SCIENCE CHINA
Information Sciences

• Supplementary File •

GTE: Learning Code AST Representation Efficiently and Effectively

Yihao Qin[1,2], Shangwen Wang[1,2*], Bo Lin[1,2], Kang Yang[1,2] & Xiaoguang Mao[1,2]

[1]*College of Computer Science and Technology, National University of Defense Technology, Changsha* 410073*, China;*
[2]*Key Laboratory of Software Engineering for Complex Systems, National University of Defense Technology, Changsha* 410073*, China.*

Appendix A Technical Details

Appendix A.1 Approach Overview

The overview of the GTE framework is shown in Figure A1. Firstly, GTE parses the code snippets into ASTs and constructs directed graphs based on the ASTs. Then, a batch of AST-based graphs is transformed into several blocks through graph batch and graph block algorithms to enable parallel tree encoding computation. Subsequently, the sequence-based models are integrated into the GTE framework to construct the GTE-based models. Finally, the encoded vector representations of the code snippets are fed into downstream models for specific tasks.

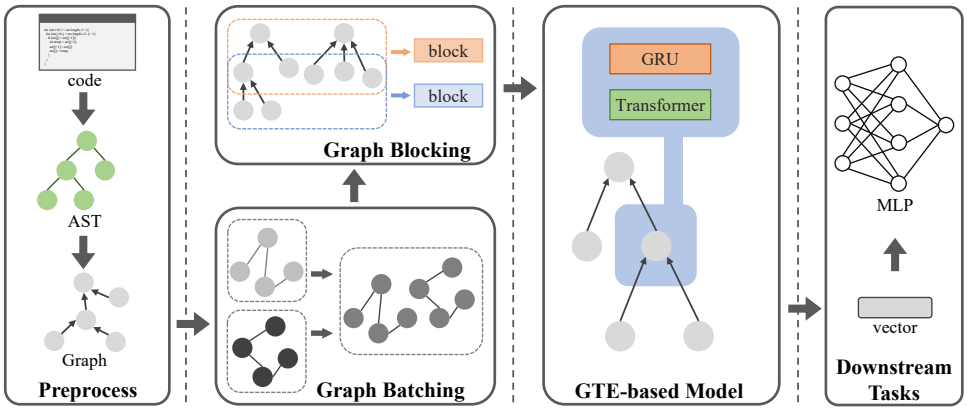

Figure A1 Overview of the GTE framework.

Appendix A.2 Graph Block Algorithm

Inspired by existing tree encoding models and the neighbor sampling method used for training on large graphs, we introduce the graph block algorithm for GTE. The basic intuition of the algorithm is to extract all nodes and edges in the subgraph that belong to adjacent layers of the corresponding AST as a block. Each block is a subgraph consisting of two sets of nodes: the source nodes (children) and the destination nodes (parents). By extracting multiple such blocks from the graph, the direction and scope of graph message propagation are limited, thereby achieving the intent to encode the AST more efficiently based on the message passing processes that may happen simultaneously on multiple ASTs. The specific steps of the algorithm are shown in Algorithm A1.

The algorithm takes a batch of code graphs generated by AST as input and outputs a list of graph blocks. First, the graph batch algorithm merges a batch of graphs G into a single graph g' (line 3). Since the node IDs of the original graphs will be changed during the graph batch process, the root node IDs of each subgraph's corresponding AST need to be calculated and restored using the size of each subgraph (lines 4-14). To accomplish this, we record the height of the corresponding AST tree (line 7) and determine the root node ID of the current graph g in the merged graph g' based on the size s of the AST tree (line 12) and the last recorded root node ID in IDX (line 11). Note that the size and height information of the corresponding AST tree for each graph g in G has been recorded during preprocessing.

