Structural-semantics Guided Program Simplification for Understanding Neural Code Intelligence Models

Chaoxuan Shi
State Key Lab for Novel Software Technology, Nanjing University
Jiangsu, Nanjing, China
cxshi@smail.nju.edu.cn

Tingwei Zhu
State Key Lab for Novel Software Technology, Nanjing University
Jiangsu, Nanjing, China
tingweizhu33@smail.nju.edu.cn

Tian Zhang*
State Key Lab for Novel Software Technology, Nanjing University
Jiangsu, Nanjing, China
ztluck@nju.edu.cn

Jun Pang*
FSTM & SnT, University of Luxembourg
Esch-sur-Alzette, Luxembour
jun.pang@uni.lu

Minxue Pan*
State Key Lab for Novel Software Technology, Nanjing University
Jiangsu, Nanjing, China
mxp@nju.edu.cn

Abstract

Neural code intelligence models are cutting-edge automated code understanding technologies that have achieved remarkable performance in various software engineering tasks. However, the lack of deep learning models' interpretability hinders the application of deep learning based code intelligence models in real-world scenarios, particularly in security-critical domains. Previous studies use program simplification to understand neural code intelligence models, but they have overlooked the fact that the most significant difference between source code and natural language is the code's structural semantics.

In this paper, we first conduct an empirical study to identify the critical code structural semantic features valued by neural code intelligence models, and then we propose a novel program simplification method called SSGPS (Structural-Semantics Guided Program Simplification). Results on three code summarization models show that SSGPS can reduce training and testing time by 20-40% while controlling the decrease in model performance by less than 4%, demonstrating that our method can retain the critical code structural semantics for understanding neural code intelligence models.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Feature selection**.

Keywords

Neural Code Intelligence Model, Program Simplification, Interpretable AI, Code Structural Semantics

*Corresponding author

Internetware '23, August 4–6, 2023, Hangzhou, China

ACM ISBN 979-8-4007-0894-7/23/08…$15.00
https://doi.org/10.1145/3609437.3609438

ACM Reference Format:
Chaoxuan Shi, Tingwei Zhu, Tian Zhang, Jun Pang, and Minxue Pan. 2023. Structural-semantics Guided Program Simplification for Understanding Neural Code Intelligence Models. In *14th Asia-Pacific Symposium on Internetware (Internetware '23), August 4–6, 2023, Hangzhou, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3609437.3609438

1 Introduction

Neural code intelligence models are now the cutting-edge technologies for automated code understanding and have achieved remarkable performance in a variety of software engineering tasks, including code summarization [19], code search [3], and vulnerability detection [4]. The powerful generalization performance of deep neural networks enables these models to learn rich representations by passing raw data through several transforming layers [13]. As a result, the burden of feature engineering is significantly reduced in challenging fields such as computer vision, natural language processing, and source code understanding.

Before the popularity of deep learning, researchers typically rely on their domain expertise to solve automated code intelligence tasks using heuristic rules [20, 21], pattern matching [6, 10], or other techniques. As a result, their tools are quite explainable, enabling them to provide users with a convincing explanation of the tool's effectiveness. However, deep learning based code intelligence models lack this ability. Despite years of research dedicated to interpreting deep neural models' behavior, they stubbornly maintain their "black box" characteristics [15]. The lack of deep neural models' interpretability hinders the application of neural code intelligence models in real-world scenarios, especially in security-critical domains.

Previous studies have attempted to understand neural code intelligence models by using program simplification techniques based on delta debugging [17, 18, 23] or attention weights [2, 31]. However, most of these studies only regard source code as plain text, overlooking the fact that the most significant difference between source code and natural language lies in the code's structural semantics. Meanwhile, several researchers have incorporated the structural representation of code, such as the Abstract Syntax Tree (AST), as multi-modal auxiliary information into neural code intelligence models, resulting in better performance than models solely based

on plain source code [7, 9]. Therefore, it is essential to identify the key code structural semantic features that neural code intelligence models focus on. This will help us understand why code structural representation can aid neural code intelligence models better completing various software engineering tasks.

The attention mechanism provides a way to evaluate the importance of each input element of the neural model, and the element with a higher attention weight is generally more essential [2]. To uncover the critical structural semantics learned by neural code intelligence models, we first conduct an empirical study on SiT [29] — a code summarization model based on structure-induced Transformer and using AST as input. Our study aims to find out the following:

- What type of AST node does the model pay more attention to when encoding representations and generating predictions?
- Is there a dissimilar distribution pattern for the impact of different node types to their neighbors when encoding representations?

To answer the above two questions, we extract the attention weights obtained by each node type in the SiT encoder and decoder and use them to evaluate the significance of different node types. Our findings show that the model pays more attention to node types related to rich functional semantics such as method signatures and member calls. Besides, the impact distribution of different node types to their neighbors is distinct. Some node types have a critical impact on all their neighbors, while others only make sense to neighbors corresponding to the same grammar structure.

Based on our empirical findings, we propose a novel program simplification method called SSGPS (Structural-Semantics Guided Program Simplification). SSGPS simplifies the program at the AST level while retaining the essential structural semantic features of the program and eliminating unnecessary information. SSGPS considers both the node's contribution to encoding representation and generating predictions and adopts a two-stage pruning scheme to enhance the reduction efficiency.

We apply SSGPS to three code summarization models, namely SiT [29], SCRIPT [8], and AST-Trans [24]. We measure the simplification performance of SSGPS by relative size and time cost in training and testing. Meanwhile, we assess whether the simplified program retains the key structural semantics by examining the decrease in the baseline models' evaluation scores. Experimental results show that SSGPS can reduce training and testing time by 20-40% while controlling the decrease in models' performance by less than 4%, which indicates that SSGPS successfully retains the structural semantic information that the neural code intelligence model attaches importance to.

Overall, our contributions in this paper are threefold:

- We conduct an in-depth study about the critical code structural semantic features valued by neural code intelligence models.
- We propose a novel program simplification approach SSGPS based on our empirical findings, which simplifies the program at the AST level while preserving the key code structural semantics.
- We extensively evaluate SSGPS on three code summarization models and our experimental results demonstrate the effectiveness of SSGPS.

Structure of the paper. Section 2 introduces the background knowledge and related work of our research, and Section 3 provides our empirical study and its results. We present our novel program simplification method SSGPS in Section 4, while Section 5 records the setup, results, and corresponding analysis of our experiments. Section 6 analyzes the threats to the validity of our research and Section 7 concludes the paper with future work.

2 Background and Related Work

This section provides background information on our research, including code structural representation and the attention mechanism. We also introduce related work on code intelligence models understanding based on program simplification.

