Enhancing Code Completion with Round Splitting and Unique Traversal of Abstract Syntax Tree

Jiahao Li[1,2,a], Linbo Zhu[2,b*], Bowen Lv[1,c], Jun Ding[2,d]

[1]AHU-IAI AI Joint Laboratory, Anhui University, Hefei, China
[2]Institute of Artificial Intelligence, Hefei Comprehensive National Science Center, Hefei, China

[a]WA22301177@stu.ahu.edu.cn,[b]lbzhu@iai.ustc.edu.cn,
[c]WA21301045@stu.ahu.edu.cn,[d]djun@iai.ustc.edu.cn

Abstract. Code completion is one of the crucial features of Integrated Development Environments, enhancing user coding efficiency by providing code suggestions. Research indicates that code completion methods based on Abstract Syntax Tree (AST) representations can extract rich syntax and structure information embedded in the code. However, the current processing of the AST often results in a significant loss of information. To address this, we have designed round-splitting and unique traversal algorithms to optimize the AST processing. The round-splitting algorithm achieves tree splitting with minimal disruption, preserving the structural information of the tree to the greatest extent. The unique traversal algorithm ensures a one-to-one mapping relationship between the tree and the sequence after traversal, thereby reducing information loss. We conducted experiments on benchmark datasets, demonstrating our algorithms' effectiveness in the code completion task.

Keywords: code completion, abstract syntax tree, round-splitting algorithm, unique traversal algorithm

1 Introduction

Code completion is a vital feature within integrated development environments. While users are writing code, the system provides a suggestion list containing keywords, variable names, and other content based on the context of the user's code.

To implement code completion, researchers initially tried non-machine learning methods such as type analysis [1]. However, these methods showed significant limitations when facing complex semantic structures. Traditional machine learning methods such as N-gram still could not effectively solve these problems [2][3]. Therefore, researchers are mainly using deep learning models to implement code completion [4][5]. The essence of code completion is a generative task, and GPT series models can better complete such tasks, so we choose the GPT-2 model as our core model [6]. As a special kind of text, code contains rich structural and semantic information, so it is important to choose a reasonable representation. To retain as much code information as possible

T. Yao et al. (eds.), *Proceedings of the 2024 3rd International Conference on Engineering Management and Information Science (EMIS 2024)*, Advances in Computer Science Research 111,
https://doi.org/10.2991/978-94-6463-447-1_32

while minimizing resource consumption, we chose Abstract Syntax Tree (AST) for code representation.

After representing the code as an AST, it is necessary to perform split and traversal operations on the tree. However, current mainstream methods tend to be goal-oriented, resulting in the loss of information from the AST. Therefore, based on AST representation, we aim to retain as much rich information from the tree as possible, and we have optimized the split and traversal operations of the AST. Our contributions include the following:

- We propose a round-splitting algorithm for the AST, which refines the split granularity of the AST and reduces the destruction of the tree structure.
- After achieving the split of large trees, we have designed the unique traversal algorithm to ensure a one-to-one mapping in the traversal process, preserving the complete structure of the tree.
- We have demonstrated that using the above algorithms has a beneficial effect on the code completion task, and the benefits of the two algorithms can be superimposed.

2 Background

2.1 Code Completion Task

The code completion task studied in this paper is token-level code completion, completing the minor units in the source code compilation process, such as variable names and keywords. Our core task is to calculate the probability of all tokens in a given dictionary when a part of the program is provided and return a recommendation list after sorting by likelihood. The formula is defined as follows:

$$List = sort\big(P(m|C)\big), \forall m \in M \tag{1}$$

In this paper, C represents a code snippet, M denotes the dictionary, and m stands for a specific token within the dictionary. The $sort$ stands for a sorting operation, and it will place the recommendations with higher probabilities at the front.

2.2 Abstract Syntax Tree (AST)

AST is transformed from the context-free grammar of the code, which reasonably stores the syntax and structural information of the code [7]. Because of this, the size of the AST is substantial. The GPT-2 model has input size limitations, and it is difficult for the model to capture the rich information in AST that is large in size and depth [8]. Therefore, it is necessary to perform a split operation on the AST, dividing an overly large AST into reasonably sized subtrees as required. For our model to receive the AST, it needs to be traversed into a sequential form.

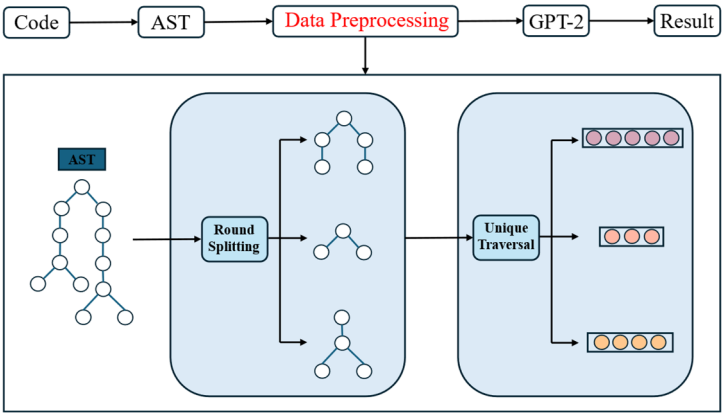

Fig. 1. The whole process to achieve code completion.

