. Hence, choosing an appropriate GNN model as the backbone of the graph encoder module influences the representation capability of TerGEC. Because of the difference in initial design purposes, GNN models have their preferences in dealing with different tasks. Generally speaking, GAT performs well in node feature representation, and GIN does better in graph-level applications [34]. To choose a proper backbone for termination analysis, we investigate three popular GNN models, namely GCN [39], GAT [50], and GIN [53], and evaluate the performance of their corresponding TerGEC. Table 4 shows the performance of TerGEC with different backbones in Python and C program datasets.

From the results of Table 4, we can conclude that different backbones affect the performance of TerGEC. GAT achieves the best performance in dataset 125M_HumanEval and TPDB, and GIN wins in the other three. If we analyze the average node number of the five datasets, we can discover that dataset 125M_HumanEval and TPDB are higher than others. This phenomenon confirms the advantages of GAT in node representation, which is more suitable for graphs with more nodes. On the contrary, when the node number descends, the superiority of GIN in graph representation appears. Meanwhile, we should note that the difference among

Table 5
Experimental results about balance factors.

Model	125M_HumanEval		125M_MBPP		1_3B_HumanEval		1_3B_MBPP		TPDB	
	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)	mAP (%)	AUC (%)
TerGEC	**70.05**	**84.11**	**74.03**	**91.10**	**61.38**	**90.20**	**60.61**	**93.33**	**68.35**	**91.58**
$\lambda = 0.1$	60.20	83.04	68.31	87.53	57.63	90.42	61.06	93.25	64.72	82.49
$\lambda = 0.3$	58.97	84.04	68.04	88.06	58.04	90.97	60.29	93.45	63.95	80.47
w/o λ	58.33	82.12	67.38	89.89	57.51	88.99	56.21	93.18	63.97	79.80
w/o α	61.84	83.70	68.82	88.63	59.72	89.11	60.12	91.73	63.64	82.49

different backbones might be minute. Thus, it is necessary to consider carefully when selecting specific backbones. Moreover, as can be seen from Table 2 and Table 4, when using the same backbone, TerGEC always has better performances than the original baselines. But there are also some counter-examples. For instance, in the 1_3B_MBPP dataset, when using GCN or GAT as the backbone, the performance decreases slightly if using TerGEC. Due to the difference is minute here, we conjecture it can be attributed to the random initialization of the model. On the whole, our proposed TerGEC performs better in most datasets and backbones, which proves the effectiveness of TerGEC.

Answer to RQ3: The choice of different backbones can affect the performance of TerGEC. Specifically, GAT is suitable for datasets with more nodes like 125M_HumanEval and TPDB, and GIN does better in graph representation in the other three datasets.

6.5. RQ4: effectiveness of balance factors

To cope with the imbalance classes problem in termination analysis, TerGEC introduces the inter-class learning module and corresponding balance factors. There are two kinds of balance factors in TerGEC. One of them is the coefficient α in equation (14), which aims to give a higher weight to non-terminating samples that are in the minority, and another is the hyperparameter λ in equation (15), which is used to balance the proportion between inter-class and intra-class semantic information. To evaluate the effectiveness of the two balance factors, we conduct ablation experiments of TerGEC by removing α and λ respectively. That is to set $\alpha = 1$ in equation (14) and $\lambda = 0.5$ in equation (15). In addition, we also set $\lambda = 0.1$ and $\lambda = 0.3$ to observe the change of performance when λ is varied. Table 5 demonstrates the performances in detail. The other settings are also the same as that RQ1.

Table 5 distinctly presents a drop in prediction performance if removing no matter α or λ, proving that balance factors are indispensable in TerGEC. The decline could be 3.87% - 11.72% for the mAP metric and 1.21% - 11.78% for the AUC metric in the five datasets. To be specific, the drop caused by the loss of λ is more obvious than that of α. In fact, due to the more complex calculation requirements of the inter-class loss, its value is usually higher than the intra-class loss. When increasing the value of λ, the importance of intra-class loss is weakened, and the performance decreases. Balance factor λ plays an important role here in balancing the inter-class loss and the intra-class loss, bringing TerGEC to the optimum of classification. Similarly, balance factor α in equation (14) enlarges the effects from the non-terminating samples and conducts TerGEC to learn more semantic information from non-termination. Therefore, the two balance factors help TerGEC improve its performance in an imbalanced setting.

Answer to RQ4: Balance factors are effective when disposing of the class-imbalance problem in termination analysis. The coefficient α and hyperparameter λ boost the performance of TerGEC in the five Python and C program datasets.

