Deep Code Search with Naming-Agnostic Contrastive Multi-View Learning

JIADONG FENG, Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, Xiamen, China
WEI LI, School of Electronic and Computer Engineering, Peking University, Shenzhen, China
SUHUANG WU, Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, Xiamen, China
ZHAO WEI, YONG XU, and JUHONG WANG, Tencent, Shenzhen, China
HUI LI, Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, Xiamen, China

Software development is a repetitive task, as developers usually reuse or get inspiration from existing implementations. Code search, which refers to the retrieval of relevant code snippets from a codebase according to the developer's intent that has been expressed as a query, has become increasingly important in the software development process. Due to the success of deep learning in various applications, a great number of deep learning-based code search approaches have sprung up and achieved promising results. However, developers may not follow the same naming conventions and the same variable may have different variable names in different implementations, bringing a challenge to deep learning-based code search methods that rely on explicit variable correspondences to understand source code. To overcome this challenge, we propose a Naming-Agnostic Code Search (NACS) method based on contrastive multi-view code representation learning. NACS strips information bound to variable names from Abstract Syntax Tree (AST), the representation of the abstract syntactic structure of source code, and focuses on capturing intrinsic properties solely from AST structures. We use semantic-level and syntax-level augmentation techniques to prepare realistically rational data and adopt contrastive learning to design a graph-view modeling component in NACS to enhance the understanding of code snippets. We further model ASTs in a path view to strengthen the graph-view modeling component through multi-view learning. Extensive experiments show that NACS provides superior code search performance compared to baselines and NACS can be adapted to help existing code search methods overcome the impact of different naming conventions. Our implementation is available at https://github.com/KDEGroup/NACS.

Jiadong Feng and Wei Li contributed equally to this work.

This work was partially supported by Natural Science Foundation of Xiamen, China (No. 3502Z202471028), National Natural Science Foundation of China (Nos. 62002303 and 42171456), and CCF-Tencent Open Fund (No. RAGR20210129).

Authors' Contact Information: Jiadong Feng, Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, Xiamen, China; e-mail: jdfeng@stu.xmu.edu.cn; Wei Li, School of Electronic and Computer Engineering, Peking University, Shenzhen, China; e-mail: weilileopku@gmail.com; Suhuang Wu, Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, Xiamen, China; e-mail: wusuhuang@stu.xmu.edu.cn; Zhao Wei, Tencent, Shenzhen, China; e-mail: zachwei@tencent.com; Yong Xu, Tencent, Shenzhen, China; e-mail: rogerxu@tencent.com; Juhong Wang, Tencent, Shenzhen, China; e-mail: julietwang@tencent.com; Hui Li (corresponding author), Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China, Xiamen University, Xiamen, China; e-mail: hui@xmu.edu.cn.

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 the author(s) 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.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1556-472X/2025/7-ART116
https://doi.org/10.1145/3737878

CCS Concepts: • **Information systems** → **Retrieval models and ranking**; • **Software and its engineering** → **Software notations and tools**; • **Computing methodologies** → **Learning latent representations**;

Additional Key Words and Phrases: code search, multi-view learning, graph self-supervised learning, graph neural network

Associate Editor: Victor S. Sheng

ACM Reference format:
Jiadong Feng, Wei Li, Suhuang Wu, Zhao Wei, Yong Xu, Juhong Wang, and Hui Li. 2025. Deep Code Search with Naming-Agnostic Contrastive Multi-View Learning. *ACM Trans. Knowl. Discov. Data.* 19, 6, Article 116 (July 2025), 26 pages.
https://doi.org/10.1145/3737878

1 Introduction

Code search takes search queries that manifest the software developer' intent as inputs and returns desired code snippets. To enhance software development productivity and quality, programmers use code search tools to find high-quality code snippets in existing projects when they debug, write code, look for code to reuse, or learn API usage [48]. Existing studies show that during software development, roughly one-fifth of the development time is used to search code examples [8, 69], indicating that code search has become an essential part of software development.

Due to the pivotal role of code search in software development, much effort has been devoted to improving the quality of code search. Early works mainly use keyword matching between code snippets and search queries written in **Natural Languages (NLs)** [48]. These methods typically employ traditional information retrieval algorithms [6, 53, 57] to measure the relevance between queries and candidate code snippets. Recently, the success of deep learning techniques has greatly promoted the development of code search approaches [48, 83, 89]. Various deep learning techniques have been introduced to improve code search, including but not limited to **Deep Neural Networks (DNNs)** [27], meta learning [13], and pre-training [25, 29]. They encode queries and code snippets into low-dimensional spaces and measure their relevance through the similarity (e.g., cosine similarity) between representation vectors.

However, existing works lack the consideration of the impact of different naming styles (we call it *naming issue* in this article for simplicity). Software developers may not follow the same naming convention, resulting in different naming of the same variable [29]. The naming issue constitutes a barrier for effectively deploying deep learning techniques in code search since (1) it is hard to find correspondences when the same variable has different variable name in different code snippets, and (2) many deep learning techniques rely on entity correspondences (i.e., the same object is always represented by the same representation vector). In the literature, very few works have been done on remedying the naming issue. Guo et al. [29] propose to model the dataflow graph, which represents the dependency relation between variables and is the same under different abstract grammars for the same source code, to avoid the impact of different naming conventions in code representation learning. Nevertheless, source code is different from plain text as it requires strict grammar. How to avoid the impact of the naming issue when modeling the **Abstract Syntax Tree (AST)**,[1] which represents the abstract syntax of source code and is prevalently modeled in code representation learning [1, 48, 83], is under-exploited. Previous code representation learning methods not only capture syntactic information contained in ASTs but also encode AST node names, which contain variable names, as the semantic information of ASTs. Figure 1 provides

[1]The background of AST is provided in Section 3.1.1.

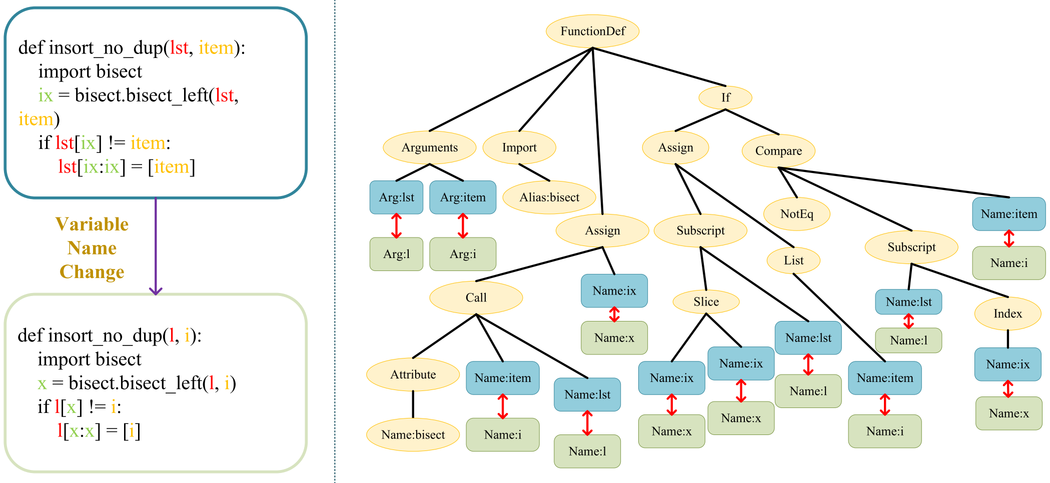

Fig. 1. Examples of two code snippets and their corresponding ASTs.

an example to explain why AST-based code representation learning methods suffer from the naming issue. The left part of Figure 1 provides examples of two Python code snippets. The code snippet at the bottom is the result after changing variable names in the code snippet on the top. The right part of Figure 1 shows ASTs of the two code snippets. The blue blocks correspond to the original variable names, and they are replaced by the green blocks after changing variable names. It is difficult for contemporary code representation learning techniques to understand that corresponding variables in the two code snippets (e.g., lst and l) are the same due to their different names [29].

To overcome the challenge brought by the naming issue, in this article, we propose a **Naming-Agnostic Code Search (NACS)** method based on contrastive multi-view code representation learning. NACS adopts the idea of contrastive learning [50] and multi-view learning [43] to handle the impact of different naming conventions. In summary, our contributions are as follows:

— To our best knowledge, we are the first to study how to handle the impact of different naming conventions when modeling ASTs for the code search task.
— To alleviate the reliance on the explicit correspondence of the same variable in different code snippets in code representation learning, we strip information bound to variable names from input ASTs and focus on capturing intrinsic properties solely from ASTs' structures. Then, we design different semantic-level and syntax-level augmentation techniques to prepare realistically rational data and adopt contrastive learning to design a graph-view modeling component in NACS to enhance the understanding of code snippets.
— To avoid the information loss caused by ignoring explicit variable correspondences, we design another path-view modeling component that models AST paths. Path-view modeling is used to strengthen the graph-view modeling component through multi-view learning.
— We conduct experiments to illustrate the effectiveness of NACS on the code search task. We show that (1) the overall performance of NACS is superior to existing code search methods; (2) existing code search methods perform poorly when facing the naming issue, while the performance of NACS is not affected; and (3) the idea of NACS can be adapted to help existing code search methods handle the naming issue in code representation learning.

2 Related Work

In this section, we will illustrate the related work of NACS.

2.1 Code Representation Learning

Software is ubiquitous in modern society, and there is an ongoing demand for developing new software and enhancing existing software [83]. Considering the importance of software development, a great number of assistance tools (e.g., code completion tool [71, 75], code search tool [53, 89], code summarization tool [26, 44], issue-commit recovery tool [87] and code refactoring tool [49]) have been developed to boost the efficiency of software development. Code and its affiliated documents (e.g., code comments) are important elements in software development. Strengthening the understanding of code can improve assistance tools and help engineers construct better software [42]. Inspired by the success of deep learning techniques, researchers have recently explored applying DNNs to enhance code representation learning, and these works can be roughly classified into two categories.

One direction is to specifically design a DNN architecture to model programs. Ben-Nun et al. [7] show that a single **Recurrent Neural Network (RNN)** over the Intermediate Representation of the source code can outperform previous code comprehension approaches. Zhang et al. [88] split each large AST into a sequence of small statement trees which are encoded by Recursive Neural Network. Based on the sequence of statement vectors, a bidirectional RNN model is used to produce the vector representation of a code fragment. Alon et al. [3, 4] and Peng et al. [60] model a code snippet as the set of compositional paths in its AST and adopt RNN or Transformer to learn code representations. Allamanis et al. [2], Brockschmidt et al. [10], and Cvitkovic et al. [18] propose to use graphs to represent both the syntactic and semantic structure of code. Then, **Graph Neural Network (GNN)** is applied for code representation learning over graphs. Jayasundara et al. [36] and Bui et al. [11] propose TreeCaps that fuses capsule networks with tree-based convolutional neural networks to enhance code comprehension. Wang et al. [74] adopt contrastive learning and GNN to better learn the hierarchical relationships between AST nodes so that the AST hierarchy can be modeled in code representation learning.