3 Approach

As shown in Figure 1, the whole process to achieve code completion is displayed. After converting the code into the AST, our round-splitting algorithm divides the large AST into reasonably sized subtrees. Then, using our unique traversal algorithm, subtrees are traversed into a sequential form. After that, the sequence is encoded and fed into the GPT-2 model for learning. Finally, the model is used to obtain the result. In this section, we will focus on the data preprocessing stage and describe how to optimize the splitting and traversal of the AST by our algorithm.

3.1 Round Splitting Algorithm

The current mainstream split method is the sliding window split method, which dramatically disrupts node dependency relationships [9]. To address this issue, we analyzed the node characteristics of the AST and found that certain specialized nodes contain structural information. The nodes, like class nodes, often represent the beginning of a code structure and are very suitable as split points. Furthermore, to preserve the integrity of the code structure as much as possible, the size of the structure that these nodes contained can be further distinguished, which can be used as a standard to set split rounds. As shown in Figure 2, we designed the round-splitting algorithm.

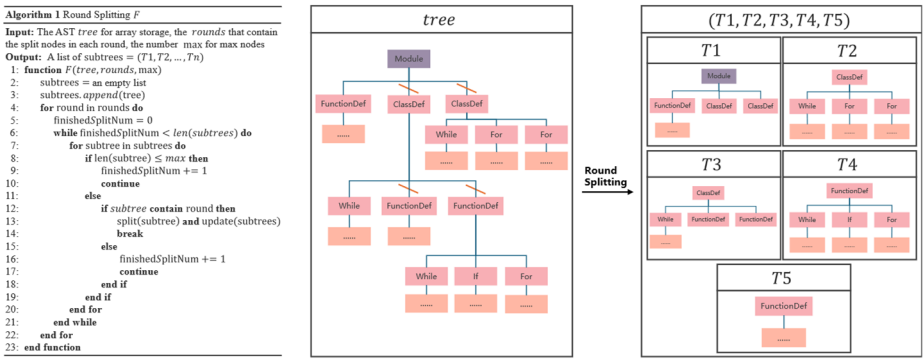

Fig. 2. The round-splitting algorithm and an example.

The round-splitting algorithm is based on three main strategies:

1. The round-splitting is performed according to the size of the structure information contained in the nodes. In round 1, nodes representing classes are split. In round 2, nodes representing functions are split. In round 3, nodes representing the loop or condition are split. Strategy 1 preserves large-scale code structures as much as possible.
2. Upon completing each split, immediately evaluate whether the splitting termination condition is met and terminate the split promptly, reducing meaningless splits.
3. Both trees retain the split node after splitting to preserve information further.

Figure 2 also shows an example. Our algorithm selects specialized nodes as split points according to the split rounds, effectively reducing the disruption of node dependencies during the splitting process. When the round-splitting algorithm fails to achieve the splitting objective, the sliding window splitting method is used for post-processing.

3.2 Unique Traversal Algorithm

The mainstream traversal methods currently in use are depth-first traversal or breadth-first traversal. Although these two algorithms can convert a tree into a unique sequence, restoring the original tree structure from this sequence is not unique. In other words, neither method provides a one-to-one mapping traversal, which will result in missing parent-child or sibling relationships between nodes in the traversed sequence. To solve this problem, it is necessary to provide an algorithm to preserve the containment relationships between nodes, and then reduce the loss of information in the traversal process. We were inspired by how compilers mark the start and end of code blocks with "start" and "end" identifiers during syntax analysis, thereby establishing correct syntax or parse trees. As shown in Figure 3, we designed the unique traversal algorithm.

Algorithm 2 Unique Traversal F

Input: The AST $tree$ for array storage
Output: A flattened sequence seq

```
function F(tree, seq = None)
    if seq is None then
        seq = an empty list
    end if
    root = the root of tree
    if root is a leaf then
        content = the value of root
        seq.append(content)
    else
        content = the type of root
        seq.append(content :: start)
        for child in children of root do
            F(child, seq)
        end for
        seq.append(content :: end)
    end if
end function
```

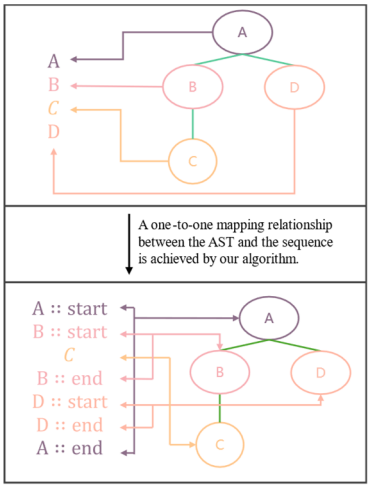

Fig. 3. The unique traversal algorithm and an example.

