FROM PSEUDO-CODE TO SOURCE CODE: A SELF-SUPERVISED SEARCH APPROACH

Adithya Kulkarni*, **Mohna Chakraborty***, **Yonas Sium***
Iowa State University
Ames, Iowa, USA
{aditkulk, mohnac, yas}@iastate.edu

Sai Charishma Valluri
Iowa State University
Ames, Iowa, USA
svalluri@iastate.edu

Wei Le
Iowa State University
Ames, Iowa, USA
weile@iastate.edu

Qi Li
Iowa State University
Ames, Iowa, USA
Qli@iastate.edu

ABSTRACT

Identifying algorithm implementations in source code is crucial for code comprehension, reference retrieval, and program synthesis. This paper presents *PC2SC*, a novel framework for mapping pseudo-code to source code without manual annotations. We introduce p-language, a structured representation that encodes control flow, mathematical expressions, and natural language descriptions of algorithms. A static analyzer extracts these features, converting pseudo-code into p-code, then embedded into a shared vector space with source code using self-supervised learning for retrieval. Given pseudo-code as input, *PC2SC* returns a ranked list of matching code snippets. Evaluations on the Stony Brook Algorithm Repository and GitHub projects demonstrate that *PC2SC* outperforms state-of-the-art code search tools in both C and Java. It successfully retrieves correct implementations within the top 25, 10, and 1 ranked results for 98.5%, 93.8%, and 66.2% of queries, respectively. In GitHub projects, it identified 74 algorithm implementations out of 87 queries. *PC2SC* bridges the gap between algorithmic descriptions and executable implementations, offering a scalable, language-independent solution for algorithm retrieval and paving the way for future advancements in cross-language code search and automated synthesis.

1 INTRODUCTION

Understanding and retrieving algorithm implementations from source code is a fundamental problem in software engineering, with applications in code comprehension, debugging, program synthesis, and software verification. Establishing explicit mappings between algorithms and their implementations aids in understanding unfamiliar code, detecting semantic clones, and identifying bugs. Reference implementations from real-world codebases provide insights into data structures, optimizations, and corner cases, aspects often missing in pseudo-code. Incorrect algorithm implementations have been identified as a leading cause of failures in safety-critical systems such as self-driving cars Garcia et al. (2020). Beyond debugging, algorithm-code mappings are crucial for automated testing Carzaniga et al. (2014); Elyasov et al. (2015), program repair Sidiroglou-Douskos et al. (2015), API design Ding et al. (2015), and evaluating deep learning models for code analysis Zhao & Huang (2018); Sui et al. (2020); Wang & Su (2019); Wang (2019); Guo et al. (2021). Additionally, linking pseudo-code to source code facilitates program synthesis, where pseudo-code serves as an intermediate representation Kulal et al. (2019a).

Existing approaches to code search primarily rely on natural language queries, while clone detection identifies syntactic or semantic similarities in code. Deep Code Search (DCS) Gu et al. (2018) and OCoR Zhu et al. (2020) leverage deep learning on code-comment pairs to retrieve implementations, while tools such as SLACC Mathew et al. (2020) and GraphCodeBERT Guo et al.

*The first three authors contributed equally to this research.

(2021) detect semantic clones. However, these methods do not explicitly link algorithmic descriptions to their implementations. Recent work on pseudo-code retrievals, such as PseudoSeer Toksoz et al. (2024), demonstrates the growing importance of structured pseudo-code representations. Additionally, cross-language clone detection using large language models (LLMs) Moumoula et al. (2024) has shown that embedding models can effectively detect semantically equivalent code snippets across languages. A comprehensive discussion of related works is presented in Appendix A.1.

This paper presents *PC2SC*, a framework for searching and mapping pseudo-code to source code, enabling language-independent retrieval across Java and C. Unlike natural language-based search, pseudo-code provides a structured representation of algorithms, making it a more reliable input for automated code retrieval. Our approach bridges the gap between abstract algorithm descriptions and real-world implementations by leveraging control flow, mathematical expressions, and natural language semantics.

PC2SC consists of three key components: algorithm analysis, representation learning, and vector-based code retrieval. We extract control flow, mathematical expressions, and natural language features from pseudo-code and source code, encoding them into a shared vector space via graph-based representation learning. Interprocedural Control Flow Graphs (ICFGs) capture structural relationships within code, while a scalable vector-based search algorithm identifies and clusters matching source code nodes to form precise code fragments. While implementations may introduce optimizations, we hypothesize that core control flow structures remain preserved, and mathematical expressions and function/variable names serve as distinguishing features.