2.1 Background

Code structural representation. Code structural representations such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG), are constructed from plain source code and describe the structural semantics of source codes. These code structural representations play an essential role in source code comprehension, helping us understand the code's functionality and concrete behaviors. Of these representations, AST is the most widely used in current code intelligence tasks, likely because it provides richer structural semantics than CFG and DFG. In this paper, we focus on the importance of different AST node types and mine the key structural semantic features that are valuable for the neural code intelligence model.

Attention mechanism. The attention mechanism serves as the foundation of our evaluation of AST node types' importance and has become an essential component in many popular attention-based models, such as the well-known Transformer [25] and GAT [26]. It can be defined as a function that maps a query vector and a set of key-value vector pairs to an output vector. The output vector is obtained by taking a weighted sum of the value vectors, where the weight assigned to each value vector is determined by a compatibility function of the query with the corresponding key. Since the discriminative weights reflect the impact of input elements on the outputs of neural models, we utilize the attention weights obtained from each AST node type to assess its importance.

2.2 Related Work

Besides our work, there have been several other studies that aim to understand neural code intelligence models using program simplification. AutoFocus [2] evaluates the impact of a specific code fragment by deleting it from the original program and confirms that the attention weights have a similar effect to code perturbation. WheaCha [28], a method based on reducing and mutating, distinguishes the critical and useless input features in the code method name prediction models. P2IM [17] captures vulnerability signals from models' prediction by applying prediction-preserving input simplification based on delta debugging [30]. Similarly, SIVAND [17] simplifies the program through delta debugging to extract the features that play a key role in the text input of source code for the code intelligence models. SIVAND-Perses [18] further uses Perses [22] to conduct syntax-guided code simplification and extracts the critical source code features to the model while retaining the syntax characteristics. More recently, DietCodeBERT [31] analyzes the importance of different tokens and statements to the code pre-training models through attention weights, and based

on empirical results, it simplifies the source code to accelerate the pre-training models' fine-tuning and testing.

Despite the excellent results of previous work, they still have some incomplete considerations. At first, most of the previous studies only regard the source code as plain text, overlooking the fact that the most significant difference between source code and natural language is the code's structural semantics. Secondly, delta debugging is an iterative algorithm, in which each iteration needs to input the simplified code into the model to obtain the output. Therefore, the efficiency of the delta debugging based approaches such as SIVAND, P2IM, and SIVAND-Perses is very limited, taking a massive amount of time (>30000 hours) to process more than 1 million functions [31], which makes it difficult to apply them on the large-scale code dataset. Besides, DietCodeBERT only considers the key tokens and statements in the CodeSearchNet [12], which makes the generalization ability poor, as performance may decline when applied to other data sets.

3 Empirical Study and Analyse

In order to identify the key code structural semantic features valued by neural code intelligence models, we design an empirical study based on attention weights obtained by different AST node types. In the following subsection, we present our study design and empirical analysis.

3.1 Study Design

In this section, we describe our study methodology and experimental setup. Different from previous works, we concentrate on identifying the critical code structural semantic features valued by neural code intelligence models. Intuitively, each AST node represents a specific source code behavior, such as variable declaration and method invocation, and the edges between AST nodes contains the high-level structural semantics between different grammatical units. Therefore, we evaluate the key structural semantic features by analyzing the AST nodes' attention weights obtained from the attention-based model. Besides, neural code intelligence models often adopt the Encoder-Decoder structure. The encoder and decoder have different functionality: the encoder converts the input in a certain format to a fixed-dimensional intermediate representation, while the decoder generates the model's final predictions based on the intermediate representation. In our research, we separate the encoder and decoder to analyze their possible different behaviors.

We choose a neural intelligence model named SiT [29] as the basic model for research, which takes the AST node sequence as input and incorporates their adjacency relationship into the model. As an exhaustive research, we start with mining the most important AST node types in the encoder and decoder, respectively. Next, we investigate the impact of different AST node types to their different neighbors in the encoder. In a word, we design our empirical study by answering the following research questions:

- **RQ1: What type of AST node does the model pay more attention to when encoding representation and generating predictions?** We explore the critical local structural semantic features that Sit focused on when converting code to intermediate representations in the encoder and generating final predictions in the decoder.

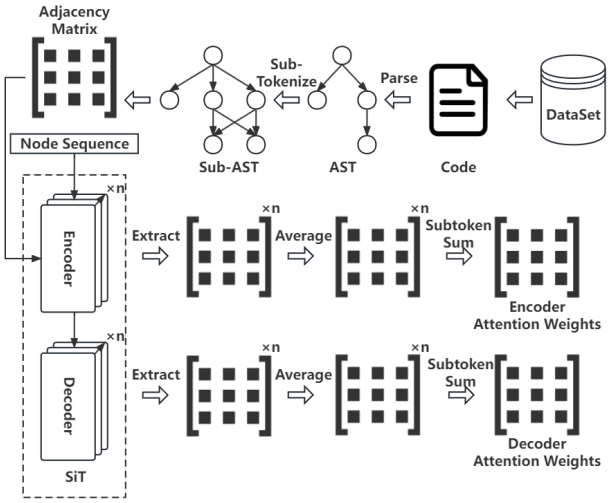


Figure 1: Process of attention weight extraction.

- **RQ2: Is there a dissimilar distribution pattern for the impact of different node types to their neighbors when encoding representations?** We further explore how SiT uses the structure information of AST when encoding representations, and whether different types of AST nodes have different influence patterns on their neighbors.

Following previous studies [2, 31], we use the attention weights of AST nodes to measure their importance. Figure 1 shows the process of attention weights extraction. Specifically, for each code data in the dataset, we parse it into AST and then sub-tokenize each AST node's value to obtain Sub-AST. The motivation behind constructing Sub-AST is to address the out-of-vocabulary challenge, spliting each AST nodes' value that are in form of the `CamelCase` or `snake_case` into sub-tokens. Next, we convert the Sub-AST to its preorder node sequence and adjacency matrix, and then input them into SiT to get the attention weights matrix from the encoder and decoder. We average the attention weights of all layers and heads for each node and sum the attention weights of all Sub-AST nodes to their original AST node.

Considering that there may be multiple nodes of the same type in an AST, instead of simply using the attention weights obtained by each node, we use the sum of the attention scores of the same type of nodes to indicate their importance to the whole AST. Otherwise, this will cause those important node types with high frequency to gain the wrong low importance. For example, suppose there are two method invocation type nodes in an AST, and their attention scores are 0.1 and 0.2, respectively. This means the importance of the method invocation node type should be the total attention score divided by the number of AST with this node, which is $(0.1 + 0.2)/1 = 0.3$, rather than the total attention score divided by the total number of this type of nodes, which is $(0.1 + 0.2)/2 = 0.15$.