Another direction, which recently attracts great attention, is to pre-train Bert-style [19], large-scale models over massive unlabeled code data in a self-supervised manner to gain a better understanding of programs. CodeBERT [25] is a bimodal pre-trained model for NL and **Programming Language (PL)**. It treats source code as plain text and uses a hybrid objective function including standard masked language modeling [19] and replaced token detection [16] in pre-training. CBERT [12] is a Bert-based pre-training model for C language. It uses masked language modeling and whole-word masking (i.e., mask all tokens with the same string type) as pre-training objectives. GraphCodeBERT [29] captures dataflow (semantic-level structure) instead of syntactic-level structure (e.g., AST) of source code in the pre-training stage. In addition to the standard masked language modeling task, GraphCodeBERT further adopts two structure-aware pre-training tasks: one is to predict code structure edges, and the other is to align representations between source code and code structure. UniXcoder [28] utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverages cross-modal contents like serialized ASTs and code comments to enhance code representation in pre-training.

2.2 Code Search

Code search, which is also called code retrieval in the literature, refers to the retrieval of relevant code snippets from a large codebase according to programmer's intent that has been expressed as a query [48, 78]. Code search lowers the cost of learning the usage of new APIs by providing code

examples. Code search also reduces programmers' cognitive burden and boosts the efficiency of software development by providing reusable code snippets that can be easily adapted. Therefore, code search has become increasing important in the software development process and attracted considerable attention.

Early works of code search focus on applying traditional information retrieval or NL processing methods to retrieve relevant code snippets [78]. Sourcerer [5] is a code search engine based on the text search engine Lucene, and it searches for relevant code snippets using TF-IDF and some heuristics designed for source code. Portfolio [58] finds relevant code snippets by combining various NL processing and indexing techniques with PageRank [9] and spreading activation [17] algorithms. CodeHow [53] uses API documentation to help identify which API a query is likely to refer to. The text similarity score between the query and the API documentation is computed using Vector Space Model [65]. These methods simply treat queries and code snippets as plain texts, which cannot capture the semantic relationship between the query and the code.

With the development of deep learning, more and more recent works try to use DNNs to bridge the semantic gap between PL in code and NL in query by embedding code snippets and NL descriptions into a high-dimensional vector space Gu et al. [27] firstly apply RNN in code retrieval and propose DeepCS. The relevant score is estimated by measuring cosine similarity between query representation and code representation. DeepCS is trained to reduce the distance of matching code-query pairs while keeping unrelated couples apart. Multiple features are considered in DeepCS, including method name, API sequence, code tokens, and code descriptions. PSCS [70] extracts AST paths for code search. An AST path in PSCS is the path extracted from the AST by walking from one terminal to another. A path has non-terminal nodes in the middle and terminals at both ends. PSCS deploys Bi-LSTM to encode AST paths and uses them to represent code snippets in code search. DeGraphCS [86] transfers source code into variable-based flow graphs based on the intermediate representation technique and applies GNN to model the VFG for code search. Ling et al. [47] propose a deep graph matching method for code search. They represent both query texts and code snippets with the unified graph-structured data. Then, query graph and code graph are encoded by relational GNN [66] for computing the relevant score. Zhang et al. [89] study the bias issue of deep code search and propose a reranking-based method to alleviate code search biases.

2.3 Graph Self-Supervised Learning (GSSL)

Graph supervised/semi-supervised learning relies on scarce data labels to supervise model learning. Differently, GSSL [76] models graphs through handcrafted auxiliary tasks (pretext tasks) that acquire supervision signals from intrinsic graph properties [52]. Since GSSL alleviates label reliance, it draws more and more attention recently [77, 79].

Existing GSSL methods can be grouped into four types according to the used pretext tasks:

- Generation-based graph SSL approaches take the full graph or a subgraph as the model input and reconstruct one of the components. Typical pretext tasks include feature generation and structure generation. Feature generation learns to reconstruct the feature information of graphs, e.g., mask node or edge features, and then recover the masked features [33, 38, 85, 91]. Structure generation learns to reconstruct the topological structure information of graphs, e.g., randomly mask edges, and then recover edges [38, 91].
- Auxiliary property-based methods adopt traditional graph tasks such as clustering [68, 85], graph partitioning [85], shortest distance prediction [61], or node similarity prediction [39, 91] as the pretext task. These methods have a similar training paradigm with supervised learning since both of them learn with "sample-label" pairs [52]. The difference is auxiliary property-based graph SSL generates pseudo label without manual labeling.

— Contrast-based methods are based on **Mutual Information (MI)** maximization [32]. They maximize the MI between related graph instances and minimize the MI between unrelated graph instances [32, 52]. Two stages are typically required for contrast-based graph SSL. In the graph augmentation stage, different graph instances are generated through node feature masking [38], node feature shuffling [63], edge dropping/adding [84], or subgraph sampling [31, 33, 37, 62]. In the graph contrastive learning stage, pretext tasks are leveraged to maximize the MI between positive instances. Contrast can be conducted at same scale or cross scales. The former discriminates instances in an equal scale (e.g., node versus node) [62, 84] while the later discriminates instances across different granularities (e.g., node versus subgraph) [56, 64].
— Compared to methods that only consider one pretext task, hybrid methods adopt several pretext tasks to leverage different types of supervisions [52, 85]. These methods optimize various pretext tasks in a manner of multi-task learning.

2.4 Multi-View Learning

Multi-View Representation Learning (MVL) learns representations from multi-view data [43, 81]. Multi-view data is prevalent in real-world applications where objects are depicted by multi-modal information. Different views of the same object usually contain complementary information and thus MVL can learn more comprehensive representations than single-view learning methods.

Due to the powerful feature abstraction ability, deep learning techniques have exerted considerable influence over the research of MVL. Various deep learning techniques have been adapted for MVL:

— Multi-view CNN models from multiple feature sets with access to multi-view information of the target data to obtain more discriminative common representations. Two types of multi-view CNN exist: one-view-one-net multi-view CNN and multi-view-one-net multi-view CNN. One-view-one-net mechanism adopts one CNN for each view and extracts feature representation of each view separately, then multiple representations are fused [24, 67]. Multi-view-one-net mechanism feeds multi-view data into the same neural network to get the overall representation [21].
— Multi-view RNN is designed for modeling multi-view sequential data. For example, Mao et al. [55] designed a multi-view RNN containing a vision network, a language network, and a multi-view network for image captioning. Karpathy and Fei-Fei [40] propose a multi-view alignment model based on RNN to narrow the interview relationship between visual and textual data for MVL.
— Multi-view GNN is tailored for MVL over graph data. Fan et al. [22] designed the one2multi graph auto-encoder that learns node representations by using content information to reconstruct the graph structure from multiple views. Ma et al. [54] observe that atoms and bonds affect the chemical properties of a molecule. They exploit both node (atom) and edge (bond) information simultaneously to build an expressive, multi-view GNN.
— Multi-view auto-encoder endorses auto-encoder with the ability of learning from multi-view data. For instance, Ngiam et al. [59] designed a bimodal auto-encoder to find a reconstruction of both audio and video views by minimizing the reconstruction error of the two input views and reconstructed representation. Feng et al. [23] propose the correspondence auto-encoder for cross-modal retrieval, which simultaneously learns the shared information of multiple modalities and the specific information in each individual modalities.
— Multi-view GAN aims to overcome the limitation of the single-view GAN, i.e., the single-pathway GAN may learn incomplete representation. For example, Donahue et al. [20] designed

the bidirectional GAN to train an inference network and a generator jointly, which learns an inverse mapping that projects data back into the latent space. Tian et al. [72] propose a two-pathway GAN to maintain the completeness of the learned embedding space.

— Multi-view contrastive learning is designed for learning consistent representations among different views even when labels are scarce and the multi-view information is incomplete or inconsistent. Lin et al. [45, 46] propose a novel objective that incorporates representation learning and data recovery into a unified contrastive learning framework from the view of information theory. To overcome the problems of view inconsistency and instance incompleteness, Yang et al. [82] propose robust multi-view clustering with incomplete information (SURE). SURE is a novel contrastive learning paradigm which uses the available pairs as positives and randomly chooses some cross-view samples as negatives.

3 Our Framework NACS

In this section, we will illustrate the details of NACS, a novel naming-agnostic contrastive multi-view learning-based code search method. NACS mainly consists of three parts: augment data (Section 3.1), pre-train via naming-agnostic contrastive multi-view code representation learning (Section 3.2), and enhance code search (Section 3.3).

3.1 Data Augmentation

Data augmentation, an essential step in contrastive learning, creates realistically rational data through applying certain transformation strategies that do not affect the information of original data too much. Without requiring manual labeling, data augmentation enriches the supervision signals.

We adopt different **Program Transformation (PT)** strategies and **Structure Transformation (ST)** strategies to generate augmented data for code snippets in NACS. They are applied over the AST of a code snippet and help NACS better capture intrinsic code features and distinguish different code snippets without requiring more manual labels.

3.1.1 AST. Any PL has an explicit context-free grammar, and it can be used to parse source code into an AST which represents the abstract syntactic structure of source code [30]. We start by providing the definition of the AST:

Definition 3.1 (AST). An AST of a code snippet c is a tuple $\langle \mathcal{NT}, \mathcal{T}, r, f \rangle$ where $\mathcal{NT}$ is the non-terminal node set, $\mathcal{T}$ is the terminal node set, r is the root node, and $f : \mathcal{NT} \rightarrow (\mathcal{NT} \cup \mathcal{T})$ is a mapping function that maps a non-terminal node to its child nodes.

An AST is a tree where each non-leaf node corresponds to a *non-terminal* in the context-free grammar specifying structural information (e.g., `ForStatement` and `WhileStatement`). Each leaf node corresponds to a *terminal* (e.g., variable names and operators) in the context-free grammar encoding program text. An AST can be converted back into source code easily. From the AST shown in Figure 1, we can see that each non-leaf node contains a type attribute (e.g., `Arguments`) and each leaf node contains a type attribute and a value attribute (e.g., `Name:ix` means that the type is `Name` and the value is `ix`). AST has been widely adopted in designing software engineering tools [88]. On the one hand, compared to plain source code, AST is abstract and it does not include all details such as the punctuation and delimiters. On the other hand, AST can describe both the lexical and syntactic structural information of the code snippet, and it provides a structural view of the source code which is pivotal in helping NACS achieve naming-agnostic code search.

```
# Dead code statement 1
# a is a new variable while b exists in the code snippet
a = b

# Dead code statement 2
# Print a random number of existing variables
print(d, e, f)

# Dead code statement 3
# Print current timestamp
import time
if 1:
  print(time.time())
```

Fig. 2. Three dead code statements.