We introduce p-language, a structured representation that encodes an algorithm's control flow and mathematical and natural language components to facilitate automatic algorithm analysis. We develop a learning-based method to convert pseudo-code into p-code, which is analyzed to extract ICFGs, mathematical operators, and textual descriptors. We enable fully automated pseudo-code analysis, bridging the gap between high-level algorithm descriptions and executable source code. We encode both pseudo-code and source code into a multi-modal feature space consisting of control flow, natural language semantics, and mathematical expressions. Control flow is captured via a Graph Autoencoder (GAE) with a Graph Convolutional Network (GCN) encoder, trained via self-supervised learning on ICFGs. Natural language semantics are extracted using CodeBERT Feng et al. (2020), which aligns code tokens with natural language descriptions. Mathematical expressions are separately categorized and concatenated with the existing embeddings to differentiate algorithm implementations.

Code retrieval is performed by considering both semantic similarities and structural connectivity within the ICFG. The retrieval process consists of (1) candidate selection, where node-level embeddings from pseudo-code and source code are compared, and (2) fragment construction, where matched nodes are grouped based on proximity within the ICFG. Finally, retrieved code fragments are ranked based on structural coherence and query alignment. To evaluate the effectiveness of our proposed tool *PC2SC*, we collected 103 algorithm pseudo-code descriptions from Introduction to Algorithms Cormen et al. (2009) and The Algorithm Design Manual Skiena (1998). We curated two large-scale code repositories: (1) the Stony Brook Algorithm Repository, containing 67 Java and 27 C projects, and (2) a GitHub-based code database comprising 10 Java and 10 C projects. Across three real-world search scenarios, we executed 65 queries on the Stony Brook database and 87 queries on the GitHub database. *PC2SC* successfully retrieved correct implementations within the top 25, 10, and 1 ranked results for 98.5%, 93.84%, and 66.15% of queries, respectively, across major open-source projects such as Guava, AlgoDS, kdtree, and Concorde. Additionally, *PC2SC* discovered 74 previously unknown algorithm implementations in GitHub projects, demonstrating its practical utility. *PC2SC* outperformed state-of-the-art tools such as DCS Gu et al. (2018) and OCoR Zhu et al. (2020), highlighting its effectiveness in cross-language algorithm retrieval.

The key contributions of this paper include: (1) p-language: A structured representation for automated algorithm analysis, (2) Representation learning framework: Integrating control flow, mathematical expressions, and natural language semantics for algorithm-code matching, (3) Vector-based search algorithm: Efficient retrieval and ranking using ICFG-based structural matching, (4) *PC2SC*: A multi-language tool for algorithm retrieval across Java and C, and (5) Dataset contribution: A publicly available dataset of pseudo-code, p-code, and corresponding algorithm implementations.

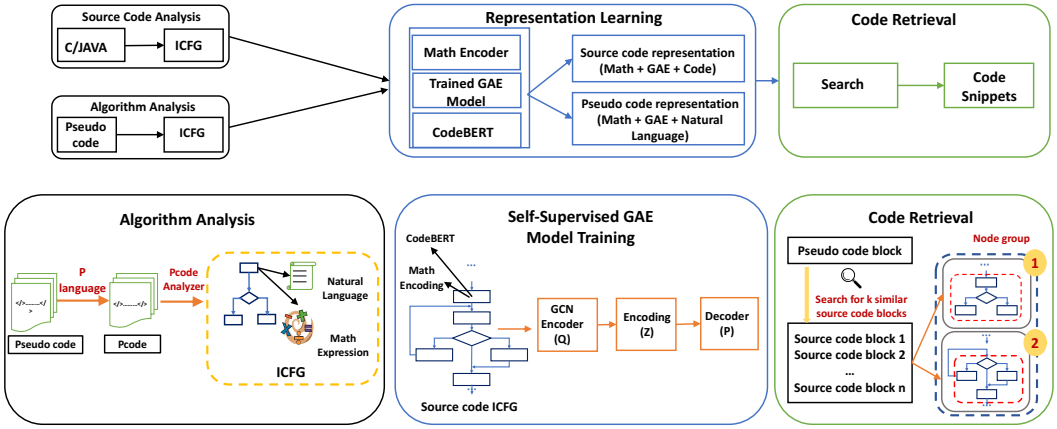

Figure 1: Workflow of *PC2SC*

2 OVERVIEW

This section presents the overall workflow of *PC2SC*, detailing its core components and how it retrieves algorithm implementations from source code. A detailed example illustrating the mapping between pseudo-code and source code is provided in Appendix A.5.

