AST-Trans: Code Summarization with Efficient Tree-Structured Attention

Ze Tang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
2228291607@qq.com

Xiaoyu Shen*
Alexa AI
Amazon
Berlin, Germany
gyouu@amazon.com

Chuanyi Li, Jidong Ge
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
lcy,gjd@nju.edu.cn

Liguo Huang
Department of Computer Science
Southern Methodist University
Dallas, Texas, USA
lghuang@lyle.smu.edu

Zhelin Zhu, Bin Luo
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
zzl,luobin@nju.edu.cn

ABSTRACT

Code summarization aims to generate brief natural language descriptions for source codes. The state-of-the-art approaches follow a transformer-based encoder-decoder architecture. As the source code is highly structured and follows strict grammars, its Abstract Syntax Tree (AST) is widely used for encoding structural information. However, ASTs are much longer than the corresponding source code. Existing approaches ignore the size constraint and simply feed the whole linearized AST into the encoders. We argue that such a simple process makes it difficult to extract the truly useful dependency relations from the overlong input sequence. It also incurs significant computational overhead since each node needs to apply self-attention to all other nodes in the AST. To encode the AST more effectively and efficiently, we propose AST-Trans in this paper which exploits two types of node relationships in the AST: ancestor-descendant and sibling relationships. It applies the tree-structured attention to dynamically allocate weights for relevant nodes and exclude irrelevant nodes based on these two relationships. We further propose an efficient implementation to support fast parallel computation for tree-structure attention. On the two code summarization datasets, experimental results show that AST-Trans significantly outperforms the state-of-the-arts while being times more efficient than standard transformers [1].

*Work done before joining.
[1]All the codes and data are available at https://github.com/zetang94/ICSE2022_AST_Trans.git

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510224

CCS CONCEPTS

• **Software and its engineering → Documentation**; • **Computing methodologies → Natural language generation**.

KEYWORDS

tree-based neural network, source code summarization

ACM Reference Format:
Ze Tang, Xiaoyu Shen, Chuanyi Li, Jidong Ge, Liguo Huang, and Zhelin Zhu, Bin Luo. 2022. AST-Trans: Code Summarization with Efficient Tree-Structured Attention. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510224

1 INTRODUCTION

The summary of source code is a brief natural language description explaining the purpose of the code [29]. The code to be summarized can be with different units. In this work, we focus on summarizing the subroutines or defined methods in a program.

Previous studies have shown that such a short description can assist program developers to quickly digest the code without traversing over it themselves [43]. Nonetheless, maintaining high-quality code summaries requires expensive manual labor in reality. In many projects, these summaries are often mismatched, missing or outdated which slow down the developing progress [18]. Automatic code summarization can greatly save developers' time by avoiding writing such summaries manually for every single code snippet.

The traditional methods utilized handcrafted rules like Software Word-Usage Model (SWUM) [43] or stereotypes [30] to synthesize the code summaries. However, when identifiers or methods are poorly named, they cannot extract accurate keywords to produce good summaries. Some used Information Retrieval (IR) techniques [13, 14] to mine summaries from similar existing code banks which, unfortunately, cannot generalize to unseen code snippets with different functions.

Recently, with the development of open source platforms such as Github, more and more data for code summarization can be easily extracted from online resources. Data-driven strategies based on

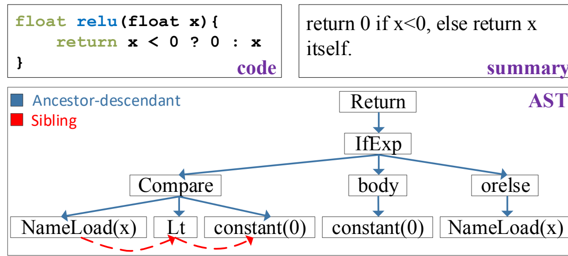

Figure 1: Example of code-AST-summary triples. We mainly need to understand the ancestor-descendent and sibling relationships in the AST to generate a summary.

neural networks start to raise more and more attention [20, 37–39, 56]. Current state-of-the-arts all follow the Transformer-based encoder-decoder architecture [5, 8, 45, 48, 49] and can be trained end-to-end with code-summary pairs. Since the source code is highly structured and follows strict programming language grammars, a common practice is to also leverage the Abstract Syntax Tree (AST) to help the encoder digest the structured information. The AST is usually linearized by different algorithms like pre-order traversal [21], structure-based traversal (SBT) [18] and path decomposition [4], then fed into the encoder. Several works also proposed architectures specific for tree encoding like tree-LSTM [11, 51].

However, the linearized ASTs, as containing additional structured information, are much longer than their corresponding source code sequence. Some linearization algorithms can further increase the length. For example, linearizing with SBT usually makes the size times longer. This makes the model extremely difficult to accurately detect useful dependency relations from the overlong input sequence [2]. Moreover, it brings significant computational overhead, especially for state-of-the-art Transformer-based models where the number of self-attention operations grows quadratically with the sequence length. Encoding ASTs with tree-based models like tree-LSTM will incur extra complexity because it needs to traverse the whole tree to obtain the state of each node.

In this work, we assume that the state of a node in the AST is affected most by its (1) ancestor-descendent nodes, which represent the hierarchical relationship across different blocks, and (2) sibling nodes, which represent the temporal relationship within one block. We show an example of code summarization in Figure 1. As can be seen, we need the ancestor-descendent relationship to understand the high-level procedure, and the sibling relationship to understand the low-level details within a block. Capturing these two relationships are enough for producing the summary and modelling the full attention among all nodes is unnecessary.

Based on this intuition, we propose AST-Trans, a simple variant of the Transformer model to efficiently handle the tree-structured AST. AST-Trans exploits ancestor-descendant and sibling relationship matrices to represent the tree-structure, and uses these matrices to dynamically exclude irrelevant nodes in different self-attention layers. The absolute position embedding from the original Transformer is replaced with relative position embeddings defined by the two relationship matrices to better model the dependency. We further describe several implementations of the proposed AST-Trans and have a comprehensive analysis of their computational complexity. In short, the contributions of this paper are as below:

- We propose AST-Trans that can efficiently encode long AST sequences with linear complexity, in contrast with the quadratic complexity of the standard Transformer.
- We perform a comprehensive analysis, with both theoretical and empirical evidences, on the computational complexity of different implementations.
- We validate our proposed model on two datasets of Java and Python. Experimental results show that AST-Trans outperforms the state-of-the-arts by a substantial margin.
- We compare representative methods for AST encoding and discuss their pros and cons.

Paper Organization The remainder of this paper is organized as follows. Section 2 presents background knowledge on the Transformer and AST. Section 3 elaborates on the details of AST-Trans, section 4 presents its different implementation and the complexity is analyzed in section 5. Section 6 explains the experimental setup and analyzes the results. Section 7 discusses threats to validity. Section 8 surveys the related work. Finally, section 9 concludes the paper and points out future research directions.

2 BACKGROUND

Transformer. The Transformer architecture was initially proposed for neural machine translation [49]. It consists of multi-head stacked encoder and decoder layers. In each encoder stack, the inputs first flow through a self-attention sublayer, and then are fed into a position-wise feed-forward network followed by a layer normalization. The decoder has a set of the cross-attention layers to help the decoder focus on relevant parts of the input sequence. The Transformer architecture removes the recurrence mechanism in favor of the self-attention. As each word in a sentence simultaneously flows through the encoder and decoder stack, the model itself does not have any sense of the word order. Therefore, a position embedding is added to each word embedding to inform the order information.

Abstract Syntax Tree (AST). An Abstract Syntax Tree (AST) uniquely represents a source code snippet in a given language and grammar [4]. The leaves of the tree are terminals, usually referring to variables, types and method names. The non-leaf nodes are non-terminals and represent a restricted set of structures in the programming language, e.g., loops, expressions, and variable declarations. For example, in Figure 1, variables (such as *NameLoad(x)*) are represented as terminals of AST. Syntactic structures (such as *Compare*) are represented as non-terminals. Since the variable and method names can be rather freely defined, directly processing the source code can be challenging. Its corresponding AST, due to its strict structure, often serves as substitute when encoding the source code.

3 AST-TRANS

This section details our proposed AST-Trans. For an AST, it will be firstly linearized into a sequence. Then the ancestor-descendent and sibling relationships among its nodes will be denoted through

[2]Indeed, encoding the overlong AST with SBT even underperforms directly encoding the source code when using Transformer with relative position embeddings [1].

Table 1: Linearized AST of the tree in Fig 1 with POT,SBT and PD.