After obtaining the root node IDs IDX and height information $HEIGHT$ of all subgraphs in the merged graph g', we extract blocks from g' through a loop (lines 16-30). The number of loop runs is calculated through the maximum height of all subgraphs in the batch (lines 16). In each iteration of the loop, we first build the subgraph $\_subgraph$ induced on the inbound edges of the given nodes based on the IDX of all subgraphs and g' (line 18). Based on the subgraph $\_subgraph$ and its directed edges, we can identify and generate a block that consists of all the source and target nodes and the edges between them, the block is then

* Corresponding author (email: wangshangwen13@nudt.edu.cn)

Algorithm A1 Constructing graph blocks.

```
1: Input: G: A batch of AST graphs.
2: Output: B: A list of blocks for the input graphs.
3: g' = graphBatching(G) ;      // Batching graphs to a single graph.
4: IDX = [ ] ;      // A list to record the root node index of each graph.
5: HEIGHT = [ ] ;      // A List to record the height of each graph.
6: for g in G do
7:     HEIGHT.append(g.height)
8:     if IDX is empty then
9:         IDX.append(0)
10:    else
11:        _lastId = getLastId(IDX)
12:        IDX.append(_lastId + g.size)
13:    end if
14: end for
15: B = [ ]
16: _loop = getMax(HEIGHT)
17: while _loop > 0 do
18:    _subgraph = getInSubgraph(g', IDX)
19:    _block = generateBlock(_subgraph, IDX)
20:    B.insertAhead(_block) ;      // Add the new block to B.
21:    IDX' = [ ] ;      // Temporary list for updating IDX.
22:    for id in IDX do
23:        PRE = getPredecessors(g', id)
24:        if PRE is not empty then
25:            IDX'.extend(PRE)
26:        end if
27:    end for
28:    IDX = IDX'
29:    _loop = _loop - 1
30: end while
31: return B
```

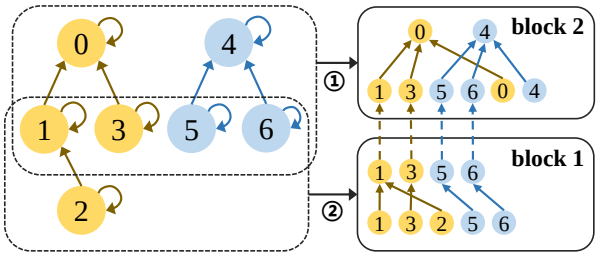

Figure A2 An example of graph block, the numbers in the hollow circles indicate the order in which blocks are generated.

appended to the list B (lines 19,20). After generating a block, new target node IDs IDX' can be obtained based on the current IDX for generating a new block in the next iteration. Specifically, for each node ID id in IDX (line 22), all predecessor nodes PRE in g' are added to IDX' (lines 23-26), and the target node IDs IDX' will become the source node IDs IDX for generate the next block (line 28). Finally, the algorithm returns the generated block list B.

Figure A2 illustrates an example of the graph block algorithm. The graph is comprised of two subgraphs, each marked in gold and blue respectively. The maximum height of the corresponding ASTs is 3, resulting in the generation of two blocks in two steps. Note that to encode the AST tree structure, the order of the input blocks for the GTE model is reversed from the order in which the blocks are generated.

Appendix A.3 Probing Tasks

In addition to code semantics, we further design a series of probing tasks to more intuitively measure the models' perception of program AST structural features. The overall process of the probing task is illustrated in Figure A3. In the first step, we record structural features from the AST corresponding to each program. Based on existing works [1–3], we choose **height**, **size**, **number of leaf nodes**, and **number of non-leaf nodes** of the AST as measuring features. Note that the number of non-leaf nodes can be derived from leaf nodes and size in Table B1. In the second step, we encode the program related to each AST into a vector representation using models that have been trained on the code classification task. In the final step, we use the generated vectors as inputs and recorded features as labels for the probing networks. Intuitively, the probing networks decode the vectors generated by the trained code representation models through a simple MLP to measure how comprehensiveness and accuracy the models encode the structural information of the code.