Figure 1 illustrates the workflow of *PC2SC*. Given an algorithm's pseudo-code as input and a codebase as the search database, *PC2SC* retrieves fine-grained code snippets corresponding to the algorithm's implementation. The retrieved snippets may consist of a function, selected code blocks within a function, or multiple interconnected functions across files.

PC2SC consists of three core components: (1) Algorithm Analysis: Converts pseudo-code into p-code, a structured representation in p-language, which explicitly encodes control flow, mathematical expressions, and natural language descriptors. A static parser extracts ICFGs, mathematical operators, and textual descriptions, (2) Representation Learning: Applies self-supervised graph learning using a GAE with a GCN to generate vector representations for p-code and source code. Each node embedding captures mathematical, structural, and natural language features for similarity detection and (3) Search and Retrieval: Computes semantic similarities between pseudo-code and source code nodes, clustering matched code snippets based on ICFG connectivity. A ranking function scores fragments based on structural coherence and query coverage, ensuring highly relevant results.

By integrating graph-based representation learning with semantic search, *PC2SC* achieves accurate and language-independent retrieval of algorithm implementations across Java and C.

3 AUTOMATIC ALGORITHM ANALYSIS

Algorithms are commonly described using pseudo-code, which conveys computational logic in a structured yet human-readable format. This section presents our automated approach for extracting three key features—*control flow*, *mathematical expressions*, and *natural language descriptions*—to enable precise algorithm-to-code mapping.

3.1 THREE KEY FEATURES IN ALGORITHM DESCRIPTIONS

Algorithms inherently contain three fundamental types of information: (1) Control flow: The sequence and structure of computational steps, (2) Mathematical expressions: Computation-specific formulas and operators, and (3) Natural language descriptions: Informal explanations of computational intent. The control flow of an algorithm typically falls into one of three categories Cormen et al. (2009); Xu (2017): Sequential execution, Conditional branching (`if-else`), and Iteration (`while`, `for`, `repeat-until`). Pseudo-code frequently combines control flow with mathematical operations and natural language explanations, making it a semi-structured representation.

To enable automated analysis, we introduce p-language, a structured representation that explicitly annotates control flow, mathematical expressions, and natural language elements within pseudo-code. A static analyzer extracts these features and converts pseudo-code into p-code, which serves

as an intermediate representation for further analysis. The full p-language grammar and parsing rules are detailed in Appendix A.6.

3.2 P-LANGUAGE AND ITS STATIC ANALYZER

To facilitate automated pseudo-code analysis, we designed p-language, which explicitly introduces keywords to mark control flow structures. The p-language grammar defines key components, including, Mathematical expressions, denoted using '$$' markers, Natural language descriptions enclosed within '@@' markers, and Control structures, explicitly represented using standard keywords such as 'if', 'while', 'for', and 'return'. A static analyzer parses p-code and constructs ICFGs, where nodes correspond to control flow statements, mathematical operations, and natural language descriptions. Unlike traditional source code ICFGs, p-code ICFGs do not contain implementation-specific syntax but instead retain algorithmic structure. Extended details on ICFG construction and node representation can be found in Appendix A.7.

3.3 AUTOMATICALLY CONVERTING PSEUDO-CODE TO P-CODE

We developed a learning-based approach to convert pseudo-code into p-code by classifying statements into mathematical expressions or natural language descriptions. To train this classifier we use code comments as labeled natural language descriptions and source code statements as labeled mathematical expressions, and we employ a semi-supervised label propagation algorithm Zhou et al. (2003), which refines predictions by jointly considering labeled and unlabeled data distributions. Using this method, we automatically annotate pseudo-code and generate its p-code representation without requiring manual intervention. The classifier training process and label propagation method are detailed in Appendix A.8.