Methods	Linearized AST sequence
POT	Return IfExp Compare NameLoad(x) Lt constant(0) body constant(0) orelse NameLoad(x)
SBT	(Return (IfExp (Compare (constant(0)) constant(0) (Lt) Lt (NameLoad(x)) NameLoad(x)) Compare (body (constant(0)) constant(0)) body (orelse (NameLoad(x)) NameLoad(x)) orelse) IfExp) Return
PD	Path1: Path1: Lt Compare constant(0) Path2: NameLoad(x) Compare constant(0) Path3: Path3: constant(0) Compare IfExp body constant(0) ...

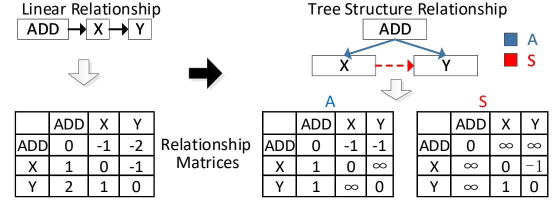

Figure 2: Example of generating position matrices for ancestor-descendent (A) and sibling relationship (S). Position matrix generated from the linear relationship is used in standard Transformers.

two specific matrices. Based on the matrices, we replace the standard self-attention with tree-structured attention to better model these two relationships. Irrelevant nodes are dynamically ruled out to reduce computational cost. We will first introduce different linearization methods (section 3.1), then explain the construction of two relationship matrices (section 3.2), and finally present the tree-structure attention to utilize the matrices(section 3.3).

3.1 AST Linearization

In order to encode the tree-shaped AST, it first needs to be converted into a sequence with a linearization method. There are the three most representative linearization methods used in current works:

(1) Pre-order Traversal (POT): It visits the tree nodes with pre-order traversal. Sequences obtained by pre-order traversal are lossy since the original ASTs cannot be unambiguously reconstructed back from them.
(2) Structure-based Traversal (SBT): It adds additional brackets [18] to indicate the parental-descendent relationship such that each sequence can be unambiguously mapped back to the AST, but it also doubles the size of the linearized sequence.
(3) Path Decomposition (PD): It represents the AST by concatenating the path between two random leaf nodes. The total number of paths can be too large for computing and therefore random sampling is needed [4].

Table 1 shows the AST in Figure 1 linearized with the above three different methods. For POT and SBT, the linearized trees can be directly fed into the encoder. For PD, the average total number of paths can be over 200, concatenating them all to train is infeasible [4]. In practice, mean pooling is run over the states of each path such that each path has one unique representation. The decoder only attends to these unique representations of paths instead of specific nodes within paths. This can affect the model when copying user-defined names (in leaf nodes) is needed.

We adopt the simplest POT linearization for our model. We show that it has already achieved SOTA results and more complex linearization methods like SBT do not help. PD does not apply to our model since it treats one path as a whole. We will show in section 6.3 that this leads to poor performance in code summarization.

3.2 Relationship Matrices

We define two kinds of relationships between nodes in the tree that we care about: ancestor-descendant (A) and sibling (S) relationships. The former represents the hierarchical information across blocks, and the latter represents the temporal information within one block. Specifically, two nodes have the ancestor-descendant relationship if there exists a directed path from root node that can traverse through them. Two nodes have the sibling relationship if they share the same parent node.

We use two position matrices $A_{N\times N}$ and $S_{N\times N}$ to represent the ancestor-descendent and sibling relationships respectively. N is the total number of nodes in AST. We denote the ith node in the linearized AST as n_i. A_{ij} is the distance of the shortest path between n_i and n_j in the AST. S_{ij} is horizontal sibling distance between n_i and n_j in the AST if they satisfy the sibling relationship. If one relationship is not satisfied, its value in the matrix will be infinity. *Note that we consider the relative relationship between two nodes*, which means $A_{ij} = -A_{ji}$ and $S_{ij} = -S_{ji}$ if a relationship exists between n_i and n_j.

Formally, we use $\mathbf{SPD}(i, j)$ and $\mathbf{SID}(i, j)$ to denote the **S**horted **P**ath **D**istance and horizontal **SI**bling **D**istance between n_i and n_j in the AST. The values in the relationship matrices are defined as:

$$A_{ij} = \begin{cases} \mathbf{SPD}(i,j) & \text{if } |\mathbf{SPD}(i,j)| \le P \\ \infty & \text{otherwise} \end{cases}$$
$$S_{ij} = \begin{cases} \mathbf{SID}(i,j) & \text{if } |\mathbf{SID}(i,j)| \le P \\ \infty & \text{otherwise} \end{cases} \quad (1)$$

P is a pre-defined threshold and nodes with relative distance beyond P will be ignored. We hypothesize that precise relative distance is not useful beyond a certain range. It can both constrain the computation complexity within a constant range and save memory space for storing the relative position embeddings. Figure 2 shows an example of generating matrix A and S, in comparison with the position matrix generated from a linear relationship, which is used in standard Transformers. In the next section, we will introduce how to use these two matrices to dynamically incorporate such relationship information through a tree-structured attention.

3.3 Tree-Structured Attention

Tree-structured attention is built on the standard self-attention with relative position embeddings and disentangled attention. It replaces the relative position embeddings derived from the linear relationship into the two matrices derived from the tree structure.

Self-Attention. Standard self-attention transforms the input sequence $\mathbf{x} = (x_1, \ldots, x_n)$ ($x_i \in \mathbb{R}^d$ which stands for the embedding of n_i) into a sequence of output vectors $\mathbf{o} = (o_1, \ldots, o_n)$ ($o_i \in \mathbb{R}^d$).

The single-head self-attention [49] can be formulated as:

$$\boldsymbol{\alpha_{ij}} = \frac{\boldsymbol{Q}(x_i)\boldsymbol{K}(x_j)^\top}{\sqrt{d}} \qquad o_i = \sum_{j=1}^{n} \sigma(\boldsymbol{\alpha_{ij}})\boldsymbol{V}(x_j) \tag{2}$$

where $\boldsymbol{Q}, \boldsymbol{K} : \mathbb{R}^d \to \mathbb{R}^m$ are query and key functions respectively, $\boldsymbol{V} : \mathbb{R}^d \to \mathbb{R}^d$ is a value function, σ is a scoring function (e.g. softmax or hardmax).

Relative position embedding. Eq 2 is a content-only attention without any position information. The initial Transformer model uses absolute position embeddings to inform about the position. Shaw et al. [36] proposed replacing them with relative position embeddings, which has shown more effective in code summarization tasks [1]. The relative position $\delta(i, j)$ reflects the pairwise distance between n_i and n_j. Denote P as the max relative distance, $\delta(i, j) \in [0, 2P]$ can be defined as:

$$\delta(i,j) = \begin{cases} 0 & \text{for} \quad i-j \le -P \\ 2P & \text{for} \quad i-j \ge P \\ i-j+P & \text{others.} \end{cases} \tag{3}$$

In this way, we can map each relative distance into an embedding representation. The relative position embeddings can be added on top of Eq 2 to inform the pairwise distance.

Disentangled Attention. Disentangled Attention [16] uses relative position embedding as bias in self-attention process. Each word is represented using two vectors that encode its content and relative position in an disentangled way. The attention computation is then divided into three parts: content-to-content, content-to-position and position-to-content, defined as:

$$\tilde{\alpha}_{i,j} = \underbrace{\boldsymbol{Q}(x_i)\boldsymbol{K}(x_j)^\top}_{\text{content-to-content}} + \underbrace{\boldsymbol{Q}(x_i)\boldsymbol{K}^{\boldsymbol{P}}_{\delta(i,j)}{}^\top}_{\text{content-to-position}} + \underbrace{\boldsymbol{Q}^{\boldsymbol{P}}_{\boldsymbol{\delta(j,i)}}\boldsymbol{K}(x_j)^\top}_{\text{position-to-content}} \tag{4}$$

where $\boldsymbol{Q^P}, \boldsymbol{K^P} \in \mathbb{R}^{(2P+1)\times m}$ represent the query and key projection matrices of relative positions. $\boldsymbol{K}^{\boldsymbol{P}}_{\delta(i,j)}$ is the $\delta(i, j)$-th row of $\boldsymbol{K^P}$ and $\boldsymbol{Q}^{\boldsymbol{P}}_{\delta(i,j)}$ is the $\delta(i, j)$-th row of $\boldsymbol{Q^P}$ respectively. The last two items, i.e., content-to-position and position-to-content, are used to measure the relative positions between a word pair.