The AST height prediction task is designed to evaluate the models' perception of code hierarchy information, we regard it as a regression task and adopt Mean Absolute Error (MAE) as the evaluation metric. The corresponding MLP and loss function can be expressed as:

$$\hat{h} = W_2(W_1 \times v_r)$$
$$\text{loss} = |\hat{h} - h|$$

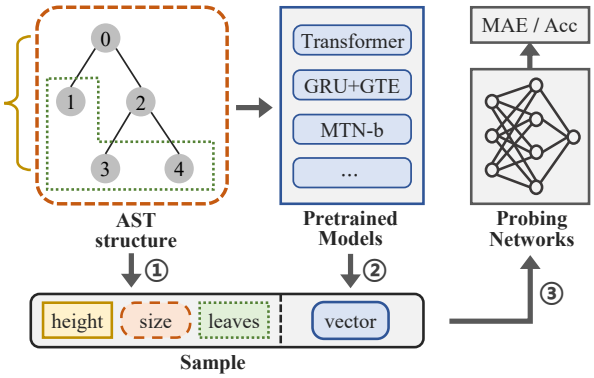

Figure A3 The process of probing task. ① Collect AST structure information; ② Generate code representation vector; ③ Take the combined sample as the input for probing tasks.

where W_1 is a weighted matrix for projecting the input vector to a hidden state, W_2 then generates the predicted height $\hat{h}$, h is the expected value.

The probing tasks that predict the size, leaf nodes, and non-leaf nodes of the AST are for measuring the models' awareness of the overall structure of the AST. Since accurate prediction using regression tasks is difficult due to the potentially significant differences in the size of ASTs across programs, we follow previous works [1, 2] to treat them as classification tasks and use the same MLP structure as in the height prediction task. Specifically, we divide the samples into small, medium, and large categories according to different AST size distributions. We omit the relevant formulas here for brevity.

Appendix B Experiment Design

Appendix B.1 Research Questions

To comprehensively evaluate the GTE-based models' ability for capturing code semantics and code structural information, we design the following research questions:

- **RQ1:** ***Performance on program classification tasks.*** This RQ aims to investigate the ability of GTE-based models in learning code semantics from several aspects, including GTE-based models v.s. other SOTA models, as well as the comparison between various GTE-based models.
- **RQ2:** ***Efficiency on training process.*** This RQ intends to compare the efficiency of GTE-based models with other code representation models, we measure the time cost and training convergence process of different models.
- **RQ3:** ***Contributions of components in GTE-based models.*** This RQ investigates how the different components in the GTE-based models affect the performance, which could help understand the contribution of different model design choices and AST ingredients.
- **RQ4:** ***Measure on structure encoding ability.*** This RQ demonstrates the ability of GTE-based models for preserving code AST structural information during training on the program classification task, which provides a degree of interpretability of the performance differences between the models.

Appendix B.2 Dataset

The GTE-based model is designed for encoding the AST structure of the code effectively and efficiently. We evaluate our model on two downstream tasks that are considered to have a strong correlation with code structure information. Where the program classification task evaluates the performance of models in distinguishing program semantics, and the probing task intuitively evaluates the ability of models in encoding code AST structures. Table B1 summarizes the datasets we use.

Table B1 Datasets for Evaluation.

	Program Classification				**Probing Task**
	Java250	**Python800**	**C++1000**	**C++1400**	**Java250(test)**
train	45,000	144,000	300,000	252,000	9,000
valid	15,000	48,000	100,000	84,000	3,000
test	15,000	48,000	100,000	84,000	3,000
total	75,000	240,000	500,000	420,000	15,000
average leaf nodes	230.13	134.66	316.35	380.71	230.46
average AST size	345.80	195.14	464.23	557.64	346.26
average blocks	13.79	10.07	10.99	11.69	13.77

Program classification. The program classification task is required to classify programs into multiple categories according to their functionalities. To better evaluate the model performance in learning code semantics of different programming languages, we utilize the Project CodeNet [4] dataset which has been properly preprocessed and possesses abundant program samples from Python, Java, and C++.