4 REPRESENTATION OF CODE AND ALGORITHMS

We construct a unified representation for both *p-code* and source code to enable effective algorithm-to-code search. We convert both representations into ICFGs and extract each node's mathematical and natural language features. We then apply a GNN framework, namely the GAE Kipf & Welling (2016), to learn a shared embedding space, ensuring meaningful comparisons between *p-code* and source code representations.

4.1 COMPARING P-CODE AND SOURCE CODE

P-code facilitates the automated extraction of three key features essential for algorithm retrieval. First, it captures control flow, which defines the logical execution order, including loops and branching structures. Second, it extracts mathematical expressions, representing the operators and computations that characterize the algorithm's numerical properties. Third, it incorporates natural language descriptions, providing human-readable explanations of the computational steps. Since different algorithms may exhibit similar ICFG structures, additional node features are integrated to enhance differentiation. These include natural language semantics, encoded using CodeBERT Feng et al. (2020), and explicit representations of mathematical operations to preserve numerical computations.

CodeBERT encodes both source code and natural language descriptions into a shared vector space, enabling cross-modal similarity detection. We apply different encoding strategies for *p-code* and source code. Further details on ICFG construction and mathematical feature encoding are provided in Appendix A.9.

4.2 TRAINING A MODEL FOR ICFGS VIA SELF-SUPERVISED LEARNING

We utilize a Graph Autoencoder (GAE) framework to encode the control flow structure of both p-code and source code. This self-supervised learning approach eliminates the need for human annotations and follows an encoder-decoder architecture. The framework consists of a Graph Encoder, which employs a Graph Convolutional Network (GCN) to generate embeddings from input graphs, and a Graph Decoder, which reconstructs the graph structure using the learned embeddings.

The training process leverages link prediction, where the model learns to predict whether an edge should exist between nodes. To ensure balanced learning, we randomly sample non-existing edges as negative samples. Further details on the GCN encoder, training objective, and loss function are provided in Appendix A.10.

5 CODE RETRIEVAL

PC2SC retrieves algorithm implementations by searching for semantically similar code fragments based on pseudo-code queries. This process consists of two main steps: (1) Matching nodes between p-code and source code, and (2) Grouping nodes into meaningful code fragments.

5.1 MATCHING NODES

Given a pseudo-code query, we first convert it into p-code and generate its corresponding ICFG (Section 3.2). The source code database is also represented as ICFGs (Section 4). To identify relevant source code, we compute cosine similarity (S_C) between each p-code node (e_i^P) and source code node (e_j^S):

$$S_C(e_i^P, e_j^S) = \frac{e_i^P \cdot e_j^S}{||e_i^P||\ ||e_j^S||}. \tag{1}$$

Cosine similarity measures the semantic closeness of vector embeddings in the shared representation space. For efficient large-scale retrieval, we use FAISS [1], an approximate nearest neighbor search tool, to index the source code embeddings. The top k most similar source code nodes are selected as candidate matches. Further details on FAISS indexing and vector-based retrieval are provided in Appendix A.11.

5.2 NODE GROUPING

Once relevant source code nodes are matched, we group them into coherent code fragments. We employ agglomerative hierarchical clustering Han et al. (2011) to merge nodes based on their proximity in the source code ICFG. The distance metric for merging clusters is defined as:

$$distance = \max\{sp(x, y) : x \in C_1, y \in C_2\}, \tag{2}$$

where $sp(x, y)$ is the shortest path distance between nodes x and y. If no path exists, the distance is set to ∞, and clusters remain separate. We rank candidate clusters using a scoring function that prioritizes high query coverage and structural coherence:

$$\gamma = QC + \frac{1}{distance'}, \tag{3}$$

where QC is the number of p-code nodes covered, and $distance'$ is the largest shortest path within the cluster. This ensures that closely connected, semantically relevant code fragments are ranked higher. Additional details on clustering, distance metrics, and ranking functions are included in Appendix A.12.

6 EVALUATION

To assess the effectiveness of *PC2SC*, we designed a comprehensive evaluation addressing four key research questions: **RQ1:** Can *PC2SC* effectively retrieve algorithm implementations from source code? **RQ2:** How does *PC2SC* compare to state-of-the-art code search models? **RQ3:** Does *PC2SC* generalize across different programming languages? **RQ4:** What is the contribution of individual features (control flow, mathematical expressions, and natural language) to retrieval accuracy?