Besides, for content-to-position computation, as all possible relative positions are always in $[0, 2P]$, the scores of query content $\boldsymbol{Q}(x)$ to all key positions $\boldsymbol{K^P}$ can be first computed as $\boldsymbol{Q}(x)\boldsymbol{K^P}^\top$, and then gathered into $\tilde{\alpha}$ with $\delta(i, j)$ as index. In this way, The relative position embedding can be reused for all query contents and thus reduce the space complexity to $O(2Pm)$.

Attention with Tree-Structured Relationships. Our method essentially replaces $\delta(i, j)$, the relative distance defined under the linear relationship, with $\delta_R(i, j)$ where R stands for either the ancestor-descendent relationship A or the sibling relationship S in the tree structure. $\delta_R(i, j)$ is defined as:

$$\delta_R(i,j) = \begin{cases} R_{ij} + P + 1 & \text{if} \quad R_{ij} \in [-P, P] \\ 0 & \text{if} \quad R_{ij} = \infty \end{cases} \tag{5}$$

R_{ij} refers to either A_{ij} or S_{ij} defined in Eq 1. As there are two kinds of relationships, we consider only one relationship in each head so that *it will not add any additional parameter on top of the standard Transformer.* h_A heads will use $\delta_A(i, j)$ and the rest h_S heads will use $\delta_S(i, j)$. Information from the two relationships will be merged together through multi-head attention. We then replace $\delta(i, j)$ in Eq 4 with $\delta_R(i, j)$ in Formula 5, and apply a scaling factor of $\frac{1}{\sqrt{3d}}$ on $\tilde{\alpha}_{i,j}$ (because it has 3 items). The final output vector is computed as in Eq (6), where $\boldsymbol{V^P}$ represents the value project matrix of relative distances and $\boldsymbol{V^P_{R_{ij}}}$ is the R_{ij}-th row of $\boldsymbol{V^P}$.

$$\tilde{o}_i = \sum_{j}^{j \in \{j | \delta_R(i,j) > 0\}} \sigma\left(\frac{\tilde{\alpha}_{i,j}}{\sqrt{3d}}\right)(\boldsymbol{V}(x_j) + \boldsymbol{V^P_{R_{ij}}}) \tag{6}$$

Note that *we only compute the attention weights for node pairs where $\delta_R(i, j) > 0$)*, which is similar to the idea of sliding window [7] and can reduce the time and space complexity of the self-attention process. We will discuss its implementation and analyze its complexity in sections 4 and 5 respectively.

4 EFFICIENT IMPLEMENTATION

A limitation of the full attention mechanism in standard Transformers is the computational and memory cost that grows quadratically with the sequence length. AST-Trans we proposed can alleviate this problem since the attention scores only need to be computed for node pairs where $\delta_R(i, j) > 0$. Nevertheless, *a memory and computational efficient implementation of AST-Trans that supports parallel processing is non-trivial.* The essence of AST-Trans is similar to previous works that apply sliding windows to constrain the attention within a fixed range [7, 54]. With sliding windows, the node pairs in the sequence data can be planned into a linear distribution (by ignoring node pairs with $\delta(i, j) = 0$ or $2P - 1$) and computed in parallel with matrix partitioning. However, this technique does not apply to us since the position distribution of relevant nodes changes with every tree structure, which makes matrix blocking infeasible. In this section, we present the following 5 alternative implementations of AST-Trans and discuss the pros and cons:

Mask. Mask out the attention scores where $\delta_R(i, j) = 0$ after computing the full attention among all nodes. *It has the same quadratic time and space complexity as in the standard Transformer.*

Loop. Loop over node pairs where $\delta_R(i, j) > 0$ and compute the attention scores. It is memory and computational efficient but *does not support parallel processing.*

Sparse. We can store δ_R as a sparse tensor $ST(\delta_R)$ and deep learning frameworks, such as Pytorch, can automatically skip operations with zero elements when multiplying a sparse tensor with a normal tensor. The mask operation can be optimized (for example, content-to-position attention scores in Eq 4 can be computed by gathering $Q(x)K^{P\top}$ with $ST(\delta_R)$). However, it can only apply to content-to-position and position-to-content. For content-to-content, we still have to use the **Mask** or **Loop** strategy since the production of two sparse tensors is not directly supported.

Gather with COO (GC). On the basis of **Sparse**, the content-to-content computation can be optimized by additional gather operations. The core idea of GC is to put query-key pairs that need to be computed into one-to-one correspondence, and store them as dense matrices. Coordinate format (COO) is a common way to store sparse tensors, where only non-zero elements are stored as tuples of

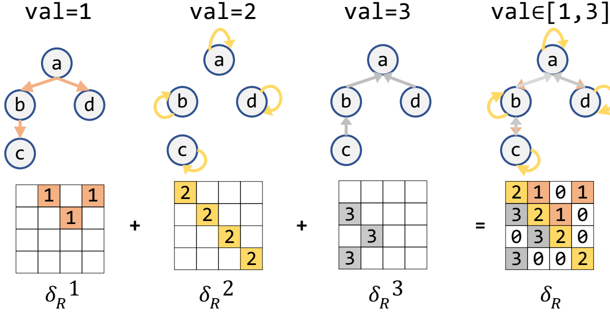

Figure 3: Decompose the relative distance matrix δ_R of the tree "abcd" with max relative distance $P = 1$.

element indices and the corresponding values. Let COO_{row}/COO_{col} denotes the list of row/column indexes, and COO_{val} denotes the list of values in the COO format of δ_R. We then use them as indexes to gather the query and key of content as:

$$Q_{row} = Q(x)[COO_{row}; :]; K_{col} = K(x)[COO_{col}; :]$$
$$Q^P_{val} = Q^P[COO_{val}; :]; K^P_{val} = K^P[COO_{val}; :]$$

By this way, each column in the query content Q_{row} corresponds to the same column in the key content K_{col}. Then we can use matrix dot production to compute attention scores:

$$\alpha_{coo} = Q_{row} \odot K_{col} + Q_{row} \odot K^P_{val} + Q^P_{val} \odot K_{col}$$

where $\odot$ indicates dot production. α_{coo} is a vector and corresponds to the non-zero values in $\tilde{\alpha}$ (Eq. 4), and $\tilde{\alpha}[COO_{row}[i]; COO_{col}[i]] = \alpha_{coo}[i]$. The content-to-position or position-to-content can be computed the same as in **Sparse**, and the total number of gather operations in attention computation is 4 times of non-zero elements in δ_R: 2 for gathering the content and 2 for gathering the position.

Gather with decomposed COO (GDC). To reduce the number of gather operations in **GC**, we can add a matrix decomposition operation on top of it. First, we decompose δ_R by COO_{val} such that each sub-matrix δ^s_R contains only node-pairs with the same relative distance s. An example is shown in Figure 3, where the original δ_R contains 3 distinct values and we decompose it into 3 sub-matrices accordingly. We transfer each sub-matrix δ^s_R into its COO format and use COO^s to indicates the sub-matrix with $val = s$. For each sub-matrix COO^s, we gather content embeddings of nodes by:

$$Q_{row_s} = Q(x)[COO^s_{row}; :], K_{col_s} = K(x)[COO^s_{col}; :]$$

where Q_{row_s} indicates the query content ordered by COO^s_{row}, and K_{col_s} represents the key content ordered by COO^s_{col}. The attention scores can then be computed as:

$$\alpha_{coo_s} = (Q_{row_s} + Q^P_s) \odot (K_{row_s} + K^P_s) - (Q^P_s \odot K^P_s)$$

where α_{coo_s} corresponds to the attention scores of node pairs in δ^s_R. Note that α_{coo_s} is a vector of the same shape as COO^s_{row}. By padding all COO^s to the same length, the attention scores can be computed in parallel and the final attention scores equal to the sum of all α_{coo_s}:

$$\alpha_{coo} = \sum_{s=1}^{2P+1} \alpha_{coo_s}$$

There are 3 benefits of this approach compared with **GC**:

- K^P and Q^P can be reused, as each Q_{row_s} and K_{row_s} have the same relative distance s. The position embeddings of s can be directly added into the content without gather operations.
- Only a quarter of number of gather operation is needed (discussed in 5.3).
- Only one dot production is required, as the second $Q^P_s \odot K^P_s$ can be reused and only $(Q_{row_s} + Q^P_s) \odot (K_{row_s} + K^P_s)$ needs to be calculated.

See Appendix A for the complete algorithm.