Probing Task. The probing task takes as input the vectors of trained code representations and decodes them using the MLP to probe whether the model encodes the desired features. In order to evaluate the trained models' ability to perceive the AST

structure of programs that have not been seen before, and include models such as MTN-b that are only valid on Java programs, we choose the testset of Java250 as the dataset for the probing task. To avoid the problem of data imbalance, we further divided the samples into SMALL, MEDIUM, and LARGE categories according to the distributions shown in Table B2.

Table B2 The Distribution of the samples in Java250(test).

	SMALL	**MEDIUM**	**LARGE**
number of AST nodes	<206	[206, 310]	>310
number of samples	4996	5121	4883
number of leaf nodes	<136	[136, 205]	>205
number of samples	4836	5136	5028
number of non-leaf nodes	<69	[69, 100]	>100
number of samples	5076	4928	4996

Appendix B.3 Baselines

We conduct a comprehensive comparison between the GTE-based model and various code representation models. We categorize them into three different categories according to the model architectures:

• **Tree-structured models.** We compare GTE with other tree-structured models that take the whole AST as input, including Code-RNN [5], Tree-LSTM [6], and MTN-b [7]. We also consider the TBCNN [8] and ASTNN [9] that perform convolution-like operations on the AST as baselines.

• **GNN-based models.** Code representation approaches based on graph neural networks including RGCN [10] and GGNN [11]. Since AST is the main input in this work, we did not consider GNN-based methods such as HPG [12] that use additional program information.

• **Transformer-based models.** We take the vanilla Transformer [13], the GREAT [14] model which injects the AST structure information into the self-attention mechanism, and the latest Transformer-based model HiT [15] which utilizes hierarchy information of AST structure as baselines. We also consider the code pre-trained models CodeBERT [16] and GraphCodeBERT [17].

Appendix B.4 Implementation and Training

We utilize the tree-sitter [1)] for parsing the programs to ASTs, and the DGL library [2)] for converting ASTs into graphs and performing data persistence operations. We build our model with PyTorch [3)] and the deep learning experiment management tool Sacred [4)].

For program classification, we set the hidden state dimension of GTE-based models to 512, the maximum number of epochs to 50, and the batch size to 64. In the training process, we adopted the AdamW optimizer, the learning rate is 0.0003 with linear decay after 5 warm steps. In order to make fair comparisons and control variables, we use flattened ASTs as input sequences and the same vocabulary as the GTE-based model for training the vanilla Transformer. For the probing task, the hidden state dimension of the MLP in all 4 tasks is set to 128. The model was trained on the Ubuntu18.04 system with 256G RAM, AMD-3970x CPU, and GeForce RTX4090 graphics card.

Appendix B.5 Metrics

We assess GTE's effectiveness based on Accuracy and MAE metrics for classification tasks and regression tasks, respectively. The metrics are computed as:

$$\text{Accuracy} = \sum_{i=1}^{|Y|} \mathbb{1}_{Y_i == \hat{Y}_i} \;/\; |Y|$$
$$\text{MAE} = 1/|Y| \sum_{i=1}^{|Y|} |Y_i - \hat{Y}_i|$$

where Y is the expected labels and $\hat{Y}$ is the predicted labels.

Appendix C Experimental Results

Appendix C.1 RQ1: Performance on Classification Tasks

To evaluate the ability of the GTE model to distinguish code semantics, we conducted evaluations on program classification tasks in three different program languages: Python, Java, and C++, the result is shown in Table C1. As shown in the table, The Transformer+GTE model performs the best among all baselines, with an accuracy of over 97.5% on each programming language, and an overall average accuracy of 98.03%, which improves 3.25% compared to the SOTA AST-implant Transformer-based model HiT.