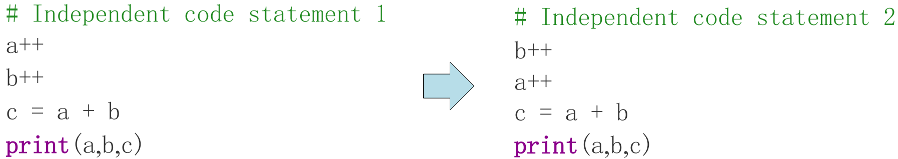

Fig. 3. Two independent code statements.

3.1.2 PT and ST Strategies. PT strategies are semantics-level augmentation while ST strategies are syntax-level augmentation. We design the following PT and ST strategies for data augmentation in NACS:

PT Strategies. We use three PT strategies to generate augmented versions of a code snippet that do not change the original *semantics* and they can help NACS capture program semantics better:

— *Insert Dead Code Statement*: NACS randomly inserts one dead code statement that does not affect the functionality of the code snippet to a random position of each code snippet. This is equivalent to adding a subtree to the corresponding AST. Figure 2 illustrates three dead code statements.
— *Swap Independent Statements*: By analyzing the dependencies of variables, NACS exchanges two statements in each code snippet that do not depend on each other. Figure 3 illustrates an example of swapping independent statements. In the example, the order of a++ and b++ does not affect the result, and they are independent statements.
— *Change Loop Statements*: By replacing corresponding nodes in the AST, NACS changes for-loop to while-loop or *vice versa*. If multiple iteration structures exist in one code snippet, one of them will be randomly picked and changed. Figure 4 depicts an example of changing for-loop to while-loop. In the example, the for-loop and while-loop are equivalent, and changing for-loop to while-loop does not affect the result.

ST Strategies. In addition to modeling the semantic meaning of code snippets, capturing the structural features by learning the AST of each code snippet as a graph is commonly adopted in code representation learning [2, 75]. As ASTs represent syntactic meaning of code snippets, structure-level learning is equivalent to capturing *syntactic* information. Recently, graph contrastive learning has attracted considerable attention and achieves promising performance in various graph-based

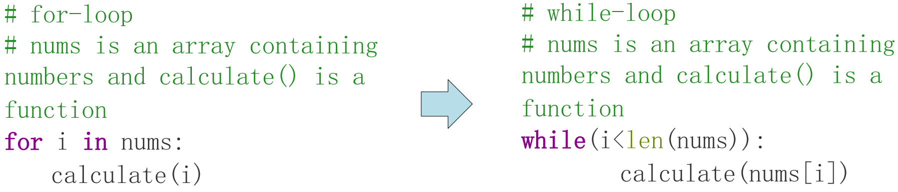

Fig. 4. An example of two equivalent loops.

applications [77]. Nevertheless, existing graph data augmentation strategies are not suitable for code representation learning. We design two graph transformation methods to generate structure-augmented data of the AST of each code snippet, which enhances the structure-level understanding of NACS:

— *Subtree Dropping*: Existing node dropping or edge dropping methods for augmenting graph data [84] randomly drop certain portions of vertices and/or edges in the graph. However, doing so will transfer an AST to a disconnected graph and significantly change the structural information of the AST. We opt to randomly discard one subtree of the AST, and the probability of discarding a subtree is inversely proportional to the number of nodes in the subtree.
— *Feature Shuffling*: We randomly exchange features of nodes in an AST. The motivation of this strategy is that, after the transformation, the new graph is still partially similar to the original AST (i.e., similar in structure but different in node features). The difference between the transformed graph and the original AST graph can help NACS better understand the intrinsic properties of the AST.

For a code snippet c of which the AST is denoted by x_c^q, three positive samples x_c^{p1}, x_c^{p2}, and x_c^{p3} are generated in each epoch of training. x_c^{p1} is constructed by randomly adopting two of the three PT strategies over c together, while x_c^{p2} and x_c^{p3} are generated by adopting two ST strategies individually. Since PT strategies require traversing and analyzing ASTs, they are costly, and we randomly adopt two of the three PT strategies together for each code snippet. Differently, the overhead of ST strategies IS relatively smaller. Subtree dropping involves AST traversal without the costly analysis. Feature shuffling does not require AST traversal. Thus, both ST strategies are adopted on each code snippet, but they are deployed independently to avoid causing a large deviation from the original code snippet.

3.2 Naming-Agnostic Contrastive Multi-View Code Representation Learning

To overcome the difficulty of aligning variables in different code snippets with same meaning, we design a naming-agnostic contrastive multi-view code representation learning component in NACS. The core idea is to strip information bound to variable names from input ASTs and focus on capturing intrinsic properties solely from their structures. To avoid information loss brought by the decoupling, in addition to the common modeling approach that models ASTs as graphs, we model ASTs through multi-view learning over both graph view and path view of ASTs, which helps enhance the understanding of NACS on AST structures. Furthermore, graph contrastive learning is adopted with the help of augmented data introduced in Section 3.1 to amplify supervision signals to alleviate the deficiency of manual labels. Figure 5 provides an overview of pre-training NACS via naming-agnostic contrastive multi-view code representation learning.

In the following, we will illustrate the graph-view modeling component and the path-view modeling component in NACS and how graph contrastive learning is used in each part.

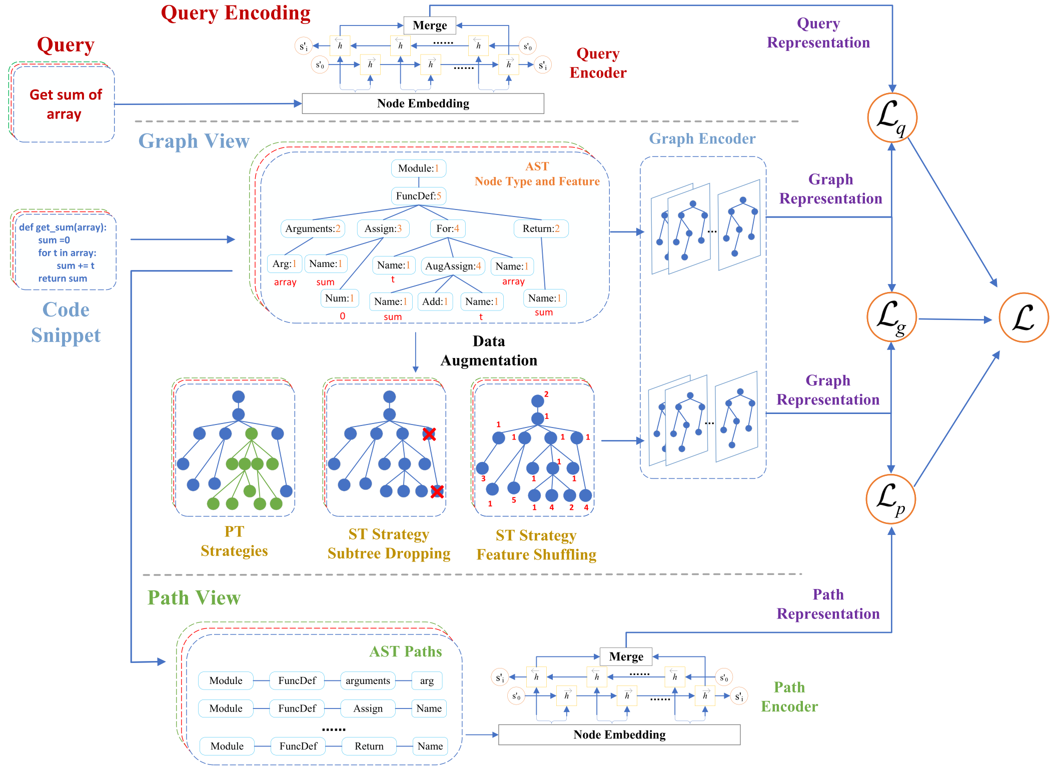

Fig. 5. Overview of pre-training NACS via naming-agnostic contrastive multi-view code representation learning.

3.2.1 Graph-View Modeling. Firstly, NACS captures the syntax-level (structural) information via modeling ASTs as graphs. This is the *core modeling component* in NACS. We treat each AST as a graph and pre-train a **Graph Topology Encoder (GT-Encoder)** to capture the topological information of ASTs (i.e., syntactic information of code snippets). The core of GT-Encoder is a GNN, and we leverage the **Graph Isomorphism Network (GIN)** [80]. But other GNN architectures can also be adopted. We adopt the AST graph discrimination task as the pre-training task for graph-view modeling. Through contrasting samples, NACS is trained to move an AST graph close to its positive samples but far away from its negative samples in the representation space, helping NACS distinguish AST graphs. The detailed steps are as follows:

(1) A mini-batch of N code snippets (and their ASTs) is randomly selected from the codebase. PT and ST strategies are applied on each code snippet to generate its three positive samples. This way, we have a total of $3N$ converted AST graphs.
(2) For each node s in an AST graph g, we encode its AST node type $\mathbf{b}_s^{\text{type}}$ and its degree $\mathbf{b}_s^{\text{degree}}$ (i.e., all dimensions of $\mathbf{b}_s^{\text{degree}}$ are the degree of s in the AST graph) as part of node features. Additionally, following the initialization method used in existing GNN pre-training approaches [62], we conduct eigen-decomposition on each AST graph's normalized graph Laplacian, *s.t.* $\mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\top}$, and then add the top eigenvectors of $\mathbf{U}$ to construct node features of s:

$$\mathbf{b}_s = \mathbf{b}_s^{\text{type}} \oplus \mathbf{b}_s^{\text{degree}} \oplus \mathbf{b}_s^{\text{lapla}}, \tag{1}$$

where $\mathbf{b}_s$ is node features of s, $\mathbf{b}_s^{lapla}$ is an eigenvector of $\mathbf{U}$ that corresponds to s, "$\oplus$" denotes the concatenation operation. Then, each AST is fed to a GIN encoder to generate the node representations:

$$\mathbf{h}_s^{(k)} = \mathrm{MLP}^{(k)}\left((1+\epsilon^{(k)}) \cdot \mathbf{h}_s^{(k-1)} + \sum_{u \in \mathcal{N}(s)} \mathbf{h}_u^{(k-1)}\right), \tag{2}$$

where $\mathrm{MLP}(\cdot)^{(k)}$ denotes the multi-layer percetron for the kth layer, $\mathbf{h}_s^{(k)}$ represents the output for the AST node s at the kth layer of GIN, $\mathcal{N}(s)$ is the set of neighbors of s, ϵ is a learnable parameter, and we set $\mathbf{h}_s^{(0)} = \mathbf{b}_s$. Then, we perform a mean pooling operation on representations of all nodes in g output by GIN. The result is fed to a **Multilayer Perceptron (MLP)** to get the graph representation $\mathbf{r}_g^{(i)}$ of g from the ith. The final graph representation $\mathbf{r}_g$ of the AST graph g is the sum of $\{\mathbf{r}_g^{(1)} \cdots \mathbf{r}_g^{(K)}\}$ where K is the number of layers:

$$\begin{aligned} \mathbf{r}_g^{(i)} &= \mathrm{MLP}\left(\frac{1}{M_g}\sum_{s=1}^{M_g} \mathbf{h}_s^{(i)}\right) \\ \mathbf{r}_g &= \sum_{i=1}^{K} \mathbf{r}_g^{(i)}, \end{aligned} \tag{3}$$

where M_g indicates the number of nodes in the AST graph g.