We conduct our experiments on two corpora: one Java dataset [11] and one Python dataset [27]. The Java dataset contains 87136 Java code snippets with comments written by the developers and the

Python dataset contains 87226 Python code snippets with comments. Following the default partitioning of the dataset, the proportions of training, validation and testing set are 8:1:1 and 6:2:2, respectively, for the Java and Python dataset, and we only use data from the training set in our empirical analysis. For RQ1, we transform the attention weight matrix into an attention weight sequence by averaging and analyze the node types valued by the encoder and decoder respectively. For RQ2, we compute the average weights of each pair of node types gained from the encoder attention weights matrix and explore the different impact modes of different node types.

3.2 Results and Analyse

In this section, we provide the empirical results and analysis.

RQ1: What type of AST node does SiT pay more attention to in the SiT encoder and decoder? Figure 2 reveals the average attention weights of different types of Java AST nodes in the SiT encoder and decoder. The results show that the encoder and decoder assign more attention to part of the node types such as `MemberReference`, `MethodInvocation`, `ReferenceType`, `ConstructorDeclaration` and `MethodDeclaration`. We speculate that these node types contain richer code semantic information such as method names, variable names, and type names, which is essential for understanding the functionality of the source code. Furthermore, except for `ReturnStatement`, the node types that represent statement types receive lower attention weights in both SiT encoder and decoder, possibly because they are difficult to relate to semantically rich parts of the source code, whereas `ReturnStatement` is usually associated to the method's returned content. This indicates that the SiT model may be not concerned with control flow-related structures, such as loops and conditional branching.

Among all the Java AST node types, `MemberReference` has the highest attention weight in both the encoder and decoder, 2.99e-1 and 2.11e-1, respectively. The node type that obtained the lowest attention weight in the encoder is `Statement`, only 3.11e-4, while in the decoder it is `ContinueStatement`, with a weight of 9.46e-3. The standard deviation of these weights is 5.48e-2 in the encoder and 4.33e-2 in the decoder.

Interestingly, although the SiT encoder and decoder have similar attention weights distribution patterns, the distribution in the encoder is a bit more extreme, which is represented by a larger standard deviation and higher attention weights obtained by rich semantics node types. This is somehow expected, indicating that in the encoder, SiT tends to progressively propagate key information from semantic-rich nodes to others, while in the decoder, SiT prefers to focus on all nodes comprehensively and generate final predictions based on their encoded representations.

Similar conclusions can be drawn from the experimental results of the Python dataset . Both the SiT encoder and decoder focus more on rich-semantics node types such as `Name`, `Attribute`, `FunctionDef` and `Call`, among which `Name` gets the highest average attention weight. In addition, `While`, `For`, `Continue`, `Break`, and other nodes related to the type of statement are assigned very limited attention weights. Compared to results on the Java dataset, the standard deviation of attention weights for Python AST node types is a little higher, which is 6.70e-2 in the encoder and 5.50e-2 in the decoder.

RQ2: Is there a dissimilar distribution pattern for the impact of different node types to their neighbors when encoding representations? Figure 3 shows four examples of the average attention weights obtained by different Java AST node types from their neighbors, including `MemberReference`, `MethodDeclaration`, `TypeArgument` and `VariableDeclaration`. According to these examples, there are indeed dissimilar distribution patterns in the impact of different node types to their neighbors, which can be roughly divided into two categories.

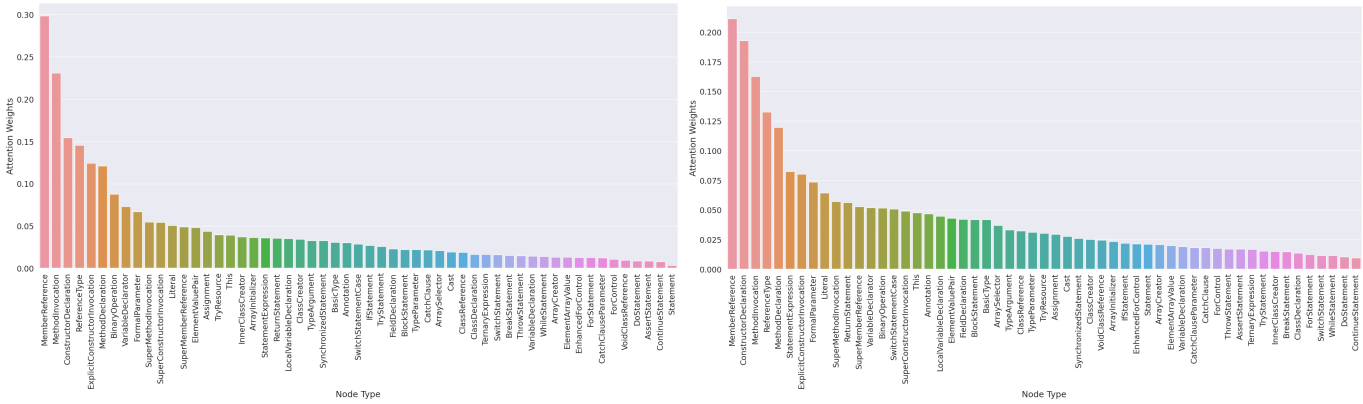
The first category is those node types that are considered to be critical to SiT in RQ1. Although there is a clear difference between the size of attention weights among different neighbor types, these nodes have an obvious influence on all their neighbors. Figure 3a represents the impact of `MemberReference` node type on its neighbors, with a minimum attention weight of 5.1e-2, which is still a large value for some other node types. The node type of `MethodDeclaration` is slightly different. As we can see from Figure 3b, it has more importance to the nodes related to function signatures such as `Annotation`, `FormalParameter` and `TypeParameter`, followed by other nodes, but it still maintains a high attention weight overall.

The second category is those node types that only appear in a specific grammar structure, which only have a high impact to neighbors related to the specific grammar structure. For example, the `TypeArgument` node type is related to generic programming in Java, which is usually associated with a specific type. As shown in Figure 3c, the `TypeArgument` node type has the greatest impact on the `ReferenceType` node type, showing the semantics of parametrical types in generic programming. Similarly, the declaration of a variable is closely linked to the type of the variable and the specific declarator. Besides, the declaration of iterative variables is extremely meaningful to the `for` statement. Therefore, as Figure 3d shows, the `VariableDeclaration` node type has a high impact on `ReferenceType`, `BasicType`, `VariableDeclarator`, `ForControl` and `EnhancedForControl` nodes, while it has little influence on other neighbor nodes.

Similar observations can also be made from the experiments on the Python dataset. Python AST node types such as `Name` and `FunctionDef` can be classified as the first category. These node types obtain the top average attention weights and have a significant impact on all their neighbor nodes. For the second category, the node type `Comprehension` is related to a specific syntax in Python, which provides users with a short and concise way to use predefined sequences to construct new sequences (such as lists, sets, dictionaries, etc.). Python supports 4 types of comprehension: List Comprehensions, Dictionary Comprehensions, Set Comprehensions, and Generator Comprehensions. Thus, it is considered critical by such node types as `ListComp`, `DictComp`, `SetComp`, and `GeneratorExp`, which represent the above 4 types respectively, while the attention weights of other neighbors are quite low.