When we look at the results in GNNs and Transformers in Table C1, we are surprised to find obvious performance increases over the original models simply by introducing the GTE framework without changing the model computation mechanism. After combining with the GTE framework, the average accuracy of the original GGNN and Transformer models improves by 3.26% and 10.1%, respectively. To explain this, we notice that programs are encoded hierarchically along the structure of ASTs via the GTE framework, which results in different effects on the GNN and Transformer architecture. On the one hand, GTE abandons the potentially redundant calculation process in the GNNs by specifying the transmission direction of the messages. On the other hand, by transforming the sequence model into a tree-structured model, GTE naturally introduces additional code hierarchy information without additional structure implantation steps. For further evaluation, we consider two large pre-trained models

1) https://tree-sitter.github.io/
2) https://www.dgl.ai/
3) https://pytorch.org/
4) https://github.com/IDSIA/sacred

Table C1 Accuracy for Program Classification Task.

	Model	Python800	Java250	C++1000	C++1400
GNNs	RGCN	91.60	91.93	92.73	92.34
	GGNN	92.23	93.64	91.72	92.48
	GGNN+GTE*	**97.29**	**94.83**	**95.62**	**95.37**
Tree-Structured	TBCNN	93.17	92.84	94.77	88.29
	ASTNN	93.80	92.86	94.61	90.17
	MTN-b*	-	92.43	-	-
	CodeRNN+GTE*	95.78	90.16	93.85	91.90
	GRU+GTE*	**97.82**	96.08	96.65	95.80
	TreeLSTM+GTE*	97.59	**97.37**	**97.86**	**97.74**
Transformers	Transformer*	95.49	92.53	83.72	79.97
	GREAT	93.27	93.36	92.76	92.50
	HiT	95.97	94.81	95.05	93.27
	Transformer+GTE*	**98.64**	97.73	**97.95**	**97.78**
	CodeBERT	97.41	96.47	86.13	83.05
	GraphCodeBERT*	98.21	**98.25**	-	-

∗ denotes our own experiment result. The best result in each architecture are bolded respectively.

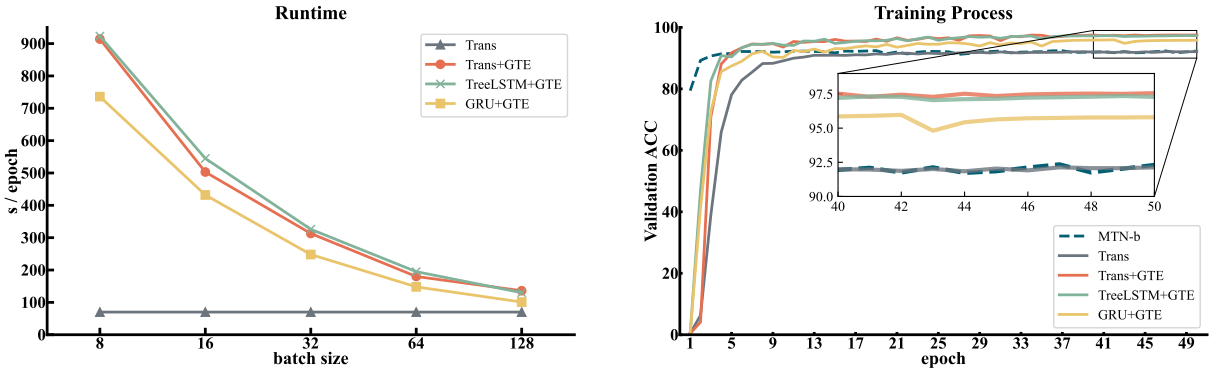

Figure C1 Runtime and training process on the Java250 dataset.

CodeBERT and GraphCodeBERT. Unexpectedly, the GTE-based model can achieve competitive or even better performance on the code classification task even though the pre-trained models possess much more parameters and code corpus, the overall average accuracy of the Transformer+GTE model is 7.26% higher than CodeBERT, on Python800 and Java250 datasets, Transformer+GTE almost achieves performance at the same level as GraphCodeBERT.