(3) We separately compute three contrastive losses based on the representation of the AST x_c^q of each code snippet c in the mini-batch and the representation of each positive sample (x_c^{p1}, x_c^{p2}, and x_c^{p3}). Taking x_c^{p1} as an example, the contrastive loss value is defined as follows:

$$l_g(x_c^q, x_c^{p1}) = \log \frac{e^{sim(\mathbf{r}_c^q, \mathbf{r}_c^{p1})/\tau}}{e^{sim\left(\mathbf{r}_c^q, \mathbf{r}_c^{p1}\right)/\tau} + \sum_{v=1}^{N} \mathbb{I}_{[v \neq c]} \left(e^{sim\,(\mathbf{r}_c^q, \mathbf{r}_v^q)/\tau} + e^{sim(\mathbf{r}_c^q, \mathbf{r}_v^{p1})/\tau}\right)}, \tag{4}$$

where $\mathbb{I}_{[v \neq c]} \in \{0, 1\}$ is the indicator function that equals 1 if $v \neq c$ otherwise 0, τ is a temperature parameter, and $sim(\mathbf{r}_1, \mathbf{r}_2)$ is the cosine similarity between $\mathbf{r}_1$ and $\mathbf{r}_2$. During optimization, we take other data samples in the same mini-batch as negative samples to avoid the long processing time. The overall graph-view contrastive learning loss in NACS is the sum of the three contrastive losses defined in Equation (4):

$$\mathcal{L}_g = \frac{1}{3N} \sum_{i=1}^{3} \sum_{c=1}^{N} l_g\left(x_c^q, x_c^{p_i}\right). \tag{5}$$

3.2.2 Path-View Modeling. In addition to the graph view, NACS models AST paths, providing another view of the structural information in ASTs. Path-view modeling is the *auxiliary modeling component* in NACS and it helps strengthen the understanding of ASTs.

For each AST x_c^q of a code snippet c, we extract all its paths from the root node to the leaf node $\mathcal{R} = \{y_{c,1}, y_{c,2}, \cdots, y_{c,z_c}\}$, where z_c represents the total number of paths in x_c^q. Since the numbers of paths in different ASTs differ, z_c is not fixed. $y_{c,i} = \{n_{c,i,1}, n_{c,i,2}, \cdots, n_{c,i,l_i}\}$, where l_i represents the path length of the AST path i and $n_{c,i,j}$ $(1 \leq j \leq l_i)$ is the AST node type of the jth node on the AST path i. We apply a Bi-LSTM, which can extract information in forward and reverse directions, to encode the AST path. For an AST path i in x_c^q, at each timestep s, the Bi-LSTM reads the embedding

of the sth node in i, then computes the hidden states $\mathbf{h}_{c,i,s}$:

$$\begin{aligned}\overrightarrow{\mathbf{h}}_{c,i,s} &= \overrightarrow{LSTM}\left(emb\left(n_{c,i,s}\right), \overrightarrow{\mathbf{h}}_{c,i,s-1}\right)\\ \overleftarrow{\mathbf{h}}_{c,i,s} &= \overleftarrow{LSTM}\left(emb\left(n_{c,i,s}\right), \overleftarrow{\mathbf{h}}_{c,i,s+1}\right),\end{aligned} \tag{6}$$

where $emb(n)$ is the embedding of the AST node type n. Note that, path-view modeling component and graph-view modeling component do not share AST node type representations, i.e., $emb(n)$ is not identical to $\mathbf{b}_s^{\text{type}}$ in Equation (1).

We concatenate the last hidden states in forward and backward directions to represent the AST path i:

$$\mathbf{h}_{c,i} = \overrightarrow{\mathbf{h}}_{c,i,l_i} \oplus \overleftarrow{\mathbf{h}}_{c,i,1}. \tag{7}$$

Then, we perform mean pooling on representations $\{\mathbf{h}_{c,1}, \cdots, \mathbf{h}_{c,g_c}\}$ of all AST paths in the AST x_c^q to obtain the final path representation $\mathbf{w}_c$ of x_c^q:

$$\mathbf{w}_c = \text{mean}\left(\{\mathbf{h}_{c,1}, \cdots, \mathbf{h}_{c,g_c}\}\right). \tag{8}$$

Taking one positive sample x_c^{p1} out of the three positive samples (x_c^{p1}, x_c^{p2}, and x_c^{p3}) for an AST x_c^q as an example, in path-view modeling, the contrastive loss of the positive pair $\left\langle x_c^q, x_c^{p1}\right\rangle$ is defined as follows:

$$l_p(x_c^q, x_c^{p1}) = \log \frac{e^{sim(\mathbf{r}_c^q, \mathbf{w}_c^{p1})/\tau}}{e^{sim\left(\mathbf{r}_c^q, \mathbf{w}_c^{p1}\right)/\tau} + \sum_{v=1}^{N} \mathbb{I}_{[v \neq c]} \left(e^{\,sim(\mathbf{r}_c^q, \mathbf{w}_v^q)/\tau} + e^{sim(\mathbf{r}_c^q, \mathbf{w}_v^{p1})/\tau}\right)}. \tag{9}$$

Equation (9) is similar to Equation (4). The difference is that, in Equation (9), NACS contrasts representations from graph-view and path-view, i.e., cross-view contrast. As a comparison, Equation (4) only contrasts graph-view representations.

The overall path-view loss function is defined as follows:

$$\mathcal{L}_p = \frac{1}{3N} \sum_{i=1}^{3} \sum_{c=1}^{N} l_p\left(x_c^q, x_c^{p_i}\right). \tag{10}$$

3.2.3 Query Encoding. Given a query $b_i = \{w_1, \ldots, w_t\}$ containing a sequence of t NL words. We use a Bi-LSTM to embed the query into its representation $\mathbf{s}_i$. The encoding procedure is similar to Equations (6)–(8).

Then, we adopt the InfoNCE loss [73], a type of contrastive loss function used for self-supervised learning, for modeling queries:

$$\mathcal{L}_q = -\frac{1}{M} \sum_{u=1}^{M} \log \frac{e^{sim\left(\mathbf{s}_u, \mathbf{r}_{(u)}\right)/\tau}}{\sum_{v=1}^{N} e^{sim\left(\mathbf{s}_u, \mathbf{r}'_{(v)}\right)/\tau}}, \tag{11}$$

where M is the number of the training queries, $\mathbf{r}_{(u)}$ is the AST graph representation of the ground-truth code snippet of the query u, and $\mathbf{r}'_{(v)}$ is the AST graph representation of the code snippet v in the codebase. $\mathbf{r}_{(u)}$ and $\mathbf{r}'_{(v)}$ are extracted according to Equation (3). Note that, for query encoding, we contrast query representations and AST graph representations. AST graph modeling is the primary modeling component in NACS. And we do not incorporate path representations (from the path view) in Equation (11) to reduce the overhead when encoding queries, as AST path encoding involves traversing multiple AST paths for each AST.

3.2.4 Putting All Together. The overall objective for optimizing the contrastive multi-view learning component in NACS is defined as:

$$\mathcal{L} = \mathcal{L}_g + \mathcal{L}_p + \mathcal{L}_q. \tag{12}$$

3.3 Enhance Code Search with NACS

Next, we describe how pre-trained NACS can be used to enhance code search.

3.3.1 Derive Code Representations. For each code snippet c in the codebase, we feed its code token sequence into the pre-trained CodeBERT [25] to obtain its code-token representation:

$$\mathbf{h}_c^{code} = CodeBERT_{code}(c). \tag{13}$$

CodeBERT is a bimodal model for NL and PL which enables high-quality text and code embeddings to be derived.

After that, the AST graph x_c^q of the code snippet c is fed to the graph encoder (Equation (3)) in the contrastive multi-view component of NACS to generate the AST representation of c:

$$\mathbf{h}_c^{AST} = NACS_{AST}(c). \tag{14}$$

Similar to the objective of query encoding defined in Equation (11), only the primary modeling method in NACS (i.e., graph-view modeling) is used at the search time to avoid the cost of traversing multiple AST paths so as to make the search phase lightweight. As the graph encoder is trained together with the path encoder in multi-view learning, it has been endowed with the ability to understand ASTs in multiple views.

In summary, we have a code-token representation $\mathbf{h}_c^{code}$ generated by the pre-trained CodeBert and an AST representation $\mathbf{h}_c^{AST}$ generated by the pre-trained NACS for each code snippet c.

3.3.2 Derive Query Representations. Each query q is fed into the pre-trained CodeBERT [25] and the query encoder of the pre-trained NACS to derive two query representations, respectively:

$$\begin{aligned} \bar{\mathbf{h}}_q^{query} &= CodeBERT_{query}(q) \\ \tilde{\mathbf{h}}_q^{query} &= NACS_{query}(q). \end{aligned} \tag{15}$$

As depicted in Figure 6, we calculate two matching scores for the query q and each code snippet c in the codebase:

$$\begin{aligned} score_1(q,c) &= sim(\mathbf{h}_c^{code},\ \bar{\mathbf{h}}_q^{query}) \\ score_2(q,c) &= sim(\mathbf{h}_c^{AST},\ \tilde{\mathbf{h}}_q^{query}), \end{aligned} \tag{16}$$

where $score_1(q,c)$ is the matching score from CodeBert and $score_2(q,c)$ indicates the matching score considering the multi-view information of x_c^q.

3.3.3 Fine-Tune NACS. Given code and query representations from the training data, we fine-tune CodeBert and NACS to enhance code search with multi-view representations. Figure 6 depicts this process. $score_1(q,c)$ and $score_2(q,c)$ are fused to indicate the matching score between a query q and a code snippet c:

$$score = score_1(q,c) + \lambda \cdot score_2(q,c), \tag{17}$$

where λ is a pre-defined parameter for balancing two parts.