6.1 EXPERIMENTAL SETUP AND IMPLEMENTATION

To evaluate *PC2SC*, we curated a dataset consisting of pseudo-code descriptions of algorithms and their real-world implementations in Java and C. The dataset includes 103 algorithm descriptions from *Introduction to Algorithms* Cormen et al. (2009) and *The Algorithm Design Manual* Skiena (1998). The source code implementations were collected from two major repositories: (1) The Stony Brook Algorithm Repository, which provides real-world algorithm implementations across 67 C projects and 27 Java projects, and (2) A set of 20 additional GitHub projects comprising 10 Java and 10 C repositories. These GitHub repositories were selected based on several criteria, including originality (not forks), project size exceeding 5MB, active maintenance, at least 1000 GitHub stars, and compatibility with Atlas for ICFG generation. The full dataset selection criteria and project filtering process are detailed in Appendix A.19.

We first converted the pseudo-code descriptions into p-code to process the dataset using a trained classifier. The classifier was trained using a labeled dataset consisting of source code comments (as

[1] https://github.com/facebookresearch/faiss

natural language labels) and executable code statements (as mathematical expressions). We then applied the Label Propagation Algorithm from the Scikit-learn library to automatically annotate the pseudo-code. The labeled pseudo-code was then parsed into ICFGs using *PC2SC*'s static analyzer, which was implemented in C and adheres to the structured grammar of *p-language* (Section 3.2). The same process was applied to Java and C source code implementations to extract their corresponding ICFGs.

To train the Graph Autoencoder (GAE) for representation learning, we used PyTorch Geometric with a two-layer Graph Convolutional Network (GCN) encoder. Each layer generated 512-dimensional embeddings, and training was performed using the Adam optimizer ($\eta = 0.01$) with a batch size of 2048. The dataset was split into 80% training and 20% validation, and early stopping was applied when validation performance plateaued. We trained three models: one exclusively on C code, one on Java code, and a mixed-language model trained on both.

We integrated FAISS, an approximate nearest neighbor search framework, for efficient search and retrieval to index and retrieve relevant source code fragments. We computed cosine similarity (Section 5.1) between the p-code embeddings and the indexed source code embeddings, selecting the top 100 candidates per query. Further details on model training, hardware configurations, and hyperparameter settings are provided in Appendix A.15.

6.1.1 Experimental Design

To systematically evaluate *PC2SC*'s retrieval accuracy and generalization capabilities, we designed three experimental settings:

1. Single Project Search (Exp 1): This experiment measured *PC2SC*'s ability to retrieve an algorithm implementation within a single project. This scenario reflects a typical use case where a developer investigates a specific library or legacy codebase to locate an algorithm implementation. The Stony Brook dataset was used for the experiment, as it provides a ground truth for which algorithms are implemented in each project, allowing for precise evaluation.

2. Multi-Project Search (Exp 2): This experiment assessed *PC2SC*'s effectiveness in retrieving an algorithm across multiple projects. This scenario simulates a practical use case where a developer searches for an algorithm implementation across multiple repositories, such as in a company's internal codebase. The experiment was conducted on the Stony Brook dataset, testing how effectively *PC2SC* can locate algorithm implementations when multiple projects are available.

3. GitHub Search (Exp 3): To evaluate *PC2SC*'s generalization ability in real-world open-source projects, we tested its performance on GitHub repositories. The experiment simulated a situation where a developer searches for an algorithm implementation in large, publicly available repositories. We used the GitHub dataset and queried the same 29 algorithms from the Stony Brook dataset to measure retrieval effectiveness in a more diverse and unstructured environment.

These three experiments were designed to evaluate *PC2SC*'s retrieval accuracy, scalability, and ability to generalize across different languages and codebases. The experimental workflow is illustrated in Appendix A.16.

6.1.2 Baseline Comparisons

To benchmark *PC2SC*'s performance, we compared it against three baselines: (1) Naïve Name-Based Search: This approach searches for function names that match algorithm names across source code files. If a match is found, the corresponding method is retrieved. This represents a simple keyword-based search strategy but does not account for semantic or structural differences, (2) Deep Code Search (DCS) Gu et al. (2018): A deep learning-based model retrieves code snippets using a neural embedding model trained on code-comment pairs, and (3) OCoR Zhu et al. (2020): A deep learning model that matches question-code similarity based on StackOverflow question-answer pairs. We retrained both DCS and OCoR on our dataset for a fair comparison using their official implementations. DCS was trained using extracted method names, API sequences, and tokens from method bodies, while OCoR was trained using comment-code pairs converted from pseudo-code descriptions. Details on baseline implementation, training configuration, and performance metrics are provided in Appendix A.20.