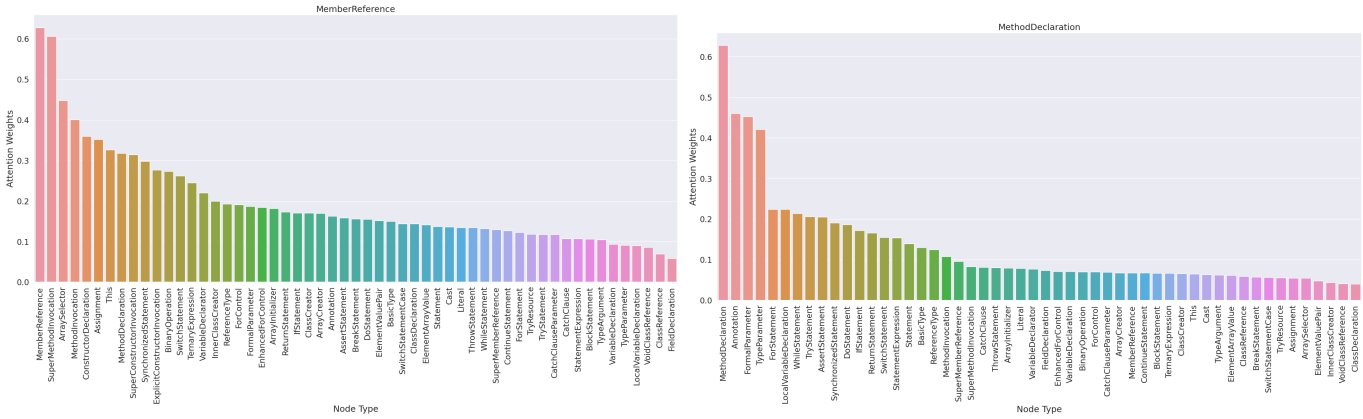
4 Structural-Semantics Guided Program Simplification

From the previous empirical study and analyses, we discover that certain AST node types are unimportant to code intelligence models and a few node types only have a significant impact on

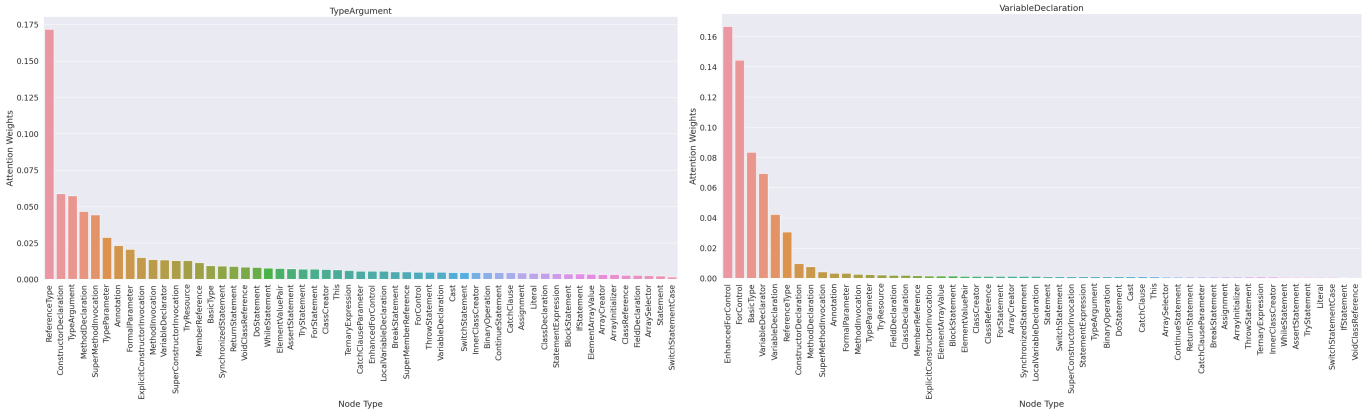


(a) Encoder (b) Decoder

Figure 2: Attention weights of different types of Java AST nodes in SiT encoder and decoder.



(a) MemberReference (b) MethodDeclaration



(c) TypeArgument (d) VariableDeclaration

Figure 3: Examples of the average attention weights of different Java AST node types obtained from their neighbors.

neighbors that belong to the same semantic behavior, which proves the existence of key code structural semantic features. By selectively pruning nodes while preserving these key features, we can effectively reduce the input size of code intelligence models, leading to improved efficiency. While a straightforward approach to node pruning involves gradually removing nodes with lower attention weights until the entire AST reaches a desired scale, this strategy proves to be inefficient. Furthermore, there is a risk that such a simplistic approach may result in the pruning of the AST to a mere collection of method and variable names, thereby compromising the structural semantics that the AST inherently possesses.

Considering that certain node types that only occur within specific syntax structures have a significant impact on other node types within the same structure, it becomes apparent that if the code intelligence model does not value the semantics of these specific structures, the nodes within them can be deemed unimportant. Hence, we propose a two-stage node pruning algorithm. In the first stage, we adopt the statement as the granularity and perform coarse-grained pruning. The aim is to eliminate syntax structures that are not valued by the code intelligence model. Subsequently, in the second stage, we proceed with fine-grained pruning using the node as the granularity. Here, we selectively remove nodes that are not valued by the code intelligence model. To preserve the key code structural semantic features, we introduce SSGPS, a Structural-Semantics Guided Program Simplification method that leverages the insights mentioned above. SSGPS employs attention weights to compute the importance of AST nodes, enabling the pruning of redundant nodes and retaining the critical code structural semantic features.

4.1 Attention Information Extracting

Similar to the empirical study in Section 3, we use the attention matrix in the SiT encoder and decoder to compute node importance. Specifically, as shown in Figure 1, we extract the attention weights matrix from the encoder and decoder to generate *EMap* and *DMap*. The input to *EMap* is a pair of node types (u, v), and the output from *EMap* tells the importance of node type u to another node type v. To compute *EMap*, we extract the attention weights of each pair of nodes from the encoder, record each node types pair's attention weights and occurrences, and eventually compute the average attention weights of each pair of node types after processing all data. On the other hand, the input to *DMap* is a single node type u, and *DMap* outputs the contribution of node type u to the model's final prediction. Different from the calculation of *EMap*, for *DMap* we reduce the dimension of the decoder's attention matrix to obtain the average attention weight of each node in input AST and then calculate the *DMap* in a similar way to *EMap*. To make full use of the attention information of multiple layers and heads, we average the attention weights of all layers and heads for each node.