7. Discussion

7.1. Limitation

Our work shows efficiency and effectiveness and enhances the performance of deep learning-based methods in termination analysis, but there are still limitations. To provide an estimation of a program's termination behavior, our method aims to learn the discrepancies in characteristics between non-terminating programs and terminating ones. However, it cannot provide formal proof of termination, which means the prediction may consist of some false positives, and the explainability is also weak. To try to alleviate the above limitations inherent in deep learning-based approaches, we designed inter-class and intra-class joint learning methodologies to improve the prediction performance of neural networks. As for explainability, we will further design a localizer to specify the defective lines and use visualization methods to provide necessary hints in future work. In addition, with the rapid development and extensive studies of pre-trained language models [43,60] and large language models [55,61], their application and adaptation in termination analysis are also worthy of trying. In the future, we will incorporate them into our framework for a more comprehensive investigation.

7.2. Threats to validity

There are some threats to the validity of our proposed TerGEC, and we list them in the following.

External Validity mainly concerns the generalizability of our findings to other datasets. TerGEC takes four custom datasets and one existing dataset into account, including Python and C languages. It is the first large-scale dataset for termination analysis. However, as TerGEC learns the semantic information from the training set for termination analysis, the dataset quality directly influences the performance. As for the four custom datasets, since we predefined a maximum execution time to identify non-termination programs, some programs that can terminate but may exceed the predefined execution time might be mislabeled. But we have set the execution time to be large enough to alleviate the threats to validity, which is set to 10 s. Thus, we believe this threat is minimal. In addition, following previous studies [5,26], we only consider programs with loops and analyze whether they can terminate, but exclude infinite recursion and corresponding non-termination issues. It is worth considering the infinite recursion problem in the future work.

Interval Validity mainly concerns the aspects that may impact the experimental results. One potential aspect is the hyperparameter setting. Finding the optimal hyperparameters is always a difficult task. In TerGEC, besides conventional hyperparameters, such as batch size and embedding dimension, a proper graph augmentation strategy, a suitable backbone, and an opportune setting of balance factors are also needed to adjust. Although we have analyzed these in section 6.3, 6.4, and 6.5, there are still some settings that are not considered, like if hyperparameter λ is set to other values, and we keep it for future work. In addition, threats are also related to the bias in the replication of baseline models. To replicate baselines on our datasets, we followed their designs and implemented them by PyTorch and PyG. We have double-checked and fully tested our replication code to reduce errors. Therefore, we believe that it is little threat to validity.

8. Conclusion

TerGEC, a graph-enhanced contrastive approach for program termination analysis, is proposed in this work. It can capture the intra-class and inter-class semantics of programs simultaneously and realize a more fine-grained analysis. In addition, graph augmentation is designed to improve robustness. Weighted contrastive loss and focal loss are also equipped in TerGEC to deal with the classes-imbalance problem. Four large-scale datasets were built to reflect the classes-imbalance problem in practice. At the time of writing, it is the first large-scale dataset study for termination analysis in software engineering. When compared with the state-of-the-art baselines, TerGEC generally achieves the best performances in both Python and C programs. Ablation experiments also demonstrate that the graph-enhanced contrastive approach that combines the intra-class and inter-class semantic information, along with our proposed classes-imbalance solutions, effectively improves the prediction performance of non-terminating programs. In the future, we aim to work on other graphs of programs, like flow charts, to provide predictions with more interpretability.

CRediT authorship contribution statement

Shuo Liu: Writing – review & editing, Writing – original draft, Validation, Software, Resources, Methodology, Data curation, Conceptualization. **Jacky Wai Keung:** Writing – review & editing, Supervision, Project administration. **Zhen Yang:** Writing – review & editing, Supervision, Project administration, Methodology, Conceptualization. **Yihan Liao:** Writing – review & editing. **Yishu Li:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong and the research funds of City University of Hong Kong (6000796, 9229109, 9229098, 9220103, 9229029).

References

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, Zheng Gao, Typilus: neural type hints, in: Proceedings of the 41st Acm Sigplan Conference on Programming Language Design and Implementation, 2020, pp. 91–105.
[2] Miltiadis Allamanis, Marc Brockschmidt, Mahmoud Khademi, 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, 2018, OpenReview.net.
[3] Miltiadis Allamanis, Henry Jackson-Flux, Marc Brockschmidt, Self-supervised bug detection and repair, Adv. Neural Inf. Process. Syst. 34 (2021) 27865–27876.
[4] Ariane Alves Almeida, Mauricio Ayala-Rincón, Formalizing the dependency pair criterion for innermost termination, Sci. Comput. Program. 195 (2020) 102474.
[5] Yoav Alon, Cristina David, Using graph neural networks for program termination, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 910–921.
[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al., Program synthesis with large language models, arXiv preprint, arXiv:2108.07732, 2021.
[7] Amir M. Ben-Amram, Samir Genaim, Ranking functions for linear-constraint loops, J. ACM 61 (4) (2014) 1–55.