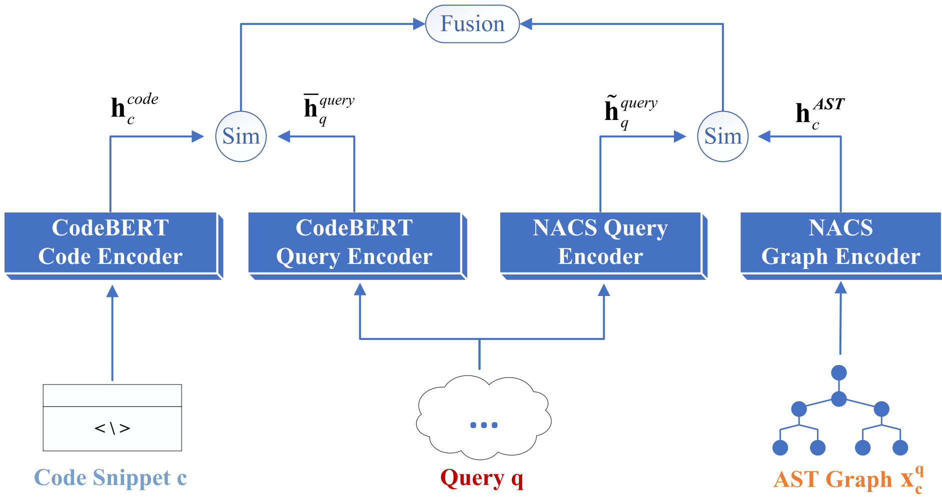

Fig. 6. Fine-tune NACS.

During optimization, given a mini-batch of N query-code training pairs, the overall framework is trained to minimize the following loss:

$$\mathcal{L}_{cs} = -\log \frac{e^{score^{(i,i)}/\tau}}{\sum_{v=0}^{N} e^{score^{(i,j)}/\tau}}. \tag{18}$$

3.3.4 Search Relevant Code Snippets with NACS. At search time, given an input query q, the relevance scores between q and each code snippet c in the codebase is computed as Equation (17). Relevant scores between q and all candidate code snippets are sorted and the top-K code snippets with highest relevance scores are returned as the search result for the query q.

This step is light-weight: the representations of all code snippets in the codebase can be pre-computed, and the two query representations (Equation (15)) can be directly retrieved from the fine-tuned CodeBert and NACS.

4 Experiments

In this section, we will report and analyze our experimental results to answer the following research questions:

— *RQ1*: How effective is NACS compared to state-of-the-art code search methods for the code search task? Is NACS able to achieve naming-agnostic code search? (Section 4.2)
— *RQ2*: How does each component in NACS contribute to the code search performance? (Section 4.3)
— *RQ3*: Can NACS be adopted to improve existing code search methods? (Section 4.4)
— *RQ4*: How do different hyper-parameter values affect the search result? (Section 4.5)

4.1 Experiment Settings

4.1.1 Data. We choose four public datasets that are commonly used in previous works in our experiments:

(1) *CodeSearchNet-Python Dataset*[2] [35]: It is the Python dataset used in the CodeSearchNet challenge. It contains 0.5 M Python functions and their corresponding NL descriptions. CodeSearchNet dataset is widely used in various code relevant tasks including but not limited to code representation learning [25, 29], code summarization [44], and code completion [15].
(2) *CodeSearchNet-Java Dataset*[3] [35]: It is the Java dataset used in the CodeSearchNet challenge. It contains 0.5 M Java functions and their corresponding NL descriptions.
(3) *CoSQA Dataset*[4] [34]: It is the dataset released by Microsoft. It contains 20,604 labeled query-code pairs. Each query is written in English while each code snippet is a Python code snippet. The data is annotated by at least three human annotators. The queries come from the search logs of the Microsoft Bing search engine, and each code snippet is a Python function crawled from GitHub.
(4) *CoSQA-Var Dataset*: To investigate the performance of different code search models and NACS when handling the naming issue, we randomly replace variable names in CoSQA dataset and construct a new dataset CoSQA-Var. For a variable in a code snippet of CoSQA, we randomly replace its variable name with another variable name used in CoSQA. The random replacement keeps the consistency of variable names: after the replacement, a variable will still use the same (new) variable name within one code snippet. The CoSQA-Var contains the same number of query-code pairs as the CoSQA dataset.

4.1.2 Evaluation Metric. To evaluate the performance, we adopt **Mean Reciprocal Rank (MRR)**, the most widely used evaluation metric for code search. MRR is the average of the reciprocal ranks of results for test queries:

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}, \tag{19}$$

where $|Q|$ is the number of queries and $rank_i$ indicates the rank of the ground-truth code snippet w.r.t. the ith query.

4.1.3 Baselines. We use seven prevalent code search models as baselines in our experiments:

(1) *DeepCS*[5] [27]: DeepCS is the pioneering code search method that adopts deep learning. DeepCS considers three aspects of source code: the method name, the API invocation sequence, and the tokens contained in the source code. The camel split tokens of methods names and the API invocation sequences are embedded by two RNNs, respectively. Code tokens are embedded via an MLP. The three representation vectors are fused into the code representation vector via a fully connected layer. Query tokens are embedded using an RNN. Finally. the matching score between a query and a code snippet is measured by the cosine similarity between the code representation vector and the query representation vector.
(2) *OCoR*[6] [90]: OCoR is proposed with a specific focus on the overlaps between identifiers in code and words in queries. OCoR uses two specifically designed components to capture overlaps: the first embeds names by characters to capture the overlaps between names, and the second introduces an overlap matrix to represent the degrees of overlaps between each NL word and each identifier.

[2]https://github.com/github/CodeSearchNet.
[3]https://huggingface.co/datasets/Nan-Do/code-search-net-java.
[4]https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/NL-code-search-WebQuery.
[5]https://github.com/guxd/deep-code-search.
[6]https://github.com/pkuzqh/OCoR.

(3) *CSRS*[7] [14]: CSRS considers both relevance matching and semantic matching for code search. It contains a relevance matching module that measures lexical matching signals between the query and the code snippets, and a co-attention-based semantic matching module to capture the semantic correlation between the query and the code snippets.
(4) *CoCLR*[8] [34]: CoCLR adopts CodeBERT to encode queries and code snippets into representations that are further fed into an MLP to measure query-code relevance.
(5) *CodeBERT*[9] [25]: CodeBERT is a bimodal pre-trained model for PL and NL. CodeBERT shares the same architecture as RoBERTa [51] and it is pre-trained via the masked language modeling task [19] and the replaced token detection task on the CodeSearchNet dataset.
(6) *GraphCodeBERT*[9] [29]: GraphCodeBERT is a pre-trained model for PL and NL. GraphCodeBERT adopts the idea of BERT [19] and uses dataflow which is the semantic-level structure that encoders the dataflow relations between variables. It is pre-trained via the masked language modeling task, the code structure prediction task and the alignment task between code and structure on the CodeSearchNet dataset.
(7) *DGMS*[10] [47]: DGMS is an end-to-end deep graph matching and searching model based on GNNs for code search. DGMS represents both queries and code snippets with the unified graph-structured data, and then use the proposed graph matching and searching model to retrieve the best matching code snippet.

4.1.4 Implementation Details. For Python datasets CodeSearchNet-Java, CoSQA, and CoSQA-Var, we extract ASTs and conduct data augmentation for NACS using Python official AST library.[11] For Java dataset CodeSearchNet-Java, we extract ASTs and conduct data augmentation for NACS using JavaParser library.[12] We use the implementations from authors for baselines. We implement NACS using PyTorch. The implementation and datasets are available at https://github.com/KDEGroup/NACS.

The experiments were run on a machine with two Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50 GHz, 256 GB main memory, and 8 GeForce RTX 2080 Ti graphics cards with 11 GB memory per card. During running, each program will monopolize one graphics card even if it does not require the complete 11 GB video memory. Performance is reported based on the average of five runs of models.

We set batch size to 48. The dimensions of AST path node embedding, AST path embedding, and query embedding are 128. The dimensions of AST graph node embedding and AST graph embedding are 128. Adam optimizer [41] is used with an initial learning rate of 0.001. NACS and baselines use same settings whenever it is possible. Otherwise, the default settings for baselines are adopted to achieve a fair comparison. We conduct a grid search to find best settings for λ and τ for NACS. We find that $\lambda = 0.0001$ and $\tau = 0.07$ bring best results and we use them as the default setting. In Section 4.5, we report the impacts of different values of λ and τ on NACS.

4.2 Analysis of Code Search Performance (RQ1)

4.2.1 Overall Performance. Firstly, we investigate the overall code search performance of all methods on CodeSearchNet-Python, CodeSearchNet-Java, CoSQA, and CoSQA-Var datasets. Table 1 shows the MRR scores of each code search model. From the results, we have the following observations:

[7] https://github.com/css518/CSRS.
[8] https://github.com/Jun-jie-Huang/CoCLR.
[9] https://github.com/microsoft/CodeBERT.
[10] https://github.com/ryderling/DGMS.
[11] https://docs.python.org/3/library/ast.html.
[12] https://javaparser.org.

Table 1. MRR Scores of Each Method on the Four Datasets

Model	CodeSearchNet-Python	CodeSearchNet-Java	CoSQA	CoSQA-Var
OCoR	0.165	0.201	0.352	0.293
CSRS	0.231	0.386	0.373	0.231
DeepCS	0.164	0.199	0.472	0.433
CodeBERT	0.665	0.667	0.652	0.558
GraphCodeBERT	0.694	0.689	0.648	0.456
CoCLR	0.637	0.631	0.647	0.515
DGMS	0.579	0.555	0.471	0.399
NACS	0.701	0.705	0.708	0.704

(1) Compared to OCoR, CSRS, DeepCS, and DGMS, pre-training-based methods CodeBERT, GraphCodeBERT, CoCLR, and NACS achieve much better performance on all the four datasets, showing the effectiveness of the Bert-style pre-training method on improving code search performance.
(2) Compared to all baselines, NACS achieves highest MRR scores, and it consistently outperforms the best two baselines GraphCodeBert and CoCLR on the four datasets, showing the robustness of NACS.
(3) The CoSQA-Var dataset is designed for evaluating the performance of code search models when variables with same meaning have different variable names in different code snippets. From Table 1, we can see that the performance of all baselines significantly degrades on the CoSQA-Var dataset (↓ 8.26–38.07%) compared to their performance on the CoSQA dataset. Differently, the performance of NACS is not affected by changing variable names in code snippets. This observation has verified the effectiveness of our proposed contrastive multi-view code representation learning: capturing the structural information in ASTs without explicitly binding code tokens to specific pre-trained representations can help overcome the problem of variable naming when modeling code snippets.
(4) NACS performs similarly on both CodeSearchNet-Python and CodeSearchNet-Java and it consistently outperforms baselines, showing the effectiveness of NACS on both Python and Java.

In summary, NACS consistently shows superior performance than all baselines. The gap is more significant on CoSQA-Var, showing the effectiveness of NACS on handling the naming issue.