This algorithm is deterministic. Given an AST as input, it produces a unique sequence. The uniqueness of the conversion from sequence to tree can be proven using mathematical induction. As shown in Figure 3, the example shows that our algorithm achieves a one-to-one mapping relationship between the AST and the sequence, reducing the loss of tree structure information during the traversal process.

4 Experiments

This section will introduce the datasets used in the experiment and the evaluation metric. Subsequently, we will elaborate on the details of the related experiments. Finally, we will analyze the results of the experiments.

4.1 Dataset and Evaluation Metric

The dataset used in this study is the PY150 dataset, a public dataset in the code completion field [10]. The training set includes 100k code snippets; the test set contains 50k. Because the code completion task is a recommendation task, we chose Mean Reciprocal Rank (MRR) as the evaluation metric for our experiment. The MRR formula is as follows:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \tag{2}$$

Where n represents the number of queries, and $rank_i$ is the position of the first relevant document in the i query.

4.2 Experiment Detail

To evaluate the effectiveness of our model more comprehensively, we selected four mainstream code completion methods as baselines, namely Code2Seq, SeqRNN, SeqTrans, and TravTrans [9][11]. These methods treat code directly as text or use AST to represent code and then use LSTM or GPT-2 as the base models. We built the base models used by the baselines. The GPT-2 model contains only six decoder modules.

4.3 Results

The experimental results are shown in Table 1, which displays the performance of various code completion methods. The CodeSeq and TrvaTrans methods, which use AST representations, do not show comprehensive advantages over the SeqRNN and SeqTrans methods that treat code directly as text. The reason is that these methods do not utilize the rich information contained in the trees effectively. Therefore, our model, called "CodeRU", has significantly improved task performance after enhancing the use of AST through our two innovative algorithms.

Table 1. MRR of Baseline and Our Model

	Prior work				**Our model**
Application	Code2Seq	SeqRNN	SeqTrans	TravTrans	CodeRU
Attribute	39.3%	51.7%	69.6%	64.3%	**67.1%**
Numeric	49.5%	47.5%	63.8%	64.1%	**67.3%**
Name	45.8%	46.5%	64.5%	69.9%	**72.3%**
Parameter	56.8%	66.0%	74.3%	66.7%	**69.2%**
All	43.7%	67.4%	75.6%	79.9%	**81.5%**

Subsequently, we conducted ablation experiments to verify the roles of the round-splitting and unique traversal algorithms separately. As shown in Table 2, when our two algorithms are used individually, the overall effect improves compared to the baseline. When the two algorithms are combined, the model achieves the best performance. Experiments prove that our two algorithms are effective, and the gain effect can be superimposed.

Table 2. Ablation Study

	Effects of different parts			**Our model**
Application	sliding window split depth-first traversal	round-splitting depth-first traversal	sliding window split unique traversal	round-splitting unique traversal
Attribute	64.3%	67.1%	64.3%	**67.1%**
Numeric	64.1%	66.5%	64.1%	**67.3%**
Name	69.9%	72.1%	70.1%	**72.3%**
Parameter	66.7%	68.3%	68.0%	**69.2%**
All	79.9%	81.2%	80.2%	**81.5%**

5 Conclusions

We delved into code completion methods predicated on the representation of AST and have developed a round-splitting algorithm and a unique traversal algorithm to address the shortcomings in AST split and traversal preprocessing steps. The effectiveness of our algorithms has been empirically validated through our experiments.
Compared to past work, our model can better learn the rich syntactic and structural information embedded within the code by mitigating information loss. However, the study of positional encoding is temporarily missing from our model. In the future, we will try to combine the AST to construct the feature positional encoding, so that the model can learn the positional information of the AST.

References

1. Tu Z, Su Z, Devanbu P. On the localness of software[C]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014: 269-280.
2. Devanbu P. On the naturalness of software[C]//Proceedings of the 6th India Software Engineering Conference. 2013: 61-61.
3. Raychev V, Bielik P, Vechev M. Probabilistic model for code with decision trees[J]. ACM SIGPLAN Notices, 2016, 51(10): 731-747.
4. Li J, Wang Y, Lyu M R, et al. Code completion with neural attention and pointer networks[J]. arXiv preprint arXiv:1711.09573, 2017.
5. Izadi M, Gismondi R, Gousios G. Codefill: Multi-token code completion by jointly learning from the structure and naming sequences[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 401-412.
6. Radford A, Narasimhan K, Salimans T, et al. Improving language understanding by generative pre-training[J]. 2018.
7. Lin C, Ouyang Z, Zhuang J, et al. Improving code summarization with block-wise abstract syntax tree splitting[C]//2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). IEEE, 2021: 184-195.
8. Shi E, Wang Y, Du L, et al. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees[J]. arXiv preprint arXiv:2108.12987, 2021.
9. Kim S, Zhao J, Tian Y, et al. Code prediction by feeding trees to transformers[C]//2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021: 150-162.
10. Lu S, Guo D, Ren S, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation[J]. arXiv preprint arXiv:2102.04664, 2021.
11. Alon U, Brody S, Levy O, et al. code2seq: Generating sequences from structured representations of code[J]. arXiv preprint arXiv:1808.01400, 2018.