[8] Amir M. Ben-Amram, Samir Genaim, On multiphase-linear ranking functions, in: Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II, Springer, 2017, pp. 601–620.
[9] Aaron R. Bradley, Zohar Manna, Henny B. Sipma, Linear ranking with reachability, in: Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17, Springer, 2005, pp. 491–504.
[10] Marc Brockschmidt, Gnn-film: graph neural networks with feature-wise linear modulation, in: International Conference on Machine Learning, PMLR, 2020, pp. 1144–1152.
[11] Max Brunsfeld, Patrick Thomson, Andrew Hlynskyi, et al., Tree-sitter https://github.com/tree-sitter/tree-sitter.
[12] Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotnỳ, Đorđe Žikelić, Proving non-termination by program reversal, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 1033–1048.
[13] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, Peter O'Hearn, Proving nontermination via safety, in: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20, Springer, 2014, pp. 156–171.
[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al., Evaluating large language models trained on code, arXiv preprint, arXiv:2107.03374, 2021.
[15] Ting Chen, Simon Kornblith, Mohammad Norouzi, Geoffrey Hinton, A simple framework for contrastive learning of visual representations, in: International Conference on Machine Learning, PMLR, 2020, pp. 1597–1607.
[16] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, arXiv preprint, arXiv:1412.3555, 2014.
[17] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter, Fast and accurate deep network learning by exponential linear units (elus), in: Yoshua Bengio, Yann LeCun (Eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.
[18] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, Peter O'Hearn, Disproving termination with overapproximation, in: 2014 Formal Methods in Computer-Aided Design (FMCAD), IEEE, 2014, pp. 67–74.
[19] Sanjoy Dasgupta, Experiments with random projection, arXiv preprint, arXiv:1301.3849, 2013.
[20] Shuo Feng, Jacky Keung, Xiao Yu, Yan Xiao, Miao Zhang, Investigation on the stability of smote-based oversampling techniques in software defect prediction, Inf. Softw. Technol. 139 (2021) 106662.
[21] Patrick Fernandes, Miltiadis Allamanis, Marc Brockschmidt, Structured neural summarization, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019, OpenReview.net.
[22] Matthias Fey, Jan E. Lenssen, Fast graph representation learning with PyTorch geometric, in: ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019.
[23] Florian Frohn, Jürgen Giesl, Proving non-termination via loop acceleration, in: Clark W. Barrett, Jin Yang (Eds.), 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019, IEEE, 2019, pp. 221–230.
[24] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al., The pile: an 800gb dataset of diverse text for language modeling, arXiv preprint, arXiv:2101.00027, 2020.
[25] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, Abhik Roychoudhury, Fuzz testing based data augmentation to improve robustness of deep neural networks, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1147–1158.
[26] Mirco Giacobbe, Daniel Kroening, Julian Parsert, Neural termination analysis, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 633–645.
[27] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, Akihisa Yamada, The Termination and Complexity Competition, Springer, 2019.
[28] Laure Gonnord, David Monniaux, Gabriel Radanne, Synthesis of ranking functions using extremal counterexamples, ACM SIGPLAN Not. 50 (6) (2015) 608–618.
[29] Marco Gori, Gabriele Monfardini, Franco Scarselli, A new model for learning in graph domains, in: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005, volume 2, IEEE, 2005, pp. 729–734.
[30] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, Ru-Gang Xu, Proving non-termination, in: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2008, pp. 147–158.
[31] Will Hamilton, Zhitao Ying, Jure Leskovec, Inductive representation learning on large graphs, Adv. Neural Inf. Process. Syst. 30 (2017).
[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Delving deep into rectifiers: surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1026–1034.
[33] Sepp Hochreiter, Jürgen Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[34] Zhenyu Hou, Xiao Liu, Yukuo Cen, Yuxiao Dong, Hongxia Yang, Chunjie Wang, Jie Tang, Graphmae: self-supervised masked graph autoencoders, in: Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2022, pp. 594–604.
[35] Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, Jianfeng Gao, Fault-aware neural code rankers, Adv. Neural Inf. Process. Syst. 35 (2022) 13419–13432.
[36] Jeroen Ketema, Alastair F. Donaldson, Termination analysis for gpu kernels, Sci. Comput. Program. 148 (2017) 107–122.
[37] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, Dilip Krishnan, Supervised contrastive learning, Adv. Neural Inf. Process. Syst. 33 (2020) 18661–18673.
[38] Diederik P. Kingma, Jimmy Ba, Adam: a method for stochastic optimization, in: Yoshua Bengio, Yann LeCun (Eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.
[39] Thomas N. Kipf, Max Welling, 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, 2017, OpenReview.net.
[40] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, ThanhVu Nguyen, Dynamite: dynamic termination and non-termination proofs, Proc. ACM Program. Lang. 4 (OOPSLA) (2020) 1–30.
[41] Yujia Li, Daniel Tarlow, Marc Brockschmidt, Richard S. Zemel, Gated graph sequence neural networks, in: Yoshua Bengio, Yann LeCun (Eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016.
[42] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár, Focal loss for dense object detection, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 2980–2988.
[43] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, Yihan Liao, et al., Delving into parameter-efficient fine-tuning in code change learning: an empirical study, arXiv preprint, arXiv:2402.06247, 2024.
[44] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng, et al., Rectifier nonlinearities improve neural network acoustic models, in: Proc. ICML, volume 30, Atlanta, Georgia, USA, 2013, p. 3.
[45] Vinod Nair, Geoffrey E. Hinton, Rectified linear units improve restricted Boltzmann machines, in: Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010, pp. 807–814.
[46] Andreas Podelski, Andrey Rybalchenko, A complete method for the synthesis of linear ranking functions, in: Verification, Model Checking, and Abstract Interpretation: 5th International Conference, VMCAI 2004 Venice, Italy, January 11-13, 2004 Proceedings 5, Springer, 2004, pp. 239–251.
[47] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, Yu Sun, Masked label prediction: unified message passing model for semi-supervised classification, in: Zhi-Hua Zhou (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19–27 August 2021, 2021, pp. 1548–1554, ijcai.org.