5 COMPLEXITY ANALYSIS

In this section, we will discuss the best, worst and average complexity of 5 implementations mentioned above. We use FLOPs (floating point operations) to measure the computational complexity. The considered operations includes: matrix multiplication, matrix dot production, add and gather operation which are the main operations involved for the attention computation. FLOPs of these operations are listed below:

$$\begin{aligned} &FLOPs(A+B) = N(m-1); FLOPS(A[C;:]) = |C| * m \\ &FLOPs(A \odot B) = Nm^2 + N(m-1) \\ &FLOPs(A \times B^\top) = N * FLOPs(A \odot B) \end{aligned} \tag{7}$$

where A and B are two matrices with shape $[N, m]$, $A[C; :]$ indicates gather A with C as the index, $|C|$ is the number of elements in C.

We will focus our analysis on attention heads using the ancestor-descendent relationship (A), but similar ideas can be used to analyze the sibling relationship (S) straightforwardly. As the complexity is related to the number of non-zero elements in δ_A (denoted with $|\delta_A > 0|$). We first analyze the range of $|\delta_A > 0|$, then present the complexity of each implementation.

5.1 Range of $|\delta_A > 0|$

THEOREM 5.1. *For any directed tree T, let $E(i)$ represent the number of paths in T with length i, L represent the length of the longest path in G, we have:*

$$E(1) > E(2) > \cdots > E(L)$$

PROOF. Assuming there are N nodes in the tree, and the root node is at level 1. Define N_j as the number of nodes at level j. For each node at level j, if $j - i > 0$, there exists one path of length i ending with this node, otherwise no such path exists. Hence, $E(i) = N - \sum_{j=1}^{i} N_j$ and $N_j > 0$. Therefore we must have $E(i) > E(i+1)$. □

THEOREM 5.2. *Every tree with N nodes has exactly $N - 1$ edges.*

PROOF. Imagine starting with N isolated nodes and adding edges one at a time. By adding one edge, we will either (1) connect two components together, or (2) close a circuit. Since a tree is fully connected and has no circuit, we must add exactly $N - 1$ edges. □

Least upper & Greatest lower bound. Let $E(0) = N$ denote the number of nodes in a tree. We have $|\delta_A > 0| = E(0) + 2(E(1) + E(2) + \ldots E(P))$ since we consider both positive and negative distance in δ_A. Based on the above two theorems, we can have:

$$E(i) \le E(i-1) - 1 \le \ldots E(0) - i = N - i$$

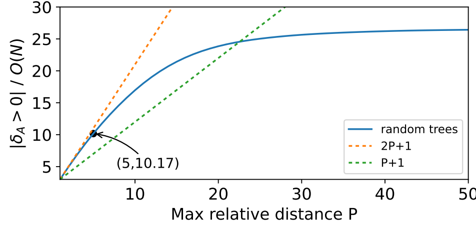

Figure 4: $|\delta_A > 0|$ in case of random trees, the abscissa is the max relative distance P and the ordinate is the non-zero elements in δ_A with the unit of $O(N)$. The coefficient decreases with growing P.

$$|\delta_A > 0| \leq N + 2(N-1+N-2+\ldots N-P) = (N-P)(2P+1)$$

It is the least upper bound for the ancestor-descendent relationship and is achieved only when each node has strictly one child node. The greatest lower bound can be achieved when the tree's depth is 2. In this situation, $E(i) = 0$ for $i \geq 2$ and $|\delta_A > 0| = 3N - 2$.

Average. We can use the Prüfer sequence [35] to simulate random trees so we can estimate the average of $|\delta_A > 0|$ with different tree structures. The tree size N is set in the range of $[50, 500]$ and the out-degree of each node is randomly selected from 1 to $N-1$ (controlled by the max value in Prüfer sequence). We did 1,000 simulation experiments and Figure 4 shows the result.

The average $|\delta_A > 0|$ when P is sampled from a uniform distribution in $[1, 50]$ is $1.16PN$. We can see that *the coefficient in Figure 4 gradually decreases*. For larger P, the average $|\delta_A > 0|$ will be much smaller than the upper bound of $(2P+1)(N-P)$.

5.2 Mask & Loop & Sparse & GC

Mask contains 1 matrix multiplication with $[N, m] \times [m, N]$ in content-to-content, 2 matrix multiplication with $[N, m] \times [m, 2P+1]$ and 2 gather operations with index shape $[N, N]$ for content-to-position and position-to-content, and 2 add operations are used for final score computation. The complexity is $(N^2 + (2P+1)N) * (m^2 + m - 1) + 2N^2 + N - 1$.

Loop As loop only computes non-zero elements in δ_A, the complexity includes 1 dot production of $|\delta_A > 0|(m^2 + m - 1)$ and 2 add operations $|\delta_A > 0| * 2(m-1)$, and equals to $|\delta_A > 0|(m^2 + 3m - 3)$.

Sparse's complexity is same as **Mask** apart from the gather operation with index shape $|\delta_A > 0|$ (the time complexity for gathering sparse tensor as index equals to the number of non-zero elements in it), which equals to $(N^2 + (2P+1)N) * (m^2 + m - 1) + 2|\delta_A > 0| + N - 1$.

GC The complexity in GC is all related to $|\delta_A > 0|$. It contains 4 gather operations, 3 dot production and 2 add operations, which leads to the complexity of $|\delta_A > 0|(m^2 + 3m + 4) + 2(2P+1)Nm$.

5.3 GDC

There are two implementation details in **GDC** to optimize the time and space complexity. Firstly, in a tree, if $s \geq P+1$, the decomposed sub-matrix COO^s has at most one non-zero value in each row. (for example, each non-root node has exactly one parent node in Figure 3.) We can fix COO^s_{row} to $[0, 1, \ldots, N-1]$ and only store the corresponding COO^s_{col}. When $s < P+1$, as the relationship is symmetric, COO^s can be represented with COO^{2P+2-s}. Based on this, when $s \geq P+1$, the query content does not need to be gathered

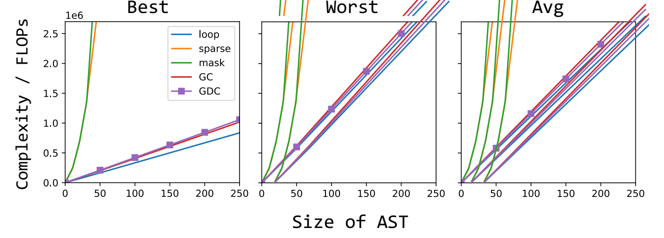

Figure 5: Theoretical complexity with $P = 5$, $m = 32$. loop has the lowest complexity but cannot be parallelized in practice.

Table 2: Statistics of Java and Python Datasets

Perspectives	Java	Python
# of Train instances	69,708	55,538
# of Validation instances	8,714	18,505
# of Test instances	8,714	18,502
Avg. # of tokens in code	120	48
Avg. # of nodes in AST	158	100
Avg. # of tokens in SBT	632	402
Avg. # of tokens in summary	18	9

(as COO^s_{row} is the same index of query), and when $s < P+1$, the key content does not need to be gathered. Hence, we only need $(2P+1)N$ gather operations from content. Secondly, padding positions do not need to be computed in dot production as the padding positions of both Q_{rows} and K_{rows} are the same. After adding the position bias, all Q_{rows} and K_{rows} can be packed before dot production, then unpacked to their original length afterwards. By this way, we only need to compute related node pairs with **one** dot production.

In consequence, the complexity of **GDC** includes $(2P+1)Nm$ gather operations, 1 dot production with shape $[|\delta_A > 0|, m]$ and 3 add operations with shape $[|\delta_A > 0|]$, which equals to $|\delta_A > 0|(m^2 + m - 1) + (6P+3)Nm + (2P+1)N$.

For better comparison, we also show the theoretical complexity in Figure 5 under the hyper-parameters in our experiments. As can be seen, **loop** has the lowest complexity but *cannot be parallelized*. **mask** and **sparse** grow quadratically with the AST size. **GDC** slightly outperforms **GC** and has a complexity close to **loop**.

6 EXPERIMENTS

In this section, we first explain the experimental setup, evaluation metrics and baseline approaches, then report the main results and perform ablation studies. The runtime speed and memory cost of different implementations are provided for comparison. Finally, we present a qualitative analysis and discuss the future directions.

6.1 Experimental Setup

Datasets. Experiments are conducted on the two public code summarization benchmarks, one in Java [19] and the other in Python [51]. To ensure the quality of comments, we filter the comments with less than 4 words, constructors, setters, getters, and tester methods, same as in Shido et al. [41]. When the comment has two or more sentences, only the first sentence is kept as the description of the

Table 3: Comparison of AST-Trans with the baseline methods, categorized based on the input type. * means implemented by ourselves.