4.2 Node Importance

When calculating the importance of an AST node, we need to comprehensively consider the role of the node in encoding intermediate representations and predicting final results. In addition, if a node only has a significant impact on some of its neighbors, and these neighbor nodes are not valued, then the importance of the node should be lowered correspondingly. Given that *EMap* and *DMap* have been obtained to denote the importance of each node type to others types and the importance of each node type when making a final prediction, the importance of a node can be calculated as

$$Importance(u) = DMap(Type(u)) + \frac{1}{\mid Neigh(u) \mid} \sum_{v \in Neigh(u)} EMap(Type(u), Type(v)) \quad (1)$$

where $Importance(u)$ means the importance of node u, $Type(u)$ denotes the type of node u and $Neigh(u)$ represents the neighbor set of node u in the AST. Equation (1) allows us to determine a node's importance by combining its own importance and its influence on other nodes. Note that there might be other ways to achieve the same objective, we choose this simple formulation which is already effective as demonstrated by experiments in Section 5.

4.3 Pruning Algorithm

Using Equation (1) and the *EMap* and *DMap* obtained from Section 4.1, given any AST node and its adjacency relationship, we can directly calculate its importance to the code intelligence model. We use L as the maximum number of nodes contained in the reduced AST. Intuitively, we can prune the most unimportant node one by one until the number of remaining nodes in the AST equals L. However, it is significantly time-consuming since each deletion of a node requires recalculating the importance of all remaining nodes. Therefore, to balance the effectiveness and efficiency of program reduction, we implement a two-stage pruning algorithm containing coarse-grained pruning and fine-grained pruning.

Algorithm 1 summarizes the whole two-stage program reduction procedure. In the first stage, we use the statement as the granularity for coarse-grain pruning, while in the fine-grain pruning stage, we use the single AST node. Before the AST level program reduction, we extract the attention weight information from the encoder and decoder of SiT and convert it into *EMap* and *DMap*, respectively.

In the coarse-grained pruning stage, we begin the reducing procedure with extracting a statement based sub-graph from a given AST, where each node corresponds to a statement-type node and several other types of nodes in the original AST. We use a map to maintain the relationship between sub-graph nodes and original AST nodes, by which the sub-graph can be easily restored to AST after coarse-grain pruning. For the importance of each sub-graph node, we use the sum of the importance of all associated AST nodes as its importance. Then, we greedily delete the sub-graph nodes that obtain the lowest importance until the current deletion will cause the current number of nodes to be less than the predefined maximum number L. It is worth emphasizing that every deletion of the sub-graph node will lead to structural changes in the sub-graph and the corresponding AST. Therefore, it is quite necessary to update the adjacency matrix and recalculate the importance of the remaining nodes after each deletion.

In the fine-grained pruning phase, we first rebuild the whole AST from the pruned sub-graph, and then remove the most unimportant AST nodes one by one until satisfying the maximum number of nodes. Similar to the coarse-grained pruning phase, we also update the adjacency matrix and recalculate the importance of the left AST nodes after each removal.

Algorithm 1 The Two-stage node pruning algorithm

Require:
The original AST, $T = (V, E)$,where
$V = \{n_1; n_2; ...; n_{|V|}\}$ and $E = \{(n_u, n_v) \mid n_u, n_v \in V\}$;
Attention map of a pair of AST node types, $EMap$;
Attention map of a single AST node type, $DMap$;
The max number of pruned AST nodes, L;
Ensure:
The pruned AST, $T_p = (V_p, E_p)$ where
$V_p = \{n_1^p; n_2^p; ...; n_{V_p}^p\}$ and
$E_p = \{(n_u^p, n_v^p) \mid n_u^p, n_v^p \in V_p\}$;

Extract a statement based subgraph T_s from T,
$T_s = (V_s, E_s)$ where $V_s = \{n_1^s, n_2^s, ..., n_{|V_s|}^s\}$ and
$E_s = \{(n_u^s, n_v^s) \mid n_u^s, n_u^s \in V_s\}$
Maintain a relationship map *SubgraphMap*, whose key is node $n^s \in V_s$ of T_s and value is the set of nodes in V related to n^s
Current number of nodes $L_c \longleftarrow \mid V \mid$
while $L_c > L$ **do**
  // coarse grain prune
  for $n^s \in V_s$ **do**
    use (1) to compute the importance of n^s,
    $Importance(n^s) = \sum_{n^* \in SubgraphMap(n^s)} I(n^*)$
  end for
  $n^{s'} = argmin_{n^s \in V_s} Importance(n^s)$
  if $L_c - \mid SubgraphMap(n^{s'}) \mid \leq L$ **then**
    break;
  else
    $V_s \leftarrow V_s \setminus \{n^{s'}\}$
    $V \leftarrow V \setminus SubgraphMap(n^{s'})$
    update E_s and E
    $L_c \leftarrow L_c - \mid SubgraphMap(n^{s'}) \mid$
  end if
end while
Rebuild $T_p = (V_p, E_p)$ from pruned subgraph T_s
while $L_c > L$ **do**
  // fine grain prune
  for $n^p \in V_p$ **do**
    use Equation (1) to compute importance of n^p,
    $Importance(n^p) = I(n^p)$
  end for
  $n^{p\prime} = argmin_{n^p \in V_p} Importance(n^p)$
  $V_p \leftarrow V_p \setminus \{n^{p\prime}\}$
  update E_p
  $L_c \leftarrow L_c - 1$
end while
$T_p = (V_p, E_p)$
return T_p;

5 Experiments

In this section, we first introduce the setup of our experiments, including datasets, metrics, baselines, and research questions. Then we present the experimental results and detailed analysis.

5.1 Experiments Setup

Datasets. We select two widely used code summarization benchmarks, one in Java [11] and the other in Python [27]. Both have been used for our empirical study in Section 3. We follow the train/valid/test partition of the original data set and replace the numeric constants, character or string constants, and Boolean constants in the code with _NUM_, _STR_, and _BOOL_. Then, we use the `javalang` and `ast` packages to convert the Java and Python source code to the corresponding AST, respectively. Besides, to solve the out-of-vocabulary problem, we split the source code and the leaf nodes in ASTs into sub-tokens that are in form of the `CamelCase` or `snake_case`. Table 1 shows the statistics of the two datasets.

Table 1: Statistics of the datasets.

Dataset	Split	Original	Cleaned
JCSD	Train	69,708	69,708
	Valid	8,714	8,714
	Test	8,714	8,714
PCSD	Train	55,538	55,532
	Valid	18,505	18,503
	Test	18,502	18,500

Metrics. To evaluate our method's effectiveness, we use two metrics. The first one is *Relative Size* (RS), which shows how much of AST nodes is remained after structural program simplification. It is calculated from the percentage of the ratio of the reduced number of AST nodes to the original number of AST nodes. The smaller the relative size, the greater the simplification of the original program. The other metric is *time cost*, including training time (fine-tuning time in RQ2) and testing time. We record the time of training from scratch, fine-tuning, and testing time of different baseline models under different experimental conditions in seconds.