[48] Wang Tan, Yi Li, Synthesis of ranking functions via dnn, Neural Comput. Appl. 33 (2021) 9939–9959.
[49] Caterina Urban, Arie Gurfinkel, Temesghen Kahsai, Synthesizing ranking functions from bits and pieces, in: Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, the Netherlands, April 2-8, 2016, Proceedings 22, Springer, 2016, pp. 54–70.
[50] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio, Graph attention networks, in: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings, 2018, OpenReview.net.
[51] Yu Wang, Ke Wang, Fengjuan Gao, Linzhang Wang, Learning semantic program embeddings with graph interval neural network, Proc. ACM Program. Lang. 4 (OOPSLA) (2020) 1–27.
[52] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, Justin M. Solomon, Dynamic graph cnn for learning on point clouds, ACM Trans. Graph. 38 (5) (2019) 1–12.
[53] Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka, How powerful are graph neural networks?, in: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019, OpenReview.net.
[54] Rongchen Xu, Jianhui Chen, Fei He, Data-driven loop bound learning for termination analysis, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 499–510.
[55] Pengyu Xue, Linhao Wu, Zhongxing Yu, Zhi Jin, Zhen Yang, Xinyi Li, Zhenyu Yang, Yue Tan, et al., Automated commit message generation with large language models: an empirical study and beyond, arXiv preprint, arXiv:2404.14824, 2024.
[56] Yihao Xue, Kyle Whitecross, Baharan Mirzasoleiman, Investigating why contrastive learning benefits robustness against label noise, in: International Conference on Machine Learning, PMLR, 2022, pp. 24851–24871.
[57] Akihisa Yamada, Keiichirou Kusakari, Toshiki Sakabe, A unified ordering for termination proving, Sci. Comput. Program. 111 (2015) 110–134.
[58] Suorong Yang, Weikang Xiao, Mengcheng Zhang, Suhan Guo, Jian Zhao, Furao Shen, Image data augmentation for deep learning: a survey, arXiv preprint, arXiv:2204.08610, 2022.
[59] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, Miao Zhang, A multi-modal transformer-based code summarization approach for smart contracts, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), 2021, pp. 1–12.
[60] Zhen Yang, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, Shuo Liu, Yishu Li, Improving domain-specific neural code generation with few-shot meta-learning, Inf. Softw. Technol. 166 (2024) 107365.
[61] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, Ge Li, et al., Exploring and unleashing the power of large language models in automated code translation, arXiv preprint, arXiv:2404.14646, 2024.
[62] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, Yang Shen, Graph contrastive learning with augmentations, Adv. Neural Inf. Process. Syst. 33 (2020) 5812–5823.
[63] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, Zhi Jin, Learning to represent programs with heterogeneous graphs, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 378–389.
[64] Linfeng Zhang, Xin Chen, Junbo Zhang, Runpei Dong, Kaisheng Ma, Contrastive deep supervision, in: Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXVI, Springer, 2022, pp. 1–19.
[65] Yifei Zhang, Hao Zhu, Zixing Song, Piotr Koniusz, Irwin King, Costa: covariance-preserving feature augmentation for graph contrastive learning, in: Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2022, pp. 2524–2534.
[66] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, Liang Wang, Deep graph contrastive representation learning, arXiv preprint, arXiv:2006.04131, 2020.
[67] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, Liang Wang, Graph contrastive learning with adaptive augmentation, in: Proceedings of the Web Conference 2021, 2021, pp. 2069–2080.