Methods	Input	Java			Python		
		BLEU (%)	METEOR (%)	ROUGE-L (%)	BLEU (%)	METEOR (%)	ROUGE-L (%)
CODE-NN[20]	Code	27.6	12.61	41.10	17.36	09.29	37.81
API+CODE[19]		41.31	23.73	52.25	15.36	08.57	33.65
Dual Model[53]		42.39	25.77	53.61	21.80	11.14	39.45
BaseTrans*[1]		44.58	29.12	53.63	25.77	16.33	38.95
Code-Transformer*[57]		45.74	29.65	54.96	30.93	18.42	43.67
Tree2Seq[11]	AST(Tree)	37.88	22.55	51.50	20.07	08.96	35.64
RL+Hybrid2Seq[51]		38.22	22.75	51.91	19.28	09.75	39.34
GCN*[22]		43.94	28.92	55.45	32.31	19.54	39.67
GAT*[50]		44.63	29.19	55.84	32.16	19.30	39.12
Graph-Transformer*[40]		44.68	29.29	54.98	32.55	19.58	39.66
Code2Seq*[4]	AST(PD)	24.42	15.35	33.95	17.54	08.49	20.93
Code2Seq(Transformer)*		35.08	21.69	42.77	29.79	16.73	40.59
DeepCom[18]	AST(SBT)	39.75	23.06	52.67	20.78	09.98	37.35
Transformer(SBT)*		43.37	28.36	52.37	31.33	19.02	44.09
AST-Trans(SBT)*		44.15	29.58	54.73	32.86	19.89	45.92
Transformer(POT)*	AST(POT)	39.62	26.30	50.63	31.86	19.63	44.73
AST-Trans		**48.29**	**30.94**	**55.85**	**34.72**	**20.71**	**47.77**

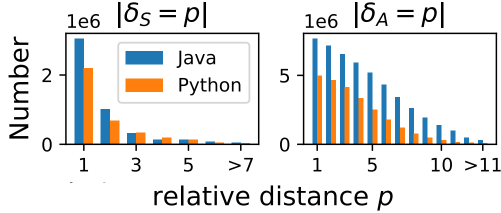

Figure 6: Distribution of relative distance p in training sets

method. Table 2 shows the statistics of the datasets. We also count the distribution of relative distances in Fig 6. As can be seen, most ancestor-descendent and sibling relationships are within the range of 5 and 10 respectively.

Pre-processing. First, we pre-process the summaries by removing the punctuations. Next, we split multi-words, such as "gettable-types", in summaries with wordninja [3] since their corresponding tokens in the source code are split too [53]. We also split the leaf nodes in ASTs into sub-tokens if they are in form of the CamelCase or snake_case. The split nodes are treated as new children of the original parent node. Finally, we reverse the children of the root node to prevent the important information, such as function names or parameters, from being cut when the size of input AST exceeds the maximum size allowed.

Hyper-parameters. If not specified, the maximum size of AST is set to 200 for all experiments, and the vocabulary sizes of both ASTs and comments are set to 30, 000. We use 4 layers of stacked encoder-decoder and set the hidden size $d = 256, m = 32$. For each attention layer, we set $h_A = 1$ and $h_S = 7$. The max relative distance for ancestor-descendant/sibling relationship P_A is set to 10/5 respectively. Feed-forward inner-layer dimension is 2048 and the activation function is gelu [17]. While training, the batch size is 128 and the maximum epochs is 500. Models are trained using the AdamW optimizer [28] with $lr = 1e{-}3, \beta_1 = 0.9, \beta_2 = 0.999, \theta = 1e{-}6$, label smoothing with $\theta_{ls} = 0.1$ [46] and dropout probability [44] of 0.2. The patience in the early stopping mechanism [32] is set to 20 and we select the model based on the BLEU in the validation set [4].

Evaluation Metrics. We evaluate the performance with corpus BLEU [33], METEOR [6], and ROUGE-L [27].

The experiments used the GPUs provided by Aliyun, which use EFLOPS [9] architecture and ACCL [10]. EFlops architecture improves the scalability and efficiency of commodilty clusters (CoW), and ACCL bring the performant efficiency of EFlops architecture to general cluster systems and Cloud scenarios.

6.2 Baselines

We compare the proposed AST-Transformer with 16 baseline methods. They can be divided into 5 groups based on the *input type*:

1: Code. Models with the code as input. It treats code as plain text and does not leverage ASTs. **Code-NN** [20] used RNN while **BaseTrans** [1] used the Transformer. On the basis of Code-NN, **Dual Model**[53] used dual learning to train code summarization and generation together. **API+CODE** [19] used multi encoders to encode code along with the API call sequence. To make up for the lack of structural information, **Code-Transformer** [57] additionally adds four structure distances, including two kinds of distance mentioned in Sec 3.2, to the code tokens and does attention computation separately for each kind of distance. Differently, it does not distinguish embeddings of different relations and uses sine and cosine functions to represent distance embeddings.

2: AST(Tree). Models with the AST as input and encode it with *tree-specific encoders*. There are two main types of such encoders. One uses Tree-LSTM, such as **Tree2Seq** [11] and **RL+Hybrid2Seq** [51]. RL+Hybrid2Seq adds the code information and deep reinforcement for training. The other treats the AST as graph and encodes

[3] https://github.com/keredson/wordninja

[4] We also report the results with best METEOR and ROUGE-L in the validation set in Appendix B

it with graph neural network (GNN) models. We consider three kinds of GNN models including **GCN** [22], **GAT**[50] and **Graph-Transformer** [40]. The edges fed to GNN includes the ancestor-descendant and sibling edges, distinguished by the edge attributes.

3: AST(PD). Models with the AST linearized with path decomposition as input. Path representation needs to be encoded from the nodes, then the whole AST representation is encoded from the path representations. **Code2Seq** [4] is the first approach using PD, and it used two LSTM models to encode hierarchical networks. For fairness of comparison, we also design a new baseline **Code2Seq(Transformer)** by replacing these two LSTM models with the Transformer.

4: AST(SBT). Models with the AST linearized with Structure-based Traversal as input. **DeepCom** [18] is the first work that uses AST (SBT) as input, which encodes it with LSTM. We design a new baseline **Transformer (SBT)** that encodes AST (SBT) with the Transformer. **AST-Trans(SBT)** is our proposed model that inputs SBT with relationship matrices.

5: AST(POT). Models with the AST linearized with pre-order-traversal as input. **Transformer (POT)** is the standard Transformer architecture with AST (POT) as input and **AST-Trans** is our proposed model with tree-structured attention.

All Transformer-based models are based on the relative position embeddings with disentangled attention mentioned in Section 3.3 with *the same number of parameters. The same hype-parameters are used through the way for a fully fair comparison.*

6.3 Main Results

The main result of AST-Trans and the baselines are presented in Table 3 [5]. *AST-Trans outperforms all the baselines on all the three metrics.* Specifically, it outperforms the best baseline by 3.61, 2.17 in BLEU, 1.65, 1.08 in METEOR and 0.87, 3.04 in ROUGE-L on the Java and Python datasets respectively.

Code vs AST (Tree) vs AST (linearized). Apart from AST-Trans, on both two datasets, *using GNNs to encode AST (Tree) achieved the best results.* The reason is that the AST has both structural and semantic information, and the other two input types both lose part of the structural information. All three variants of GNNs achieve similar results and outperform the Tree-LSTM in encoding the AST (Tree). Compared with taking the linearized AST as input, *models only using the code perform better on the Java dataset but worse on the Python dataset.* This could be related to the code length. As code and corresponding ASTs in Python are relatively shorter, encoding ASTs is more effective than in the Java dataset. Therefore, models using linearized ASTs, with the help of additional structural information, are able to outperform models using only the code.

AST(PD) vs AST(SBT) vs AST(POT). Among three linearization methods, when using the Transformer encoder/decoders, *AST (SBT) performs the best on the Java dataset and AST (POT) performs the best on the Python dataset.* AST(SBT) and AST(POT) both have their own advantages. AST(SBT) maintains more structural information than AST(POT) while AST(POT) has the shortest length among these three linearization methods. *Using the AST (PD) as input leads to poor performance on both datasets.* There are two main reasons. On the one hand, AST(PD) method was first proposed for method name completion. Method names are much shorter than the code summaries, and do not include many details. PD linearization extracts features from paths, which aggregates high-level characters but ignores the detailed information in the node. However, code summarization requires more detailed information in the code such as the type of the return value, which is stored in the leaf nodes. On the other hand, Code2Seq(Transformer) uses a hierarchical network and the amount of trained parameters is much larger. It is thereby harder to converge than Transformer(SBT) and Transformer(POT).