4.2.2 Running Time. We report the running time of each methods in Table 2. Since CodeSearchNet-Python and CodeSearchNet-Java have similar amounts of data and CoSQA and CoSQA-Var have the same amount of data, we only provide the training and the test time on CodeSearchNet-Java and CoSQA in Table 2. From the results, we have the following observations:

(1) OCoR has the longest running time. The reason is that, during training, it counts the overlaps between identifiers in code and words in queries, which is a costly operation.
(2) Pre-training-based methods CodeBERT, GraphCodeBERT, CoCLR, and NACS do not require much more training time compared to small models CSRS and DGMS. And their training time is even shorter in some cases. We can also observe that the test time of pre-training methods are generally much shorter compared to OCoR, CSRS, DeepCS, and DGMS, showing that pre-training methods are efficient during inference and do not affect user experience on code search.
(3) The running time of NACS is at the same level as the running time of other pre-training methods. On the largest dataset CodeSearchNet-Java (it is 25 times larger than CoSQA), its

Table 2. Running Time of Each Method on CodeSearchNet-Java and CoSQA

Model	CodeSearchNet-Java		CoSQA	
	Train	Test	Train	Test
OCoR	284.968 hours	4.017 hours	742.260 min	261.901 sec
CSRS	14.638 hours	6.445 hours	37.254 min	432.371 sec
DeepCS	75.762 hours	10.932 hours	195.437 min	737.075 sec
CodeBERT	15.962 hours	2,028.199 sec	41.611 min	8.815 sec
GraphCodeBERT	33.977 hours	284.899 sec	87.956 min	10.715 sec
CoCLR	55.364 hours	2,583.092 sec	109.104 min	46.374 sec
DGMS	45.191 hours	1,553.489 sec	115.999 min	12.742 sec
NACS	18.740 hours	531.692 sec	48.529 min	34.658 sec

running time is acceptable compared to other baselines, showing the scalability of NACS on large codebases.

Overall, the running overhead of NACS is at the same level as other pre-training-based methods and it shows strong scalability on large codebase.

4.2.3 Case Studies of Code Search Results. We provide two types of case studies of code search results to further illustrate that the effectiveness of NACS on handling same variables with different variable names (i.e., NACS).

Table 3 illustrates the ranking results of ground-truth code snippets w.r.t. two queries using NACS, CoCLR, GraphCodeBERT, DeepCS, and DGMS on both CoSQA and CoSQA-Var datasets. We show the ranking position of the ground-truth code snippet using each method. From the results, we can observe that:

(1) The accuracy of CoCLR, GraphCodeBert, DeepCS, and DGMS are affected when variable names are randomly replaced, and they provide worse results for the same query on the CoSQA-Var dataset compared to their original search results on the CoSQA dataset. Differently, NACS provides consistent ranking results for the ground-truth code snippet w.r.t. the same query on the two datasets, showing that the performance of NACS is not affected by the change of variable names. In other words, NACS is able to achieve naming-agnostic code search.
(2) For cases where different code search methods have comparable performance (e.g., the second case in Table 3 where both NACS and CoCLR rank the ground-truth code snippet as the top-1 search result on the CoSQA dataset), randomly replacing variable names in the data can be used to further identify the performance gap between different methods and the deficiency of existing methods (e.g., the performance difference between NACS and CoCLR on the CoSQA-Var dataset is distinguishable in the second case of Table 3).

Table 4 depicts the ranking results of ground-truth code snippets w.r.t. two queries using NACS, CoCLR, GraphCodeBERT, DeepCS, and DGMS on the CoSQA dataset. We show the top-2 retrieved code snippet results from each method and the ground-truth code snippet is shown in bold. We can see that:

(1) NACS precisely ranks ground-truth code snippets first, while the other four baselines cannot consistently rank ground-truth code snippets as the top-1 results. The observation illustrates that NACS exceeds existing code search methods in general cases.
(2) Moreover, top-1 results provided by CoCLR, GraphCodeBERT, and DeepCS for the first case in Table 4 all contain the code token "sort," which is part of the input query. The ground-truth

Table 3. Case Study 1: Ranking Results of Ground-Truth Code Snippets w.r.t. Two Queries Using NACS, CoCLR, GraphCodeBERT, DeepCS, and DGMS on Both CoSQA and CoSQA-Var Datasets

Query	Dataset	Code	NACS	CoCLR	Graph CodeBERT	DeepCS	DGMS
timestamp remove timezone	CoSQA	def fromtimestamp(cls, timestamp): 'Returns a datetime object of a given timestamp (in local tz).' d = cls.utcfromtimestamp(timestamp) return d.astimezone(localtz())	1	14	2	12	1
	CoSQA-Var	def fromtimestamp(events_sort_key,new_translated_file): 'Returns a datetime object of a given timestamp (in local tz).' base_content_type = events_sort_key.utcfromtimestamp(new_translated_file) return base_content_type.astimezone(localtz())	1	18	28	13	4
test if value is ctypes array	CoSQA	def is_integer_array(val): 'Checks whether a variable is a numpy integer array.' return is_np_array(val) and issubclass(val.dtype.type, np.integer)	1	1	4	42	112
	CoSQA-Var	def is_integer_array(seg_i): 'Checks whether a variable is a numpy integer array.' return (is_np_array(seg_i) and issubclass(seg_i.dtype.type, np.integer))	1	12	6	175	230

code snippet does not contain the code token "sort," which is possibly a reason why the three baselines cannot correctly rank the ground-truth code snippet. Hence, the three baselines emphasize on the overlap between code snippets and queries, meaning that they may only capture the token-level information and cannot fully understand the deep semantics of code snippets. Differently, NACS can correctly find the ground-truth code snippet even if it does not contain code tokens that appear in queries.

4.2.4 Relations between Learned Query Space and Code Space. Code search models aim at matching a query and the ground-truth code snippet. The similarity between the query embedding and the code embedding can somehow demonstrate the ability of a code search model to match queries and corresponding ground-truth code snippets. We use the average embeddings (**t**) of tokens in a query to represent the query and use the average embeddings (**c**) of code tokens in the corresponding ground-truth code snippet to indicate the ground-truth code snippet. We calculate the average cosine similarity between **t** and **c** and visualize the results of GraphCodeBERT, CoCLR, and NACS on CoSQA in Figure 7. From Figure 7, we can see that the average similarity between **t** and **c** for NACS is highest, showing that NACS is better at matching a query and the ground-truth code snippet.

4.3 Contributions of Each Component in NACS (RQ2)

To investigate the impact of each component in NACS, we conduct an ablation study to analyze whether each major component in NACS contributes to the overall performance of NACS. We have designed several variations of NACS and tested them on the CoSQA dataset:

- $NACS_g$: This variation does not consider contrasting graph view of ASTs shown in Equation (4), and the pre-training loss in Equation (12) does not include $\mathcal{L}_g$.
- $NACS_p$: This variation does not adopt the cross-view contrast (i.e., contrast path view and graph view) shown in Equation (9), and the pre-training loss in Equation (12) does not contain $\mathcal{L}_p$.
- $NACS_a$: This variation removes all AST-related components (graph-view contrast and cross-view contrast) and it is degraded to CodeBERT.

Table 4. Case Study 2: Ranking Results of Ground-Truth Code Snippets w.r.t. Two Queries Using NACS, CoCLR, GraphCodeBERT, DeepCS, and DGMS on the CoSQA Dataset

Query	Model	Rank 1st	Rank 2nd

using sort to move element in to new position in list	NACS	def insort_no_dup(lst, item):
		"""If item is not in lst, add item to list at its sorted position"""
		import bisect
		ix = bisect.bisect_left(lst, item)
		if lst[ix] != item:
		lst[ix:ix] = [item]
			def list_move_to_front(l,value='other'):
			""" if the value is in the list, move it to the front and return it. """
			l=list(l)
			if value in l:
			l.remove(value)
			l.insert(0,value)
			return l

	CoCLR	def __sort_up(self):
		""" Sort the updatable objects according to ascending Order"""
		if self.__do_need_sort_up:
		self.__up_objects.sort(key=cmp_to_key(self.__up_cmp))
		self.__do_need_sort_up = False
			def insort_no_dup(lst, item):
			"""If item is not in lst, add item to list at its sorted position"""
			import bisect
			ix = bisect.bisect_left(lst, item)
			if lst[ix] != item:
			lst[ix:ix] = [item]

	GraphCodeBERT	def __sort_up(self):
		""" Sort the updatable objects according to ascending Order"""
		if self.__do_need_sort_up:
		self.__up_objects.sort(key=cmp_to_key(self.__up_cmp))
		self.__do_need_sort_up = False
			def natural_sort(list_to_sort: Iterable[str]) -> List[str]:
			"""Sorts a list of strings case insensitively as well as numerically.
			"""
			return sorted(list_to_sort, key=natural_keys)

	DeepCS	def _sort_r(sorted, processed, key, deps, dependency_tree):
		if key in processed:
		return
		processed.add(key)
		for dep_key in deps:
		dep_deps = dependency_tree.get(dep_key)
		if dep_deps is None:
		log.debug('"%s" not found, skipped', Repr(dep_key))
		continue
		_sort_r(sorted, processed, dep_key, dep_deps, dependency_tree)
		sorted.append((key, deps))
			def sort_data(x, y):
			"""Sort the data."""
			xy = sorted(zip(x, y))
			x, y = zip(*xy)
			return x, y

	DGMS	def insort_no_dup(lst, item):
		"""If item is not in lst, add item to list at its sorted position"""
		import bisect
		ix = bisect.bisect_left(lst, item)
		if lst[ix] != item:
		lst[ix:ix] = [item]
			def equal(list1, list2):
			""" takes flags returns indexes of True values """
			return [item1 == item2 for item1, item2 in broadcast_zip(list1, list2)]

how to product of a list in python	NACS	def dotproduct(X, Y):
		"""Return the sum of the element-wise product of vectors x and y.
		"""
		return sum([x * y for x, y in zip(X, Y)])
			def _cumprod(l):
			"""Cumulative product of a list."""
			ret = [1]
			for item in l:
			ret.append(ret[-1] * item)
			return ret

	CoCLR	def combinations(l):
		"""Pure-Python implementation of itertools combinations(l, 2)."""
		result = []
		for x in xrange(len(l) - 1):
		ls = l[x + 1:]
		for y in ls:
		result.append((l[x], y))
		return result
			def _cumprod(l):
			"""Cumulative product of a list."""
			ret = [1]
			for item in l:
			ret.append(ret[-1] * item)
			return ret

	GraphCodeBERT	def _cumprod(l):
		"""Cumulative product of a list."""
		ret = [1]
		for item in l:
		ret.append(ret[-1] * item)
		return ret
			def circ_permutation(items):
			"""Calculate the circular permutation for a given list of items."""
			permutations = []
			for i in range(len(items)):
			permutations.append(items[i:] + items[:i])
			return permutations

	DeepCS	def _cumprod(l):
		"""Cumulative product of a list."""
		ret = [1]
		for item in l:
		ret.append(ret[-1] * item)
		return ret
			def multiply(traj):
			"""Sophisticated simulation of multiplication"""
			z=traj.x*traj.y
			traj.f_add_result('z',z=z, comment='I am the product of two reals!')