On the other hand, we measure the performance of different code summarization models under different settings using the following widely used metrics: *BLEU* [16], *ROUGE-L* [14] and *METEOR* [1]. BLEU is a corpus-level metric and it calculates constituent n-grams precision scores with a penalty for short sequences. ROUGE-L is a widely used metric in text summarization evaluation and it computes F-score using the Longest Common Subsequences (LCS). METEOR uses the harmonic average of precision and recall to measure the quality of the generated summaries. All the scores are presented in percentages and the higher the scores, the better the performance of the model.

Baselines. In our experiments, for the code summarization models, We first choose SiT as the baseline for the reason that the model is used for key feature extraction by us. Additionally, to demonstrate the generalization of key features extracted through the SiT model, we also select two code structural representations powered baseline models, SCRIPT and AST-Trans:

- SiT [29]: It extends the Transformer architecture with the structure-induced self-attention mechanism, which provides the model with the ability to extract structural features.

- SCRIPT [8]: It enables the model to learn the structural relative correlations between AST nodes by encoding the relative position information of each node in AST, which is beneficial for code semantic learning.
- AST-Trans [24]: It applies the tree-structured attention to dynamically assign weights for relevant nodes and propose an efficient and parallelizable implementation.

For all the above models, we use their open-source implementation and maintain the values for all the hyperparameters.

For the program simplification baseline methods, we select DietCodeBERT [31] which aims at lightweight leverage of large pre-training models for source code such as CodeBERT [5] by dropping the unimportant statements and tokens. Methods based on delta debugging are inefficient, taking over 30,000 hours to process more than one million functions [31], so we don't choose these methods as the baselines.

We run all models on a machine with a CPU of Intel(R) Xeon(R) Silver 4214R 2.40GHz and a GPU of Nvidia Tesla P40.

Research questions. To evaluate the effectiveness of SSGPS, we aim to answer the following research questions through intensive experiments:

- **RQ3: How effective is SSGPS in program simplification?** We evaluate the performance and time cost of SSGPS on three code summarization models and compare them with baseline models. Meanwhile, we compare the experimental results of SSGPS and DietCodeBERT and use an example to show the superiority of SSGPS.
- **RQ4: How effective is SSGPS in the fine-tuning condition?** By answering this question, we try to evaluate whether SSGPS can use the trained model to avoid training from scratch.
- **RQ5: How effective is SSGPS under different relative sizes (RS)?** We study the effect of different relative sizes on the evaluation in order to discover the proper relative size leading to the best balance of model performance and efficiency.
- **RQ6: What the effect of different information is in SSGPS?** In SSGPS, we comprehensively study the information of the encoder and decoder. We conduct an ablation study to study the effect of encoder information and decoder information.

5.2 Experimental Results

RQ3: How effective is SSGPS in program simplification? Table 2 presents the experimental results of various baseline methods on the code summarization task. They can be divided into 4 groups based on the different experimental settings, with the first three groups being used for answering RQ3. The first group includes the time costs and evaluation scores of three baseline code summarization models. In the second group, we show the influence of SSGPS on the three baseline models and finally, the experimental results of CodeBERT and DietCodeBERT on JCSD and PCSD are presented in the third group.

The results in Table 2 show that SSGPS is highly effective in code summarization models. It can reduce training and testing time by 20% to 40% while maintaining model evaluation scores within a 4% decline. On SiT and SCRIPT, SSGPS can save training and testing time costs without sacrificing model performance or even slightly improving it. For instance, on JCSD, the original SCRIPT model achieves a BLEU score of 46.01, a Rouge score of 55.97, and a Meteor score of 27.67 at the expense of 110,266 seconds for training and 302.33 seconds for testing. When using SSGPS to limit the number of input AST nodes to 128, the training time and testing time are reduced to 72,931 seconds and 215.03 seconds, respectively, with a comparable level on all evaluation scores to the original model. The impact of SSGPS on AST-Trans is slightly higher, with BLEU, Rouge, and Meteor decreasing by 2.1%, 2.2%, and 2.8% in JCSD and 2.0%, 3.8%, and 4.0% in PCSD.

In comparison to the existing program simplification method DietCodeBERT, SSGPS exhibits better stability. In our experiments on JCSD and PCSD, DietCodeBERT reduces fine-tuning time by 28.4% and 32.9% and testing time by 18.0% and 19.9%, separately. Meanwhile, the evaluation scores' declines of BLEU, Rouge, and Meteor on JCSD are 3%, 2.9%, and 4.0%, which is a passable level for users. However, on PCSD, the damage of DietCodeBERT on CodeBERT's performance is unacceptable, reaching 24%, 12%, and 25% on BLEU, Rouge, and Meteor, respectively. As for SSGPS, the largest decrease in evaluation scores occurs on the Meteor score of AST-Trans (4%), which is still within the tolerable range.

Figure 4 provides an example of program simplification using DietCodeBERT and SSGPS. The purpose of the original source code (Figure 4a) is to read content from a file with the given filename. The result of DietCodeBERT (Figure 4b) remains the code's core functionality, but there still exists redundant information that is not related to the code functionality, such as `try` and `finally`. Because SSGPS does not consider syntax correctness when performing structural pruning at the AST level, the pruned AST cannot be restored to the original code, but we can still analyze its effect from the remaining AST's node values and overall structure. Figure 4c shows that SSGPS successfully removes redundant nodes that are semantically independent while retaining nodes that are related to concrete semantics.

RQ4: How effective is SSGPS in the fine-tuning condition? Table 2 also provides the experimental results of applying SSGPS to SiT and SCRIPT under fine-tuning settings in the 4th group. Overall, comparable performance to the model trained from scratch can be achieved in 20% of the training time by fine-tuning the trained model using the dataset processed by SSGPS. On JCSD, the effectiveness of SSGPS is very significant, which can reduce the testing time of SiT and SCRIPT by 27% and 27.5%, while slightly improving BLEU, Rouge, and Meteor scores. On the other hand, on PCSD, the BLEU, Rouge, and Meteor scores of SiT and SCRIPT descend negligibly to varying degrees, ranging from 0.4% to 2.6%.

RQ5: How effective is SSGPS under different relative sizes? Figure 5 shows the BLEU score of SiT and AST-Trans under different relative sizes. It can be observed that the performance of SiT remains stable, even when the relative size drops to 0.25, compared to the original model. This may be because SSGPS utilizes attention weights information extracted from SiT, which enables it to effectively remove redundant structural features. As for AST-Trans, when the relative size is above 0.625, the BLEU score is comparable to that of the original AST-Trans. However, when the relative size drops below 0.5, the BLEU score decreases significantly. Therefore, a relative size of around 0.5 may be the best trade-off between model performance and efficiency for all baseline code summarization models.

