SCIENCE CHINA
Information Sciences

• LETTER •

March 2025, Vol. 68, Iss. 3, 139101:1–139101:2
https://doi.org/10.1007/s11432-024-4262-5

# 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*

Received 23 April 2024/Revised 26 August 2024/Accepted 26 December 2024/Published online 8 February 2025

**Citation** Qin Y H, Wang S W, Lin B, et al. GTE: learning code AST representation efficiently and effectively. Sci China Inf Sci, 2025, 68(3): 139101, https://doi.org/10.1007/s11432-024-4262-5

With the development of deep learning in recent years, code representation learning techniques have become the foundation of many software engineering tasks such as program classification [1] and defect detection. Earlier approaches treat the code as token sequences and use CNN, RNN, and the Transformer models to learn code representations. More recently, to further incorporate the structural information of code, efforts have been made by building tree-structured models (TSM) [2] or implanting structural knowledge into sequenced-based models (SBM) [1].

However, the TSMs focus on encoding code structure by taking the whole abstract syntax tree (AST) of code as input, but are inferior at processing sequence input and suffer from overhead problems due to the irregular shape of ASTs. On the contrary, the AST-implanted SBMs are superior at handling long input sequences but usually rely on extracted structural features such as sampled context paths or relative distance between tokens, which prevent models from understanding code structure precisely and completely.

In this work, we try to remove the limitations of TSM and SBM by proposing the graph-based tree encoding (GTE) framework, which makes it possible to learn code representation both efficiently and effectively. Inspired by the message-passing paradigm, GTE first transforms a batch of ASTs into a large directed graph and then splits it into several blocks along the hierarchy of the AST. Next, GTE supports message passing and updating on a bucket of computation units within a block, where the parent node in each unit has the same number of children. With the help of the computation units, powerful sequence models such as Transformer can be easily integrated into the GTE framework without any additional modification.

*Methodology.* The overall procedure of the GTE framework can be summarized into five phases.

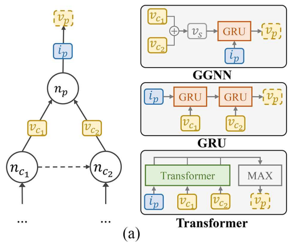
**Phase 1: preprocess.** We first parsed the code snippets $C$ into abstract syntax trees $T$. Then, by performing a depth-first traversal, we treated all nodes in an AST $t \in T$ as the set of nodes $N$ in its corresponding graph $g$, where each node's attribute is the AST node type (if it is a non-leaf node) or the code token of the AST node (if it is a leaf node). We constructed the edge set $E$ in the graph based on the parent-child relationships between nodes in the AST. In addition, for each graph $g$, we also record its number of nodes $s$ and its corresponding AST height $h$ for further processing. Formally, a code snippet $c \in C$ can be represented as $g = (N, E, s, h) \in G$, where $G$ is the set of all code graphs.

**Phase 2: graph batching and blocking.** The graph batch algorithm is commonly used for merging a batch of small graphs into a large graph, where each graph before merging corresponds to a sub-graph in the merged large graph, the nodes in the sub-graph are not connected to other nodes in the large graph. In this way, the calculations can be performed on the entire merged graph simultaneously. This algorithm is commonly utilized to accelerate computation and improve the generalization performance of GNN models. To represent the structural information of the entire code snippet, GNNs typically require a pooling layer that can effectively gather the learned knowledge in nodes into a single output. Since the design of the pooling process is non-trivial, we introduce the graph blocking algorithm to regulate the direction of data aggregation within the graph.

The basic intuition of the graph blocking 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 detailed steps of the algorithm are shown in Appendix A.

**Phase 3: GTE-based model.** Intuitively, the graph-based tree encoding (GTE-based) model possesses characteristics of both GNN and tree structure models. The left part of Figure 1(a) shows a GTE computation unit, supposing the model has calculated the hidden state $v_{c_1}$ and $v_{c_2}$ of sub-trees that take nodes $n_{c_1}$ and $n_{c_2}$ as roots, $i_p$ is cal-

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



(a)

**Table 1** Accuracy for Program Classification Task.

| Model | Py800 | Java250 | Cpp1000 | Cpp1400 |
|---|---|---|---|---|
| GGNN | 92.23 | 93.64 | 91.72 | 92.48 |
| GGNN+GTE | **97.29** | **94.83** | **95.62** | **95.37** |
| ASTNN | 93.80 | 92.86 | 94.61 | 90.17 |
| MTN-b | - | 92.43 | - | - |
| GRU+GTE | **97.82** | **96.08** | **96.65** | **95.80** |
| TF | 95.49 | 92.53 | 83.72 | 79.97 |
| HiT | 95.97 | 94.81 | 95.05 | 93.27 |
| TF+GTE* | **98.64** | **97.73** | **97.95** | **97.78** |

(b)

**Figure 1** (Color online) (a) Computation mechanism of the GTE framework; (b) accuracy for program classification task. The bold numbers represent the best results under different model architectures.

culated from the attribute of node $n_p$, a GTE computation unit aims to aggregate the information reserved in $v_{c_1}$, $v_{c_2}$, and $i_p$, and to represent the sub-tree which take node $n_p$ as root by a generated hidden state $v_p$.