	DGMS	def factors(n): """Computes all the integer factors of the number 'n'""" return set(reduce(list.__add__, ([i, n // i] for i in range(1, int(n ** 0.5) + 1) if n % i == 0)))	def dotproduct(X, Y): """Return the sum of the element-wise product of vectors x and y. """ return sum([x * y for x, y in zip(X, Y)])

Ground-truth code snippets are highlighted in yellow.

Table 5 reports the result of the ablation study. From the result, we can see that:

(1) $NACS_a$ (i.e., CodeBERT) shows the worst performance among all NACS variations, which indicates that the superior performance of NACS comes from adopting the naming-agnostic contrastive multi-view code representation learning instead of CodeBERT.

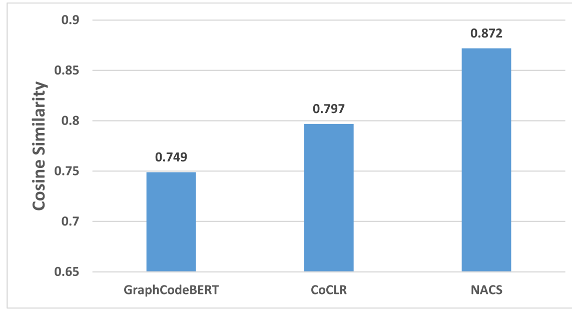

Fig. 7. Cosine similarity between query and corresponding ground-truth code snippet on the CoSQA dataset.

Table 5. Performance of Different Variations of NACS on the CoSQA Dataset

Variations	MRR Score
NACS	0.708
NACS_g	0.682
NACS_p	0.687
NACS_a	0.652

(2) NACS_g and NACS_p show better code search performance than NACS_a, showing that using graph-view contrast (NACS_g) or cross-view contrast (NACS_p) alone can improve the quality of code search. In other words, contrastive pre-training based on the structural information of ASTs can help enhance the code search model.
(3) NACS shows even better performance than NACS_g and NACS_p, demonstrating that graph-view contrast and cross-view contrast can be used together to help the code search model alleviate the impact of different naming styles and achieve high-quality code search results.

4.4 Effectiveness of NACS on Improving Other Code Search Models (RQ3)

NACS can be easily adopted to improve existing code search models: we can adopt any existing deep learning-based code search model to generate representations for candidate code snippets and queries as a replacement of CodeBERT used in Sections 3.3.1 and 3.3.2.

Table 6 reports the results of using NACS to enhance DeepCS, CoCLR, and GraphCodeBERT. In Table 6, the superscript "*" indicates the corresponding method is enhanced by NACS. By comparing results of one code search method and its improved version in Table 6, we can find that NACS indeed improves the search performance of existing methods. Furthermore, the performance of DeepCS, GraphCodeBert, and CoCLR on CoSQA-Var is much worse than their performance on CoSQA, and the decline percentages are 8.26%, 42.11%, and 20.40%, showing that existing code search methods suffer from the naming issue. As a comparison, the performance decline is unremarkable for DeepCS* (3.07%), GraphCodeBert* (2.32%), and CoCLR* (2.95%). Therefore, we can conclude that NACS can be adopted to improve the robustness of existing code search methods and help them alleviate the negative impact of different naming styles used in code snippets.

4.5 Impacts of Hyper-Parameters on Search Results (RQ4)

We investigate the impacts of λ in Equation (17) on the performance of NACS. Figure 8 illustrates the performance of NACS on CoSQA and CoSQA-Var datasets when λ is set to 0, 0.0001, 0.001, 0.01, and 0.1, respectively. We can observe that the best results are achieved when λ is set to 0.0001 and

Table 6. Performance of Existing Code Search Methods Enhanced by NACS

Method	CoSQA	CoSQA-Var	Decline Percentage
DeepCS	0.472	0.433	↓8.26%
DeepCS*	0.488	0.473	↓3.07%
GraphCodeBERT	0.648	0.456	↓42.11%
GraphCodeBERT*	0.691	0.675	↓2.32%
CoCLR	0.647	0.515	↓20.40%
CoCLR*	0.677	0.657	↓2.95%

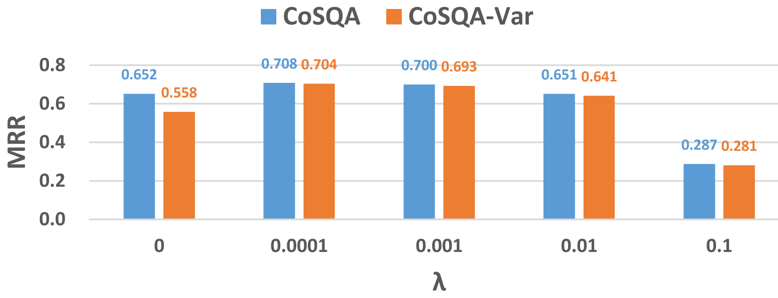

Fig. 8. Impacts of λ on the performance of NACS.

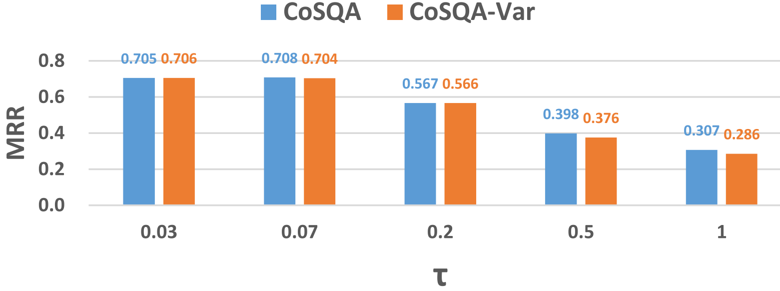

Fig. 9. Impacts of τ on the performance of NACS.

larger λ (> 0.1) will significantly degrade the performance of NACS. Overall, it is easy to tune λ as the best value falls in a relative narrow range. When λ is set to 0, NACS is identical to CodeBERT. The performance of NACS declines from 0.708/0.704 to 0.652/0.558 when λ is changed from 0.0001 to 0. The noticeable performance reduction shows that our proposed naming-agnostic contrastive multi-view code representation learning indeed improve the performance of code search.

We further report the influence of the temperature parameter τ. Figure 9 provides the performance of NACS on CoSQA and CoSQA-Var datasets when varying τ in 0.03, 0.07, 0.2, 0.5, and 1. Best performance is achieved when setting τ to 0.03 (for CoSQA-Var) and 0.07 (for CoSQA). As τ continues to increase, the performance of NACS declines. The impacts of τ on NACS show it is easy to tune and find a suitable setting for τ.

Considering the results in Figures 8 and 9, we can again confirm our conclusion obtained in Section 4.2: NACS is almost invariable to different naming conventions since its performance on CoSQA and CoSQA-Var datasets, when using same hyper-parameters, is close.

5 Conclusion

Code search provides a way for developers to reuse implementations and learn new API usage, reducing developers' cognitive burden. Due to the pivotal role of code search in software development, much effort has been devoted to improving the quality of code search. Particularly, there is a surge of works on deploying deep learning techniques to enhance code search. Nevertheless, most existing deep learning-based code search methods ignore the impact of different naming conventions, causing downgraded performance when the same variable has different variable names in different code snippets.

In this article, we propose a naming-agnostic code search method NACS by using our designed contrastive multi-view code representation learning. NACS strips information bound to variable names from AST and focuses on capturing intrinsic properties solely from AST structures. We use semantic-level and syntax-level augmentation techniques to prepare realistically rational data and adopt contrastive learning to design a graph-view modeling component in NACS to enhance the understanding of code snippets. ASTs are further modeled from a path view to strengthen the graph-view modeling component via multi-view learning. Extensive experiments show that NACS outperforms baselines, and it can also be adapted to help existing code search methods overcome the impact of different naming conventions.

In the future, we plan to explore other designs of lightweight architectures to reduce both the training code and the search cost of NACS to reduce the cost of code search and further improve the scalability of NACS. We also plan to collect more code search data containing real queries and corresponding code snippets written in more PLs to construct large-scale evaluation benchmarks for better evaluating the code search task.

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv* 51, 4 (2018), 81:1–81:37.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *ICLR*. Retrieved from https://openreview.net/pdf?id=BJOFETxR

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *ICLR*. Retrieved from https://openreview.net/pdf?id=H1gKYo09tX

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang*. 3, POPL, 40 (2019), 1–29.

[5] Sushil Krishna Bajracharya, Trung Chi Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Videira Lopes. 2006. Sourcerer: A search engine for open source code supporting structure-based search. In *OOPSLA Companion*, 681–682.

[6] Sushil Krishna Bajracharya, Joel Ossher, and Cristina Videira Lopes. 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In *SIGSOFT FSE*, 157–166.

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *NeurIPS*, 3589–3601.

[8] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *CHI*, 1589–1598.

[9] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw*. 30, 1–7 (1998), 107–117.

[10] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative code modeling with graphs. In *ICLR*. Retrieved from https://openreview.net/pdf?id=Bke4KsA5FX

[11] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *SIGIR*, 511–521.

[12] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. arXiv:2006.12641. Retrieved from https://arxiv.org/abs/2006.12641.

[13] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-domain deep code search with meta learning. In *ICSE*, 487–498.

[14] Yi Cheng and Li Kuang. 2022. CSRS: Code search with relevance matching and semantic matching. In *ICPC*, 533–542.

[15] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An empirical study on the usage of BERT models for code completion. In *MSR*, 108–119.

[16] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *ICLR*. Retrieved from https://openreview.net/pdf?id=r1xMH1BtvB

[17] Fabio Crestani. 1997. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev* 11, 6 (1997), 453–482.

[18] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open vocabulary learning on source code with a graph-structured cache. In *ICML*, Vol. 97, 1475–1485.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 4171–4186.

[20] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. 2017. Adversarial feature learning. In *ICLR*. Retrieved from https://openreview.net/pdf?id=BJtNZAFgg

[21] Qi Dou, Hao Chen, Lequan Yu, Jing Qin, and Pheng-Ann Heng. 2017. Multilevel contextual 3-D CNNs for false positive reduction in pulmonary nodule detection. *IEEE Trans. Bio-Med. Eng*. 64, 7 (2017), 1558–1567.

[22] Shaohua Fan, Xiao Wang, Chuan Shi, Emiao Lu, Ken Lin, and Bai Wang. 2020. One2Multi graph autoencoder for multi-view graph clustering. In *WWW*, 3070–3076.

[23] Fangxiang Feng, Xiaojie Wang, and Ruifan Li. 2014. Cross-modal retrieval with correspondence autoencoder. In *ACM Multimedia*, 7–16.

[24] Yifan Feng, Zizhao Zhang, Xibin Zhao, Rongrong Ji, and Yue Gao. 2018. GVCNN: Group-view convolutional neural networks for 3D shape recognition. In *CVPR*, 264–272.