Table 2: Experimental results of various baseline methods on the code summarization task (S_{in}: input size; RS: relative size; FT: fine-tuning).

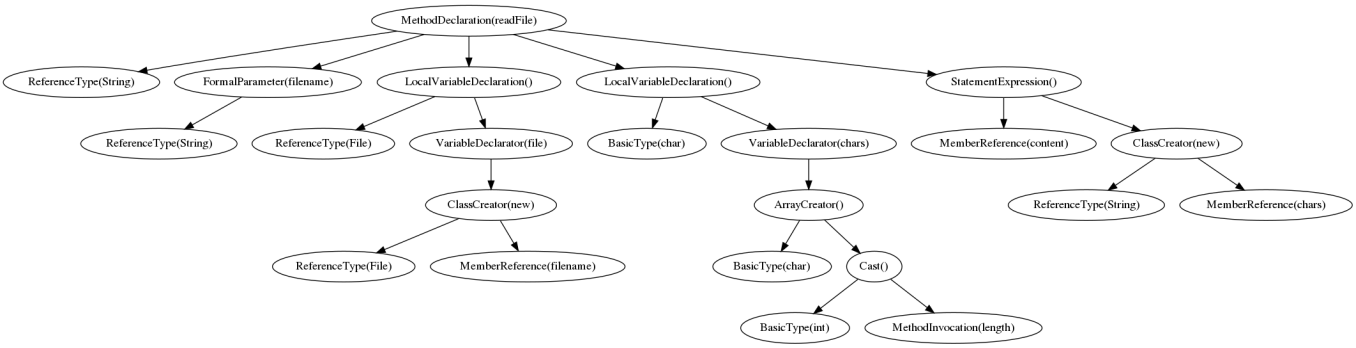
Model	JCSD							PCSD						
	S_{in}	RS	Train/FT Time	Test Time	BLEU	Rouge	Meteor	S_{in}	RL	Train/FT	Test Time	BLEU	Rouge	Meteor
SiT	256	100%	58393s	385.46s	45.17	55.21	26.83	256	100%	58011s	413.40s	36.53	49.93	21.80
AST-Trans	256	100%	107298s	154.28s	46.34	53.63	29.28	256	100%	112277s	343.91s	33.96	40.53	19.90
SCRIPT	256	100%	110266s	302.33s	46.01	55.97	27.67	256	100%	90163s	434.02s	36.63	49.96	21.93
SiT (reduced)	128	50%	44658s(-23.5%)	288.46s(-25.2%)	45.33 (+0.3%)	55.40(+0.3%)	26.91(+0.3%)	128	50%	40125s(-30.8%)	329.42s(-20.3%)	36.53(+0%)	49.93(+0%)	21.68(-0.5%)
AST-Trans (reduced)	128	50%	68843s(-30.5%)	117.03s(-24.1%)	45.35(-2.1%)	52.47(-2.2%)	28.46(-2.8%)	128	50%	67577s(-39.8%)	250.21s(-27.2%)	33.29(-2.0%)	38.99(-3.8%)	19.10(-4.0%)
SCRIPT (reduced)	128	50%	72931s(-33.9%)	215.03s(-29.9%)	46.50(+1.1%)	56.27(+0.4%)	28.35(+2.3%)	128	50%	62443s(-30.7%)	344.22s(-20.7%)	36.63(+0%)	49.83(-0.3%)	21.93(+0%)
CodeBERT	256	100%	83080s	2516s	44.63	55.94	26.53	256	100%	74083s	3653s	36.53	50.82	22.37
DietCodeBERT	128	50%	59518s(-28.4%)	2064s(-18.0%)	43.29(-3.0%)	54.31(-2.9%)	25.47(-4.0%)	128	50%	49762s(-32.9%)	2926s(-19.9%)	27.77(-24.0%)	44.53(-12.4%)	16.78(-25.0%)
SiT (fine-tuned)	128	50%	10230s(-82.5%)	281.34s(-27.0%)	45.23(+0.1%)	55.36(+0.5%)	27.09(+2.1%)	128	50%	11711s(-80.0%)	326.38s(-21.0%)	36.38(-0.4%)	49.70(-0.5%)	21.67(-0.6%)
SCRIPT (fine-tuned)	128	50%	15224s(-86.2%)	219.29s(-27.5%)	46.24(+0.5%)	56.20(+0.5%)	28.10(+1.6%)	128	50%	17749s(-80.3%)	344.65s(-20.6%)	36.15(-1.0%)	49.22(-1.4%)	21.37(-2.6%)

```java
public String readFile (String filename) {
    String content = null;
    File file = new File(filename);
    FileReader reader = null;
    try {
        reader = new FileReader(file);
        char[] chars = new char[(int)file.length()];
        reader.read(chars);
        content = new String(chars);
        reader.close();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(reader != null){
            reader.close();
        }
    }
    return content;
}
```

(a) The original source code [31]

```java
public String readFile (String filename) {
    File file = new File(filename);
    FileReader
    try {
        reader = new FileReader(file);
        reader.read(chars);
        content = new String(chars);
        reader.close();
    } finally {}
    return content;
}
```

(b) simplified by DietCodeBERT [31]



(c) simplified by SSGPS

Figure 4: An example of program simplification by DietCodeBERT and SSGPS

RQ6: What the effect of different information is in SSGPS? Table 3 presents the ablation results of SiT and AST-Trans with only encoder information or decoder information on JCSD and PCSD datasets. The purpose is to investigate the role of encoder and decoder information in the importance evaluation of AST nodes. Specifically, we only utilize encoder information and decoder information separately to calculate AST node importance in Equation 1 and analyze their impact on the performance of different models.

The experimental results show that when only encoder information or decoder information is used, SSGPS achieves satisfactory enough results on different models in different datasets, but the results are still lower than those obtained by a comprehensive consideration of both encoder and decoder information. This indicates that both encoder and decoder information is important, and it is valuable to consider them comprehensively when evaluating the importance of AST nodes.