[5]The results of BaseTrans [1] in the Python dataset are lower than reported in the paper (-6.75 BLEU, -3.44 METEOR and -7.78 ROUGE), then we set max relative distance P to 16 (kept the same as original paper) and get 27.27(-5.25) BLEU, 15.90(-3.87) METEOR, 38.58(-8.15) ROUGE-L. This reduction may be because that we additionally segment multi-words in comments.

Table 4: Ablation study on AST-Trans with/without A and S.

Model	Dataset	BLEU (%)	METEOR (%)	ROUGE (%)
AST-Trans w/o A	Java	47.74	30.21	54.56
AST-Trans w/o S		48.07	30.62	55.29
AST-Trans		**48.29**	**30.94**	**55.85**
AST-Trans w/o A	Python	34.35	20.15	46.62
AST-Trans w/o S		34.32	20.28	46.87
AST-Trans		**34.72**	**20.71**	**47.77**

Table 5: Ablation study on h_A and h_S on Java Dataset.

h_A	h_S	BLEU (%)	METEOR (%)	ROUGE-L (%)
0	8	47.74	30.21	54.56
1	7	**48.29**	30.94	**55.85**
2	6	48.28	30.94	55.64
3	5	48.25	30.92	55.66
4	4	48.23	**30.96**	55.68
5	3	48.11	30.93	55.46
6	2	48.1	30.74	55.22
7	1	48.24	30.91	55.57
8	0	48.07	30.62	55.29

Impact of relationship matrix R. We compared the performance of three kinds of inputs with or without the relation matrix R: Code-Transformer vs BaseTrans, AST-Trans (SBT) vs Transformer (SBT) and AST-Trans (POT) vs Transformer(POT). Results show that *adding R improves the performance for all these inputs and AST-Trans (POT) performs the best.* This is because Code-Transformer ignores non-leaf node information, and AST-Trans (SBT) stores duplicate information, resulting in too long sequence length. AST-Trans (POT) maintains a short sequence length without losing necessary structural or semantic information.

AST-Trans vs GNN. *AST-Trans outperforms GNNs, the best-performed baseline model in both datasets.* With the help of relationship matrix, AST-Trans includes additional relative distance information. Nodes can perceive information from its p-distance neighbors at each layer. For GNN, however, each node needs p hops to propagate information from these neighbors. In addition, AST-Trans uses multi-head mechanism to compute different relationships in different heads, while all relationships, distinguished by edge attribute, are calculated together in GNNs. AST-Trans also uses extra feed-forward layers and residual connections in the encoder, which could help improve the model generalization.

Table 6: Ablation study on P_A and P_S on Java Dataset.

P_A	P_S	BLEU (%)	METEOR (%)	ROUGE-L (%)
0	0	36.34	23.83	45.58
1	1	46.95	30.33	54.24
5	1	47.45	30.11	54.28
5	3	47.82	30.29	54.62
5	5	48.14	30.77	55.45
10	5	**48.29**	**30.94**	**55.85**

Table 7: Ablation study on the number of layers on Java Dataset.

num	BLEU (%)	METEOR (%)	ROUGE-L (%)
1	46.11	29.36	53.07
2	47.68	30.53	54.97
3	47.41	30.04	54.07
4	48.29	**30.94**	**55.85**
5	47.8	30.39	54.61
6	**48.31**	30.58	55.09

6.4 Ablation studies

We conducted ablation studies on four hyper-parameters: use of each relationship, number of heads used for ancestor-descendant (h_A) and sibling relationships (h_S), max relative distance P and the number of layers. In every study, apart from the hype-parameter that needs to be analyzed, we keep the rest settings unchanged.

Use of two relationships. We verified the impact of using ancestor-descendant or sibling relationship separately in Table 4. Results show that the performance is achieved when using them all. However, *using one of the relationships alone can already achieve close results and outperform all previous baselines.*

Number of attention heads. We change the number of heads used for the ancestor-descendant relationship h_A from 0 to 8 and fix the total number of heads to 8. As can be seem from Table 5, the best performance is obtained with $h_A = 1$ and $h_S = 7$, but there is no significant difference among all combinations of h_A and h_S. Even when one relationship is missing ($h_A = 0$ or $h_S = 0$), the effects are still marginal. However, *when both relationships are removed $h_A = h_S = 0$, the performance drops a lot.* We conjecture that this phenomenon is related to the characteristics of AST. Knowing about one relationship can help the model "guess" the other relationship properly. For example, the node "Compare" can be the child node of "WhileExp", "IFExp" or "SwitchExp", etc, but when it is the sibling of node "Case", it can only be the child of node "SwitchExp". The information about its parent can be "guessed" in attention computation with its sibling "Case". Similarly, node "NameStore" can only appear on the left side of a statement, and nodes with the same parent as it must be its right siblings. Messages of these siblings can be passed to "NameStore" through their common parent. However, there are many cases that the "guess" will not be successful. For example, statements $a > b$ and $b > a$ have the same child nodes and can only be distinguished by sibling relationship, while statements $a = b + a; b = b - a$ and $b = b - a; a = b + a$ only differ in ancestor-descendant relationship. It could be that *the testset does not have enough hard examples that need this fine-grained distinction or the current metrics are not enough to reflect the difference.*

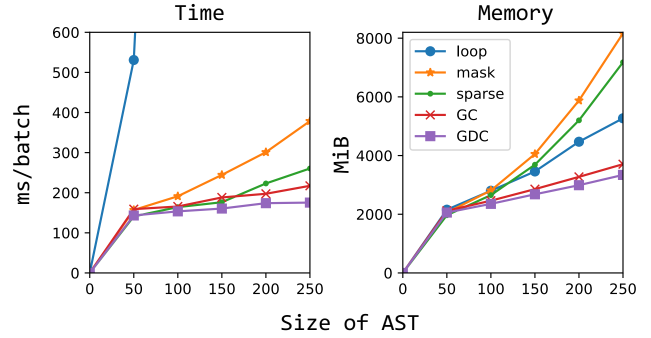

Figure 7: Runtime and memory cost of five implementations with batch size=16. The cost of the mask implementation is equal to the standard Transformer, which grows quadratically with the AST size.

Max relative distance We analyze the impact of the max relative distance P in Table 6 . According to Table 6, the out-degree and depth of most nodes in AST is in [0, 5] and [0, 10]. Therefore, the max relative distance of ancestor-descendant (P_A) and sibling relation (P_S) are selected from [1, 5, 10] and [1, 3, 5] respectively. Results show that as the relative distance grows, the performance improves too, suggesting a wider view of nodes in AST relationships is helpful. However, *the improvement is marginal and even with $P = 1$, the model performance can already outperform all other baselines.* This might be ascribed to the multi-layer stacked encoders. Even for $P = 1$, longer-distance nodes can still be attended to indirectly on upper layers. In practice, P can be set as a hyperparameter to balance the performance-efficiency trade-off.

Number of Layers Finally, we perform ablation study by varying the number of layers, and the results are presented in Table 7. In our experiments, we observe that a deeper model (more layers) performs better, but *the improvement saturates after 4 layers.*

6.5 Complexity analysis

In Fig 7, We analyzed the rum time and memory usage of different implementations mentioned in section 4. Different from the theoretical complexity which analyze the attention computation *in isolate*, operations in GPU can be computed in parallel, and there are other factors, e.g. decoder parameters, dependent libraries, vocabulary embeddings that all need memory usage. Therefore, the need for computing attention scores is only one part of it and leads to the gap between Fig 7 and 5, where the difference across implementations in Fig 7 is much larger. Nevertheless, *the trend stays the same.* Time and memory usage of **GDC** and **GC** both scale *linearly* with the AST size, while the cost of **Mask** and **Sparse** grows *quadratically.* Even with the batched parallelism in GPUs, the implementation of **mask** and **sparse** are still slower than **GDC** and **GC** while requiring significantly more memory cost. **GDC** is faster and with less memory usage than **GC**. The main reason is that **GDC** uses one quarter of gather operations compared with **GC**. **Loop** shows a linear growth in memory usage with AST size, but its time cost is much higher as it does not support parallel operations. When the AST size grows further, we can expect the difference across implementations will become larger and larger.

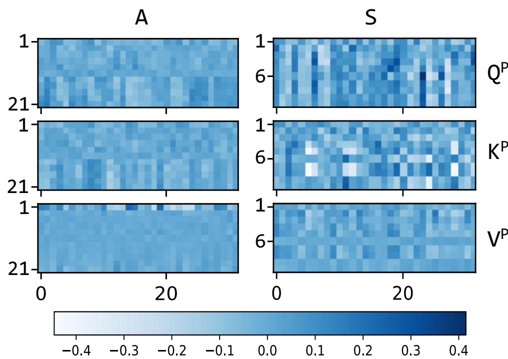

Figure 8: Heatmaps of relative position representations. x-axis is the relative position representation and the y-axis is the relative positions. The variance for the sibling relation (S) is much larger than that for the ancestor-descendent relation (A).