[25] 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 *EMNLP (Findings)*, Vol. EMNLP 2020, 1536–1547.

[26] Jaroslav M. Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2016. TASSAL: Autofolding for source code summarization. In *ICSE (Companion Volume)*, 649–652.

[27] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*, 933–944.

[28] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *ACL (1)*, 7212–7225.

[29] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *ICLR*. Retrieved from https://openreview.net/pdf?id=jLoC4ez43PZ

[30] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press.

[31] Kaveh Hassani and Amir Hosein Khas Ahmadi. 2020. Contrastive multi-view representation learning on graphs. In *ICML*, Vol. 119, 4116–4126.

[32] R. Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Philip Bachman, Adam Trischler, and Yoshua Bengio. 2019. Learning deep representations by mutual information estimation and maximization. In *ICLR*. Retrieved from https://openreview.net/pdf?id=Bklr3j0cKX

[33] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay S. Pande, and Jure Leskovec. 2020. Strategies for pre-training graph neural networks. In *ICLR*. Retrieved from https://openreview.net/pdf?id=HJlWWJSFDH

[34] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ web queries for code search and question answering. In *ACL/IJCNLP (1)*, 5690–5700.

[35] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436. Retrieved from https://arxiv.org/abs/1909.09436

[36] Vinoj Jayasundara, Nghi Duy Quoc Bui, Lingxiao Jiang, and David Lo. 2019. TreeCaps: Tree-structured capsule networks for program source code processing. arXiv:1910.12306. Retrieved from https://arxiv.org/abs/1910.12306

[37] Yizhu Jiao, Yun Xiong, Jiawei Zhang, Yao Zhang, Tianqi Zhang, and Yangyong Zhu. 2020. Sub-graph contrast for scalable self-supervised graph representation learning. In *ICDM*, 222–231.

[38] Wei Jin, Tyler Derr, Haochen Liu, Yiqi Wang, Suhang Wang, Zitao Liu, and Jiliang Tang. 2020. Self-supervised learning on graphs: Deep insights and new direction. arXiv:2006.10141. Retrieved from https://arxiv.org/abs/2006.10141

[39] Wei Jin, Tyler Derr, Yiqi Wang, Yao Ma, Zitao Liu, and Jiliang Tang. 2021. Node similarity preserving graph convolutional networks. In *WSDM*, 148–156.

[40] Andrej Karpathy and Li Fei-Fei. 2017. Deep visual-semantic alignments for generating image descriptions. *IEEE Trans. Pattern Anal. Mach. Intell*. 39, 4 (2017), 664–676.

[41] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*. Retrieved from https://arxiv.org/abs/1412.6980

[42] Triet Huynh, Minh Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Comput. Surv* 53, 3 (2020), 62:1–62:38.
[43] Yingming Li, Ming Yang, and Zhongfei Zhang. 2019. A survey of multi-view representation learning. *IEEE Trans. Knowl. Data Eng*. 31, 10 (2019), 1863–1883.
[44] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving code summarization with block-wise abstract syntax tree splitting. In *ICPC*, 184–195.
[45] Yijie Lin, Yuanbiao Gou, Xiaotian Liu, Jinfeng Bai, Jiancheng Lv, and Xi Peng. 2023. Dual contrastive prediction for incomplete multi-view representation learning. *IEEE Trans. Pattern Anal. Mach. Intell*. 45, 4 (2023), 4447–4461.
[46] Yijie Lin, Yuanbiao Gou, Zitao Liu, Boyun Li, Jiancheng Lv, and Xi Peng. 2021. COMPLETER: Incomplete multi-view clustering via contrastive prediction. In *CVPR*, 11174–11183.
[47] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2021. Deep graph matching and searching for semantic code retrieval. *ACM Trans. Knowl. Discov. Data* 15, 5 (2021), 1–21.
[48] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. 2022. Opportunities and challenges in code search tools. *ACM Comput. Surv*. 54, 9 (2022), 1–40.
[49] Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. RefBERT: A two-stage pre-trained framework for automatic rename refactoring. In *ISSTA*, 740–752.
[50] Xiao Liu, Fanjin Zhang, Zhenyu Hou, Zhaoyu Wang, Li Mian, Jing Zhang, and Jie Tang. 2021. Self-supervised learning: Generative or contrastive. *IEEE Trans. Knowl. Data Eng*., 1–1.
[51] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. arXiv:1907.11692. Retrieved from https://arxiv.org/abs/1907.11692
[52] Yixin Liu, Shirui Pan, Ming Jin, Chuan Zhou, Feng Xia, and Philip S. Yu. 2023. Graph self-supervised learning: A survey. *IEEE Trans. Neural Networks Learn. Syst*. 35, 6 (2023), 5879–5900.
[53] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective code search based on API understanding and extended Boolean model (E). In *ASE*, 260–270.
[54] Hehuan Ma, Yatao Bian, Yu Rong, Wenbing Huang, Tingyang Xu, Weiyang Xie, Geyan Ye, and Junzhou Huang. 2020. Dual message passing neural network for molecular property prediction. arXiv:2005.13607. Retrieved from https://arxiv.org/abs/2005.13607
[55] Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, and Alan L. Yuille. 2015. Deep captioning with multimodal recurrent neural networks (m-RNN). In *ICLR*. Retrieved from https://arxiv.org/abs/1412.6632
[56] Costas Mavromatis and George Karypis. 2021. Graph InfoClust: Maximizing coarse-grain mutual information in graphs. In *PAKDD (1)*, Vol. 12712, 541–553.
[57] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: Finding relevant functions and their usage. In *ICSE*, 111–120.
[58] Collin McMillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol*. 22, 4 (2013), 1–30.
[59] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y. Ng. 2011. Multimodal deep learning. In *ICML*, 689–696.
[60] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could neural networks understand programs? In *ICML. PMLR*, Vol. 139, 8476–8486.
[61] Zhen Peng, Yixiang Dong, Minnan Luo, Xiao-Ming Wu, and Qinghua Zheng. 2020. Self-supervised graph representation learning via global context prediction. arXiv:2003.01604. Retrieved from https://arxiv.org/abs/2003.01604
[62] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. GCC: Graph contrastive coding for graph neural network pre-training. In *KDD*, 1150–1160.
[63] Yuxiang Ren, Bo Liu, Chao Huang, Peng Dai, Liefeng Bo, and Jiawei Zhang. 2019. Heterogeneous deep graph infomax. arXiv:1911.08538. Retrieved from https://arxiv.org/abs/1911.08538
[64] Joshua David Robinson, Ching-Yao Chuang, Suvrit Sra, and Stefanie Jegelka. 2021. Contrastive learning with hard negative samples. In *ICLR*. Retrieved from https://openreview.net/pdf?id=CR1XOQ0UTh
[65] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.
[66] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *ESWC*, Vol. 10843, 593–607.
[67] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik G. Learned-Miller. 2015. Multi-view convolutional neural networks for 3D shape recognition. In *ICCV*, 945–953.
[68] Ke Sun, Zhouchen Lin, and Zhanxing Zhu. 2020. Multi-stage self-supervised learning for graph convolutional networks on graphs with few labeled nodes. In *AAAI*, 5892–5899.

[69] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code search based on context-aware code translation. In *ICSE*, 388–400.
[70] Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. 2020. PSCS: A path-based neural model for semantic code search. arXiv:2008.03042. Retrieved from https://arxiv.org/abs/2008.03042
[71] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted code completion system. In *KDD*, 2727–2735.
[72] Yu Tian, Xi Peng, Long Zhao, Shaoting Zhang, and Dimitris N. Metaxas. 2018. CR-GAN: Learning complete representations for multi-view generation. In *IJCAI*, 942–948.
[73] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. arXiv:1807.03748. Retrieved from https://arxiv.org/abs/1807.03748
[74] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. HELoC: Hierarchical contrastive learning of source code representation. In *ICPC*, 354–365.
[75] Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *AAAI*, 14015–14023.
[76] Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao. 2022. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer, Singapore.
[77] Lirong Wu, Haitao Lin, Zhangyang Gao, Cheng Tan, and Stan Z. Li. 2023. Self-supervised on graphs: Contrastive, generative, or predictive. *IEEE Trans. Knowl. Data Eng*. 35, 4 (2023), 4216–4235.
[78] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2024. Survey of code search based on deep learning. *ACM Trans. Softw. Eng. Methodol*. 33, 2 (2024), 54:1–54:42.
[79] Yaochen Xie, Zhao Xu, Jingtun Zhang, Zhengyang Wang, and Shuiwang Ji. 2023. Self-supervised learning of graph neural networks: A unified review. *IEEE Trans. Pattern Anal. Mach. Intell*. 45, 2 (2023), 2412–2429.
[80] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks? In *ICLR*. Retrieved from https://openreview.net/pdf?id=ryGs6iA5Km
[81] Xiaoqiang Yan, Shizhe Hu, Yiqiao Mao, Yangdong Ye, and Hui Yu. 2021. Deep multi-view learning methods: A review. *Neurocomputing* 448 (2021), 106–129.
[82] Mouxing Yang, Yunfan Li, Peng Hu, Jinfeng Bai, Jiancheng Lv, and Xi Peng. 2023. Robust multi-view clustering with incomplete information. *IEEE Trans. Pattern Anal. Mach. Intell*. 45, 1 (2023), 1055–1069.
[83] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2022. A survey on deep learning for software engineering. *ACM Comput. Surv*. 54, 10s (2022), 206:1–206:73.
[84] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. 2020. Graph contrastive learning with augmentations. In *NeurIPS*.
[85] Yuning You, Tianlong Chen, Zhangyang Wang, and Yang Shen. 2020. When does self-supervision help graph convolutional networks? In *ICML*, 10871–10880.
[86] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Bailin Xiao, Wei Dong, and Xiangke Liao. 2022. deGraphCS: Embedding variable-based flow graph for neural code search. *ACM Trans. Softw. Eng. Methodol*. 32, 2 (2022), 1–27.
[87] Chenyuan Zhang, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. EALink: An efficient and accurate pre-trained framework for issue-commit link recovery. In *ASE*, 217–229.
[88] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*, 783–794.
[89] Sheng Zhang, Hui Li, Yanlin Wang, Zhao Wei, Yong Xiu, Juhong Wang, and Rongong Ji. 2023. Code search debiasing: Improve search results beyond overall ranking performance. arXiv:2311.14901. Retrieved from https://arxiv.org/abs/2311.14901
[90] Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. 2020. OCoR: An overlapping-aware code retriever. In *ASE*, 883–894.
[91] Yanqiao Zhu, Yichen Xu, Feng Yu, Qiang Liu, Shu Wu, and Liang Wang. 2020. Deep graph contrastive representation learning. arXiv:2006.04131. Retrieved from https://arxiv.org/abs/2006.04131

Received 18 October 2022; revised 6 November 2024; accepted 8 May 2025