We also compared our recommended GTE-based model with existing tree-structured models. To speed up training, we integrated Code-RNN and Tree-LSTM into the GTE framework without changing the computational method of the models. We retain the original implementation of MTN-b, as it relies heavily on the grammar specification of the Java language. The results show that the TreeLSTM+GTE and GRU+GTE models perform better than other tree-structured models, even outperforming the transformer-based HiT model with an average accuracy of 2.86% and 1.81% respectively, and only slightly worse than Transformer+GTE (0.39% and 1.44%). This suggests that the capabilities of traditional RNN architectures such as GRU and LSTM in learning code representations may not have been entirely exploited.

Another interesting finding is that the GTE framework seems to alleviate the performance variation problem of some Transformer-based models on different programming languages, more precisely, the standard deviation of the results of Transformer+GTE is 0.36%, compared to 0.97% for HiT and 6.31% for Transformer. We attribute this to the absence of sequence truncation operations in the GTE framework, which prevents feature loss in programming languages with average longer code sequences, such as C++, the average code length could be inferred from the AST size in Table B1.

➨ *The GTE framework improves the performance of various model architectures on program classification tasks and alleviates the performance variation problem of the Transformer architecture to a certain extent.*

Appendix C.2 RQ2: Efficiency on Training Process

With the graph batch and graph block algorithms, the GTE model performs message passing and reducing simultaneously on multiple computation units. To evaluate the efficiency of models based on the GTE framework, we record the average runtime for a model to train an epoch and the validation accuracy of each epoch, the result is illustrated in Figure C1. Note that we adopt the Java250 dataset for this RQ to include the MTN-b model.

The result shows that the time cost of the GTE-based models to run an epoch drops obviously with the batch size increase. Specifically, the Transformer+GTE model takes 914 seconds to run an epoch with a batch size of 8, which is nearly 13 times slower than the Transformer (70 seconds), but its runtime decreases sharply to 136 when the batch size comes to 128. Compared to the sequential model, the GTE architecture requires block-by-block computation based on the AST hierarchy, so the reduction in time

Table C2 Ablation Study.

Model	Python800	Java250	C++1000	C++1400
GRU	95.38	91.31	82.84	79.11
RNN+GTE	94.30	87.08	92.71	90.03
GRU+GTE	**97.82**	**96.08**	**96.65**	**95.80**
GRU+GTE (w/o NL)	97.49	95.39	96.46	95.47
TreeLSTM+GTE	**97.59**	**97.37**	**97.86**	**97.74**
TreeLSTM+GTE (w/o NL)	97.14	96.97	97.62	97.53
Transformer	95.49	92.53	83.72	79.97
Transformer (w/o NL)	**95.83**	91.59	**87.55**	**85.18**
Transformer (w/ AP)	95.77	**92.89**	86.07	82.10
Transformer (w/ RP)	95.89	92.41	86.55	82.16
Transformer+GTE	98.64	97.73	97.95	97.78
Transformer+GTE (w/o NL)	98.63	97.37	97.94	97.75
Transformer+GTE (w/ AP)	**98.66**	97.18	97.97	97.88
Transformer+GTE (w/ RP)	98.64	**97.86**	**98.04**	**97.99**

w/o NL: Discard non-leaf nodes in the input AST
w/ AP: Use absolute position embedding
w/ RP: Use relative position embedding

consumption slows down as the batch size continues to increase. Nevertheless, we emphasize the huge advantage of the GTE-based model in terms of calculation speed compared to the traditional tree-structured model, e.g., the MTN-b model costs 10,426 seconds to run an epoch, which makes it unaffordable to train on the large-scale dataset. Note that we omit the runtime of MTN-b in the figure for better illustration.

We also report the converging process of different models on the program classification task. Note that the performance of the MTN-b model grows faster in the first 5 epochs compared to other models as we keep the original training settings that do not set any warm-up steps for it. It can be seen from Figure C1 that the GTE-based models can reach relatively higher accuracy on the validation dataset with fewer training steps compared to the Transformer model.