6.6 Visualization and Qualitative Analysis

Visualization. We further visualize the relative position representations of ancestor-descendant (A) and sibling (S) relationships in Fig 8. As can be seen, the variance of relative position embeddings in S is much larger than in A. It implies that our model is *not sensitive to the relative distance between ancestor and descendant nodes*, as the embeddings are almost the same regardless of the positions. In contrast, the variance for sibling nodes is relatively large, and the model can distinguish the sibling nodes with different relative distances. In addition, the relative embeddings in A are demarcated between the upper and lower part, suggesting a clear distinction between ancestor and descendant nodes. It shows that our model *pays more attention to direction rather than distance in A*. It is likely that the exact distance between sibling nodes are more important than that between ancestor-descendant nodes in ASTs.

Qualitative analysis. We provide a couple of examples for qualitative analysis in Table 8. It can be observed that AST-Trans generates the closest summary to the reference, and lack of A or S hurts the quality of summarization. In the first case, the key information is the connection between the sibling nodes method call ("addAll") and parameter ("actions"). Both AST-Trans and AST-Trans w/o A generates the summary as a batch add operation, while AST-Trans w/o S misunderstands it as "adds an action". On the contrary, the meaning of the third case is to get job by the tag first then delete it. The order of execution is controlled by the ancestor-descent relationship (the method call "get" is the child node of "delete"), and AST-Trans w/o A just ignores the "get" operation. The summaries of AST-Trans w/o A and w/o S are both correct in the second case. The statements of the second case are relatively simple and ignoring the order of statements will not affect the function comprehension.

Table 8: Qualitative examples.

```
public QuickActionView addActions(Collection <Action> actions){
    checkShown();
    mActions.addAll(actions);
    return this;
}
```

AST-Trans w/o S: adds a sub - action to the menu
AST-Trans w/o A: adds the given actions to the list of actions
AST-Trans: adds a collection of actions to the quick action view
Human Written: adds a collection of actions to the quick action view

```
public java.lang.Object newInstance() {
   Object o = newInstanceImpl();
   if(o == null){
       throw new InstantiationException();
   }
   return o;
}
```

AST-Trans w/o S: creates a new object initialized to the string object
AST-Trans w/o A: returns a new instance of the object class
AST-Trans: returns a new instance of the object
Human Written: creates a new instance of a class

```
def job_delete_by_tag(tag):
    Job.objects.get(tag=tag).delete()
    return (job_get_by_tag(tag) is None)
```

AST-Trans w/o S: delete a job and return tag
AST-Trans w/o A: delete a job objects
AST-Trans: delete a job based on its tag
Human Written: deletes a job entry based on its tag

7 THREATS TO VALIDITY

There are three main threats to the validity of our evaluation. Firstly, many public datasets are proposed to explore code summarization. We select two widely used ones to evaluate the proposed AST-Transformer, but they may not be representative of other programming languages. Secondly, to ensure a fair comparison as much as possible, we build baselines on the top of the same Transformer architecture. The architecture and hyperparameter choice might be sub-optimal for certain approaches [6]. Finally, there will be a certain gap between the automatic evaluation and the manual evaluation of the summarization results. We select three different automatic evaluation methods to avoid bias as much as possible.

8 RELATED WORKS

Code Summarization. Most approaches on code summarization frame the problem as a sequence generation task and use an encoder-decoder architecture. The only difference between it and traditional machine translation is that programming languages are unambiguous and follow rigid grammar rules. Most approaches either treat the source code as natural language (i.e., a sequence of tokens without specified structures), or utilize its structural information with the help from ASTs or other parsed forms. To encode the code sequence, there exist many encoder architectures like CNN [3], RNN [20, 55] and the Transformer [1]. To leverage the tree-structured AST, tree-based models such as Recursive NN [26], Tree-LSTM [41, 51] and Tree-Transformer [15, 52], are used to encode AST directly. As tree is a special kind of graph, graph-based approaches [2, 12, 23] can also be used to encode ASTs. Some works also combine the code token sequence with the AST and observe improvement [23–25]. Our approach only needs the linearized AST

[6]Nevertheless, AST-Trans performs best among all reported results on both datasets.

and can be built upon the Transformer architecture. More importantly, it restricts the attention range and makes it possible to encode very long AST sequences.

Tree-based Neural Networks. The existing tree-based neural networks can be grouped into two categories depending on their inputs: (1) The models that directly take the tree as input [15, 31, 34, 47]. These models are strongly coupled with the tree structure, and the calculation process needs to be performed simultaneously with the tree traversal. Since trees generally have different shapes by nature, parallization of training these models is non-trivial. (2) The models that take the sequence(s) extracted from the tree as input, such as the sampled paths in the tree [4, 21], the traversal sequence with tree positional embedding [42] or the structure based traversal (SBT) sequence [18]. Taking sampled paths as input is with a certain degree of randomness and instability, and the method of tree positional embedding ignores the concept of paths in the tree (all nodes, even if not related, will participate in the calculation together). Our method improves from these two methods, which can be guaranteed that each node exchanges message on and only on all paths containing it.

9 CONCLUSION

In this paper, we present AST-Trans which can encode ASTs effectively for code summarization. In AST-Trans, each node only pays attention to nodes which share the ancestor-descendent or sibling relationships with it. It brings two benefits: (1) the model is given an inductive bias and will not get lost in the overlong AST sequence, and (2) it can reduce the computational complexity from quadratic to linear. The latter makes it possible to encode long code sequence, e.g., a whole file, which is prohibitively expensive for standard Transformers. We conduct comprehensive experiments, showing that AST-Trans achieve SOTA results on two popular benchmarks while significantly reducing the computational cost.

We believe the basic idea of AST-Trans can also be applied in other structured data like data dependence and control flow graphs. The code is made publicly available to benefit the relevant research. In future work, we plan to improve AST-Trans by incorporating more features of the code snippet, such as API sequence and node type, into the self-attention mechanism.

10 ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (61802167,61802095) ,Natural Science Foundation of Jiangsu Province (No.BK20201250),Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming, and NSF award 2034508. We thank Alibaba Cloud for its high-efficient AI computing service from EFlops Cluster. We also thank the reviewers for their helpful comments. Chuanyi Li and Jidong Ge are the corresponding authors.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 4998–5007. https://doi.org/10.18653/v1/2020.acl-main.449

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gKYo09tX

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[6] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL 2005, Ann Arbor, Michigan, USA, June 29, 2005*, Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare R. Voss (Eds.). Association for Computational Linguistics, 65–72. https://www.aclweb.org/anthology/W05-0909/

[7] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *CoRR* abs/2004.05150 (2020). arXiv:2004.05150 https://arxiv.org/abs/2004.05150

[8] Ernie Chang, Xiaoyu Shen, Hui-Syuan Yeh, and Vera Demberg. 2021. On Training Instance Selection for Few-Shot Neural Text Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 8–13.

[9] Jianbo Dong, Zheng Cao, Tao Zhang, Jianxi Ye, Shaochuang Wang, Fei Feng, Li Zhao, Xiaoyong Liu, Liuyihan Song, Liwei Peng, et al. 2020. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 610–622.

[10] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, et al. 2021. ACCL: Architecting Highly Scalable Distributed Training Systems with Highly-Efficient Collective Communication Library. *IEEE Micro* (2021).