As the computation processes between parent and child nodes have been abstracted into computation units, various GNN and sequence-based models can be integrated into the GTE framework. For the RNN architecture, we integrate the widely used GRU model into our GTE framework. The calculation process can be represented as follows:

$$v_{\text{in}} = [v_{c_1}; v_{c_2}; \cdots ; v_{c_n}],$$
$$v_p = \text{LayerNorm}(\text{GRU}(v_{in}, i_p)).$$

We also integrate the Transformer architecture into the GTE framework. The process is as follows:

$$v_{\text{in}} = [i_p; v_{c_1}; v_{c_2}; \cdots ; v_{c_n}],$$
$$v_{\text{out}} = \text{Transformer}(v_{\text{in}}),$$
$$v_p = \text{MaxPooling}(v_{\text{out}}).$$

Since the GTE framework limits the length and direction of information flow between nodes in the graph, it is worthwhile to investigate the influence of this limitation on the original GNN:

$$v_s = \sum_{i=1}^{n} v_{c_i},$$
$$v_p = \text{LayerNorm}(\text{GRU}(i_p, v_s)).$$

Unlike other tree-structured models, similar calculation processes in the GTE framework would occur simultaneously in a batch of computation units in a block. We formally present the propagation process as

$$m_{j\to k} = (i_k^{(b)}, v_j^{(b)}), (j,k) \in E,$$
$$v_k^{(b+1)} = \rho(\{m_{j\to k}, (j,k) \in E\}).$$

**Phase 4: downstream task.** For evaluating the model performance on distinguishing code semantics, we apply the generated vectors to program classification tasks. Since no pooling operation is required to aggregate different node representations, we simply take the hidden state of the root node $v_r$ as the output, and use a single fully connected layer and a softmax layer to predict the probability of each category:

$$o = W \times v_r, \quad p = \text{softmax}(o),$$
$$\text{loss} = -\sum_{i=1}^{|p|} \mathbb{1}_{y==i} \log p_i.$$

*Results and discussion.* We evaluate the models' performance in code understanding on the Project CodeNet [3] dataset, which has been properly preprocessed for different programming languages including Python, Java, and C++. We compare the GTE-based model with code representation models in various architectures such as GGNN [4], ASTNN [5], MTN-b [2], and HiT [1]. The results are shown in Figure 1(b), where TF denotes the Transformer model.

As shown in Figure 1(b), 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 GNN and SBM models, we can 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 evaluate the efficiency of models based on the GTE framework, we record the average runtime for a model to train an epoch. The result shows that the time cost of the GTE-based models to run an epoch drops obviously with the increase in batch size. Specifically, the Transformer+GTE model takes 914 s to run an epoch with a batch size of 8, which is nearly 13 times slower than the Transformer (70 s), but its runtime decreases sharply to 136 when the batch size comes to 128. More detailed evaluation results can be found in Appendix C.

*Conclusion and future work.* In this work, we try to address the limitations of tree-structured models and sequence-based models by introducing the GTE framework, which can encode AST "naturally" in a bottom-up manner with sequence models, while also conducting parallel calculations efficiently on multiple ASTs. For future work, it would be interesting to apply the GTE framework to more complex models such as the large language models.

**Supporting information** Appendixes A–C. The supporting information is available online at info.scichina.com and link.springer.com. The supporting materials are published as submitted, without typesetting or editing. The responsibility for scientific accuracy and content remains entirely with the authors.

## References

1 Zhang K, Li Z, Jin Z, et al. Implant global and local hierarchy information to sequence based code representation models. In: Proceedings of IEEE/ACM 31st International Conference on Program Comprehension (ICPC), 2023. 157–168
2 Wang W, Li G, Shen S, et al. Modular tree network for source code representation learning. ACM Trans Softw Eng Methodol, 2020, 29: 1–23
3 Puri R, Kung D S, Janssen G, et al. Project CodeNet: large-scale AI for code dataset for learning a diversity of coding tasks. 2021. ArXiv:2105.12655
4 Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: Proceedings of International Conference on Learning Representations, 2018
5 Zhang J, Wang X, Zhang H, et al. A novel neural source code representation based on abstract syntax tree. In: Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019. 783–794