➨ *The models constructed on the GTE framework significantly reduce the training overhead with larger batch sizes and achieve greater performance in the early step compared to the sequence model.*

Appendix C.3 RQ3: Contributions of Different Components

Within a model architecture, different model designs and input components can contribute to performance in varying degrees, and the introduction of the GTE framework may further lead to new changes. In this research question, we mainly focus on the influence of computation units, non-leaf nodes, and position information on the performance of the GTE-based models, the result of the ablation study is shown in Table C2.

Computation Units. The result shows that the computation units based on different model architectures perform differently in understanding code semantics. The Transformer+GTE based on the self-attentive mechanism performs the best, with an average accuracy of 98.03% on all programming languages, followed by TreeLSTM+GTE (97.64%) and GRU+GTE (96.59%). To further illustrate the important role of the computation unit, we replaced the GRU computation units in the GRU+GTE model with a most basic RNN unit, which resulted in a significant performance drop, the average accuracy of RNN+GTE model decline of 5.56%.

Non-leaf Nodes. The nodes in the AST can be divided into leaf nodes, which usually consist of identifiers associated with the code text, and non-leaf nodes, which are used to describe the syntactic structure information of the program through the type attribute of the node, but sometimes the corresponding text cannot be found directly in the code. Inferred from the statistics in Table B1, we believe that the impact of non-leaf nodes on model performance is worth exploring since the non-leaf nodes can account for a considerable proportion of the overall AST size (e.g., 33% for Java and 31% for Python). We can see from the result that the GTE-based models suffer from a consistent performance decrease after discarding the non-leaf nodes in the inputs, in contrast to the Transformer model whose average accuracy increases by 2.11%. This phenomenon shows that compare to the sequence models, the GTE-based models are more likely to make better use of non-leaf nodes in the AST. However, despite accounting for nearly one-third of the total AST size, the contribution of non-leaf nodes to the model ability is minimal, especially in the Transformer+GTE model, where the average accuracy is only slightly reduced by 0.11% after removing the non-leaf nodes.

Position Information. One of the shortcomings of the sequence model is its inability to discriminate the position of the input sequence, previous studies have shown that the use of absolute or relative position encoding can alleviate this problem and enhance the performance in specific tasks. The result of the Transformer model in Table C2 shows the effectiveness of absolute position encoding in helping the model discern code semantics, leading to an average accuracy improvement of 1.28%. When it comes to Transformer+GTE, we find that absolute and relative positional encoding barely improves model capabilities, with accuracy gains of no more than 0.3% on any languages, and the absolute positional coding even leads to worse performance on the Java dataset. It indicates that the GTE-based models partially alleviate the dependence on extra position encoding.

➨ *The Transformer computation unit is well adapted to the GTE framework, while the non-leaf nodes and position encoding make limited contributions to the capabilities of the GTE-based model on the program classification task.*

Appendix C.4 Measure on Structure Encoding Ability

To further investigate the relationship between structural information and specific downstream task, we design four probing tasks to measure the perception of GTE models on code structural information. We use the Java250 dataset for validation and exclude other types of models to avoid interference from the model architecture, the result is shown in Figure C2.