[11] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-Sequence Attentional Neural Machine Translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1078

[12] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1ersoRqtm

[13] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 223–226. https://doi.org/10.1145/1810295.1810335

[14] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky (Eds.). IEEE Computer Society, 35–44. https://doi.org/10.1109/WCRE.2010.13

[15] Jacob Harer, Christopher P. Reale, and Peter Chin. 2019. Tree-Transformer: A Transformer-Based Method for Correction of Tree-Structured Data. *CoRR* abs/1908.00449 (2019). arXiv:1908.00449 http://arxiv.org/abs/1908.00449

[16] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2021. Deberta: decoding-Enhanced Bert with Disentangled Attention. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=XPZIaotutsD

[17] Dan Hendrycks and Kevin Gimpel. 2016. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR* abs/1606.08415 (2016). arXiv:1606.08415 http://arxiv.org/abs/1606.08415

[18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210. https://doi.org/10.1145/3196321.3196334

[19] 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 Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 2269–2275. https://doi.org/10.24963/ijcai.2018/314

[20] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics. https://doi.org/10.18653/v1/p16-1195

[21] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 150–162. https://doi.org/10.1109/ICSE43902.2021.00026

[22] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[23] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 184–195. https://doi.org/10.1145/3387904.3389268

[24] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 795–806. https://doi.org/10.1109/ICSE.2019.00087

[25] Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. DeepCommenter: a deep code comment generation tool with hybrid lexical and syntactical information. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1571–1575. https://doi.org/10.1145/3368089.3417926

[26] Yuding Liang and Kenny Qili Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 5229–5236. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16492

[27] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81. https://www.aclweb.org/anthology/W04-1013

[28] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[29] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Trans. Software Eng.* 42, 2 (2016), 103–119. https://doi.org/10.1109/TSE.2015.2465386

[30] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*. IEEE Computer Society, 23–32. https://doi.org/10.1109/ICPC.2013.6613830

[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

[32] Genevieve B. Orr and Klaus-Robert Müller (Eds.). 1998. *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science, Vol. 1524. Springer. https://doi.org/10.1007/3-540-49430-8

[33] 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, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 311–318. https://www.aclweb.org/anthology/P02-1040/

[34] Jordan B. Pollack. 1990. Recursive Distributed Representations. *Artif. Intell.* 46, 1-2 (1990), 77–105. https://doi.org/10.1016/0004-3702(90)90005-K

[35] Heinz Prüfer. 1918. Neuer beweis eines satzes über permutationen. *Arch. Math. Phys* 27, 1918 (1918), 742–744.

[36] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 464–468. https://doi.org/10.18653/v1/n18-2074

[37] Xiaoyu Shen, Youssef Oualil, Clayton Greenberg, Mittul Singh, and Dietrich Klakow. 2017. Estimation of Gap Between Current Language Models and Human Performance. *Proc. Interspeech 2017* (2017), 553–557.

[38] Xiaoyu Shen, Jun Suzuki, Kentaro Inui, Hui Su, Dietrich Klakow, and Satoshi Sekine. 2019. Select and Attend: Towards Controllable Content Selection in Text Generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 579–590.

[39] Xiaoyu Shen, Yang Zhao, Hui Su, and Dietrich Klakow. 2019. Improving latent alignment in text summarization by generalizing the pointer generator. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 3753–3764.

[40] Yunsheng Shi, Zhengjie Huang, Shikun Feng, Hui Zhong, Wenjing Wang, and Yu Sun. 2021. Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 1548–1554. https://doi.org/10.24963/ijcai.2021/214

[41] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic Source Code Summarization with Extended Tree-LSTM. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*. IEEE, 1–8. https://doi.org/10.1109/IJCNN.2019.8851751

[42] Vighnesh Leonardo Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 12058–12068.

[43] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto (Eds.). ACM, 43–52. https://doi.org/10.1145/1858996.1859006

[44] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (2014), 1929–1958. http://dl.acm.org/citation.cfm?id=2670313

[45] Hui Su, Xiaoyu Shen, Zhou Xiao, Zheng Zhang, Ernie Chang, Cheng Zhang, Cheng Niu, and Jie Zhou. 2020. Moviechats: Chat like humans in a closed domain. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 6605–6619.

[46] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[47] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. 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 of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 1556–1566. https://doi.org/10.3115/v1/p15-1150

[48] Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, and Bin Luo. 2021. AST-Transformer: Encoding Abstract Syntax Trees Efficiently for Code Summarization. *arXiv preprint arXiv:2112.01184* (2021).

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need

[50] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJXMpikCZ

[51] 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, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM,

397–407. https://doi.org/10.1145/3238147.3238206

[52] Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranSˆ3: A Transformer-based Framework for Unifying Code Summarization and Code Search. *CoRR* abs/2003.03238 (2020). arXiv:2003.03238 https://arxiv.org/abs/2003.03238

[53] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 6559–6569.

[54] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html

[55] Yang Zhao, Xiaoyu Shen, Wei Bi, and Akiko Aizawa. 2019. Unsupervised rewriter for multi-sentence compression. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2235–2240.

[56] Yuxiang Zhu and Minxue Pan. 2019. Automatic Code Summarization: A Systematic Literature Review. *CoRR* abs/1909.04352 (2019). arXiv:1909.04352 http://arxiv.org/abs/1909.04352

[57] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=Xh5eMZVONGF

A ALGORITHM OF GDC

Algorithm 1 Self-Attention with Relationship matrix

Input: Hidden state $\boldsymbol{H}$, COO format of relationship martix COO, content functions $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$, relative distance projection matrix $\boldsymbol{Q^P}, \boldsymbol{K^P}, \boldsymbol{V^P}$.

1: $\boldsymbol{K_c} = \boldsymbol{K}(H), \boldsymbol{Q_c} = \boldsymbol{Q}(H), \boldsymbol{V_c} = \boldsymbol{V}(H)$
2: **for** $i = 0, \ldots, 2P+1$ **do**
3:     **for** $j = 0, \ldots, N-1$ **do**
4:         $\tilde{\boldsymbol{Q_c}}[i;j;:] = \boldsymbol{Q_c}[\boldsymbol{COO_{col}}[i*N+j];:]$
5:         $\tilde{\boldsymbol{K_c}}[i;j;:] = \boldsymbol{K_c}[\boldsymbol{COO_{row}}[i*N+j];:]$
6:         $\tilde{\boldsymbol{V_c}}[i;j;:] = \boldsymbol{V_c}[\boldsymbol{COO_{row}}[i*N+j];:]$
7:     **end for**
8: **end for**
9: $\tilde{\boldsymbol{\alpha}} = (\boldsymbol{Q_c} + \boldsymbol{Q^P}) \odot (\boldsymbol{K_c} + \boldsymbol{K^P}) - \boldsymbol{Q^P} \odot \boldsymbol{K^P}$
10: $\tilde{\boldsymbol{\alpha}} = \exp(\frac{\tilde{\boldsymbol{\alpha}}}{\sqrt{3d}})$
11: **for** $i = 0, \ldots, 2P+1$ **do**
12:     **for** $j = 0, \ldots, N-1$ **do**
13:         $\tilde{\boldsymbol{\alpha}}_{\boldsymbol{sum}}[:;\boldsymbol{COO_{row}}[i*N+j]] += \tilde{\boldsymbol{\alpha}}[i,j]$
14:     **end for**
15: **end for**
16: $\tilde{\boldsymbol{\alpha}} = \frac{\tilde{\boldsymbol{\alpha}}}{\tilde{\boldsymbol{\alpha}}_{\boldsymbol{sum}}}$
17: **for** $i = 0, \ldots, 2P+1$ **do**
18:     **for** $j = 0, \ldots, N-1$ **do**
19:         $\tilde{\boldsymbol{o}}[\boldsymbol{COO_{row}}[i*N+j];:] = (\tilde{\boldsymbol{V_c}}[i;j;:] + \boldsymbol{V^P}[i;:]) \cdot \tilde{\boldsymbol{\alpha}}[i,j]$
20:     **end for**
21: **end for**

Output: $\tilde{\boldsymbol{o}}$

For better re-implementation, we also show the algorithm of **GDC**. line 1-10 describes the attention score computation process. $\tilde{Q_c}$, $\tilde{K_c}$ and $\tilde{V_c}$ are reshaped to $[2P+1, N, d]$. Note that the attention scores $\tilde{\alpha}$ have a different shape with traditional attention scores, so we redesigned the softmax function in line 11-16. The attention scores belonging to the same query vector, distinguished by $COO_{row}[i*N+j]$, are added together as $\tilde{\alpha}_{sum}$. Then the softmax function can be formed as $\tilde{\alpha}$ divide by $\tilde{\alpha}_{sum}$. Finally in line 17-21, relative distance bias V^P is added to the value context, and then is multiplied with the attention scores $\tilde{\alpha}$.

Table 9: Comparison of AST-Trans with different model selection strategy on Java Dataset.

Model	BLEU	METEOR	ROUGE-L
AST-Trans(best_eval_BLEU)	**48.29**	30.94	55.85
AST-Trans(best_eval_METEOR)	47.02	**31.90**	55.72
AST-Trans(best_eval_ROUGE-L)	46.92	29.99	**57.01**

B THE INFLUENCE OF MODEL SELECTION STRATEGY

The results reported in the paper come from the model with best BLEU score in the validation dataset. We then separately select two other models with the best METEOR, and ROUGE-L score in the valid dataset, and then evaluate their performances on test dataset. Results in Table 9 show that the model selection strategy indeed influences the performance. This may explain why that the improvement of AST-Trans is inconsistent in different metrics.