One possible explanation for this is that the encoder and decoder have different functionalities. The encoder tends to propagate information from semantically rich nodes to others, while the decoder focuses on all nodes to generate final predictions based on their

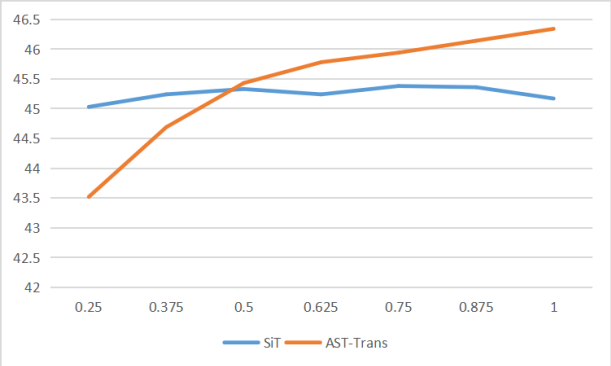


Figure 5: BLEU score of SiT and AST-Trans under different relative sizes on JCSD

Table 3: Ablation results of SiT and AST-Trans with only encoder information or decoder information on JCSD and PCSD

Model	Type	Dataset	BLEU	Rouge	Meteor
SiT	All	JCSD	45.33	55.40	26.91
SiT	Encoder Only	JCSD	45.12	55.17	26.88
SiT	Decoder Only	JCSD	45.11	55.18	26.85
SiT	All	PCSD	36.53	49.93	21.68
SiT	Encoder Only	PCSD	36.51	49.82	21.60
SiT	Decoder Only	PCSD	36.47	49.84	21.65
AST-Trans	All	JCSD	45.35	52.47	28.46
AST-Trans	Encoder Only	JCSD	45.24	52.35	28.43
AST-Trans	Decoder Only	JCSD	45.25	55.18	28.29
AST-Trans	All	PCSD	33.29	38.99	19.10
AST-Trans	Encoder Only	PCSD	32.90	37.95	18.51
AST-Trans	Decoder Only	PCSD	32.93	38.30	18.71

encoded representations. Therefore, the comprehensive consideration of both encoder and decoder information can more thoroughly capture the structural semantics and effectively remove redundant structural features.

6 Threats to Validity

Although the experimental results in Section 5 demonstrate the superior effectiveness of SSGPS, we identify the following three primary threats to the validity of our research.

- The first threat is related to the datasets used in this paper. Our experiments are conducted only in Java and Python, and although we have obtained similar results for both languages, other programming languages like C++ and PHP may have different structural semantics and attention weight distribution patterns. Further evaluations involving more diverse programming languages are necessary for future research.
- Secondly, we must acknowledge that we have only evaluated our approach on the code summarization task, which presents a potential threat to the validity of our research. While our experimental results demonstrate that our method can effectively remove redundant code structural semantic features in this task, it is essential to verify its applicability to other code intelligence tasks through additional experiments.
- In addition, the third threat to validity lies in the syntactic correctness of structural code reduction. In our approach, we mainly focus on preserving key structural semantics and removing redundant features, without paying much attention to the syntax correctness of the simplified code. This could introduce bias in our datasets and affect the results of our research.

7 Conclusion

This paper has presented an empirical analysis of the crucial structural semantic features valued by code intelligence models in Java and Python. Our findings suggested that code intelligence models prioritize semantic-rich AST node types, such as those related to function names, variable names, and type names. Furthermore, we observed that different AST node types exhibited different patterns of impact distribution on their neighboring nodes. Some node types corresponding with specific syntax structures only significantly affected neighbors that appear within the same syntax structure. Based on our empirical findings, we have proposed SSGPS (Structural-Semantics Guided Program Simplification), a two-stage program simplification algorithm that performs coarse-grained and fine-grained structural program reduction at the AST level to eliminate structural semantic information which is irrelevant to the models. Our experiments on three code summarization models showed that SSGPS can reduce training and testing time costs by 20% to 40% while maintaining the model's performance drop to less than 4%, demonstrating its effectiveness.

For future work, we plan to explore more node importance calculation strategies for the original AST node importance calculation and the statement-based subgraph node importance calculation. By incorporating these methods, we can develop more effective program simplification algorithms. Additionally, we intend to investigate a more automated approach by designing the model structure to enable the identification and removal of redundant structural semantic features through fine-tuning. As a continuation of our current research, we will conduct evaluations of the performance of SSGPS on other programming languages such as C++ and PHP. We will also extend our approach to other software engineering tasks, such as program classification, to further improve the understanding and interpretability of code intelligence models.

References

[1] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. ACL, 65–72.

[2] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: Interpreting Attention-based Neural Networks by Code Perturbation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 38–41.

[3] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 964–974.

[4] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*. ACL, 1536–1547.

[6] Stephen R Foster, William G Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE Support for Real-time Auto-completion of Refactorings. In *Proceedings of the 34th International Conference on Software Engineering*. ACM, 222–232.

[7] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code Structure–Guided Transformer for Source Code Summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[8] Zi Gong, Cuiyun Gao, Yasheng Wang, Wenchao Gu, Yun Peng, and Zenglin Xu. 2022. Source Code Summarization with Structural Relative Position Guided Transformer. In *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 13–24.

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of the 9th International Conference on Learning Representations*. http://OpenReview.net.

[10] Shin Hong and Moonzoo Kim. 2013. Effective Pattern-driven Concurrency Bug Detection for Operating Systems. *Journal of Systems and Software* 86, 2 (2013), 377–388.

[11] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. ACM, 2269–2275.

[12] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet Challenge: Evaluating the State of Semantic Code Search. *ArXiv Preprint ArXiv:1909.09436* (2019).

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[14] Chin-Yew Lin. 2004. Rouge: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. ACL, 74–81.

[15] W James Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. 2019. Definitions, Methods, and Applications in Interpretable Machine Learning. *Proceedings of the National Academy of Sciences* 116, 44 (2019), 22071–22080.

[16] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. ACL, 311–318.

[17] Md Rafiqul Islam Rabin, Vincent J Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding Neural code Intelligence through Program Simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 441–452.

[18] Md Rafiqul Islam Rabin, Aftab Hussain, and Mohammad Amin Alipour. 2022. Syntax-guided program reduction for understanding neural code intelligence models. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. ACM, 70–79.

[19] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. 2022. On the Evaluation of Neural Code Summarization. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 1597–1608.

[20] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 43–52.

[21] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically Detecting and Describing High Level Actions within Methods. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 101–110.

[22] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 361–371.

[23] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim A Laredo, and Alessandro Morari. 2021. Probing Model Signal-awareness via Prediction-preserving Input Minimization. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 945–955.

[24] Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, Zhelin Zhu, and Bin Luo. 2022. AST-Trans: Code Summarization with Efficient Tree-Structured Attention. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 150–162.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Proceedings of the 31st Annual Conference on Neural Information Processing Systems*. Mit Press.

[26] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceddings of the 6th International Conference on Learning Representations*. http://OpenReview.net.

[27] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 397–407.

[28] Yu Wang, Ke Wang, and Linzhang Wang. 2021. WheaCha: A Method for Explaining the Predictions of Code Summarization Models. *ArXiv Preprint ArXiv:2102.04625* (2021).

[29] Hongqiu Wu, Hai Zhao, and Min Zhang. 2021. Code Summarization with Structure-induced Transformer. In *The Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*. ACL, 1078–1090.

[30] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[31] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet Code is Healthy: Simplifying Programs for Pre-trained models of Code. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1073–1084.