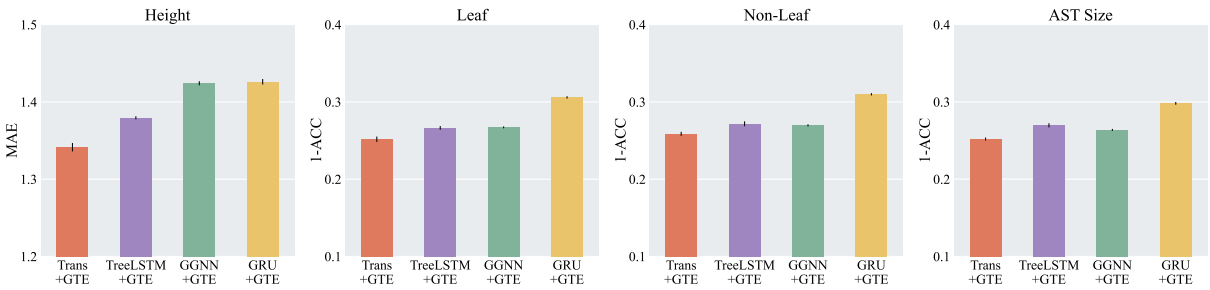

Figure C2 Probing task results on the Java250 dataset.

The result shows that the Transformer+GTE model performs best on all four probing tasks, followed by TreeLSTM+GTE, and then the GRU+GTE model. Interestingly, this performance ranking also holds for the program classification task shown in Figure C1, which means that GTE-based models that perform better in code classification tasks tend to have superior code structure awareness. However, an exception arises in the GGNN+GTE model, which is inferior to the GRU+GTE in the code classification task but outperforms the GRU+GTE model in all four probing tasks, especially in the three tasks of perceiving the overall structure of the code. We attribute this to the different mechanisms between computation units. In the GGNN computation unit, the GRU cell is "vertical" that takes the sum of child nodes as input, but the GRU cell in the GRU computation unit is "horizontal" which takes child nodes as a sequence of inputs and may lead to information loss when facing long input sequence. This exception warns us of the potential gap between learning code structural information and the specific downstream tasks.

➨ *The GTE-based model that performs better in program classification tasks tends to learn more accurate code structures in most cases, but a potential gap between structural information and downstream tasks could threaten this pattern.*

References

1 Karmakar A, Robbes R. What do pre-trained code models know about code? In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021. 1332–1336
2 Troshin S, Chirkova N. Probing pretrained models of source code. arXiv preprint arXiv:2202.08975, 2022
3 Yang, Kang, Xinjun Mao, Shangwen Wang, et al. An extensive study of the structure features in transformer-based code semantic summarization. In 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), pp. 89-100. IEEE, 2023
4 Puri R, Kung D S, Janssen G, et al. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655, 2021
5 Liang Y, Zhu K. Automatic Generation of Text Descriptive Comments for Code Blocks. Proceedings of the AAAI Conference on Artificial Intelligence, 2018, 32
6 Tai K S, Socher R, Manning C D. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2015. 1556–1566
7 Wang W, Li G, Shen S, et al. Modular Tree Network for Source Code Representation Learning. ACM Trans. Softw. Eng. Methodol., 2020, 29
8 Mou L, Li G, Zhang L, et al. Convolutional Neural Networks over Tree Structures for Programming Language Processing. Proceedings of the AAAI Conference on Artificial Intelligence, 2016, 30
9 Zhang J, Wang X, Zhang H, et al. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019. 783–794
10 Schlichtkrull M, Kipf T N, Bloem P, et al. Modeling Relational Data with Graph Convolutional Networks. In: The Semantic Web, 2018. 593–607
11 Li Y, Zemel R, Brockschmidt M, et al. Gated Graph Sequence Neural Networks. In: Proceedings of ICLR'16, 2016.
12 Zhang K, Wang W, Zhang H, et al. Learning to Represent Programs with Heterogeneous Graphs. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Virtual Event, 2022. 378–389
13 Vaswani A, Shazeer N, Parmar N, et al. Attention is All you Need. In: Advances in Neural Information Processing Systems, 2017.
14 Hellendoorn V J, Sutton C, Singh R, et al. Global Relational Models of Source Code. In: International Conference on Learning Representations, 2020.
15 Zhang K, Li Z, Jin Z, et al. Implant global and local hierarchy information to sequence based code representation models. In: 2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC), 2023. 157–168
16 Feng Z, Guo D, Tang D, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020, 2020. 1536–1547
17 Guo D, Ren S, Lu S, et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In: International Conference on Learning Representations, 2021.