6.1.3 SEARCH RESULTS VERIFICATION AND EVALUATION METRICS

The output of *PC2SC* consists of ranked code snippets, while the baselines return ranked method-level results. To validate correctness, two independent annotators manually reviewed the top 100 retrieved results per query. A code snippet was confirmed as an implementation if (1) its control flow (e.g., loops and conditions) aligned with the pseudo-code, and (2) its variables and mathematical operators corresponded to those in the pseudo-code. Since real-world implementations often introduce optimizations or structural variations, annotators also examined the broader logic to determine functional equivalence. Disagreements were discussed, and ambiguous cases were marked as "unknown".

To quantify the retrieval effectiveness of *PC2SC*, we employed two key evaluation metrics: First-Rank (F-rank) and Mean Reciprocal Rank (MRR). The F-rank metric measures the position of the first correctly retrieved result, where lower values indicate better performance. If no match is found within the top 100 results, it is denoted as "$>$100". In contrast, MRR is computed as the mean of the inverse F-rank across all queries, providing a measure of overall ranking effectiveness. Higher MRR values indicate stronger performance in prioritizing relevant results at the top of the ranked list. In Experiment 1 (Single Project Search), algorithm implementations were retrieved from multiple projects, and we reported the mean F-rank across relevant projects. Full evaluation metrics and tabulated results for all settings are presented in the Appendix (Tables 2, 3, 4, 5, and 6).

6.1.4 ABLATION STUDY: FEATURE CONTRIBUTION ANALYSIS

To evaluate the impact of individual components in *PC2SC*, we conducted an ablation study by comparing three model configurations. The CB (CodeBERT only) variant relies solely on source code and natural language embeddings for retrieval, omitting structural information. The CB+G (CodeBERT + GAE) configuration extends this by incorporating control flow graph (ICFG) encoding, capturing the execution structure of programs. Finally, the Full Model (*PC2SC*) integrates all three key features, including control flow, mathematical expressions, and natural language representations, providing a comprehensive embedding for improved retrieval accuracy.

The results demonstrate that control flow encoding significantly improves retrieval accuracy for C code, where textual representations alone are insufficient. Mathematical encoding enhances retrieval for numerical algorithms, which rely on distinct computational patterns. Overall, the full model consistently outperformed CB and CB+G, highlighting the importance of integrating all three feature types for robust algorithm retrieval. The detailed ablation study results are provided in the Appendix (Tables 2, 3, 4, 5, and 6).

6.1.5 SYSTEMS THAT CONDUCTED EXPERIMENTS

To convert pseudo code to *p-code* and to generate the ICFGs for source code and *p-code*, we used a MacBook Pro with 4GB AMD GPU, 16 GB RAM, and 6 CPU Cores. We used Google Colab with V100 GPU, 53 GB RAM, and 8 CPU Cores for GAE training and code retrieval.

Table 1: Summary of MRR Scores for Different Experimental Settings

Experimental Setting	Language	*PC2SC*	DCS	OCoR	CB+G	CB
Single Project Search (Exp 1)	Java	**0.819**	0.307	0.346	0.650	0.517
	C	**0.522**	0.357	0.281	0.472	0.34
	Mixed	**0.498**	0.304	0.289	0.349	0.243
Multi-Project Search (Exp 2)	Java	**0.606**	0.232	0.263	0.432	0.256
	C	**0.316**	0.192	0.149	0.265	0.184
	Mixed	**0.313**	0.146	0.203	0.189	0.122
GitHub Search (Exp 3)	Java	**0.580**	0.113	0.132	0.427	0.217
	C	**0.321**	0.113	0.133	0.249	0.08
	Mixed	**0.345**	0.081	0.098	0.229	0.087

6.2 RESULTS

To evaluate *PC2SC*'s effectiveness, we conducted a series of experiments comparing it with state-of-the-art code search models across different datasets. The results demonstrate that *PC2SC* consis-

tently outperforms existing methods by integrating control flow, mathematical features, and natural language representations. We present key findings and discuss the broader implications of our approach in the main paper. Full numerical results, including ranking performance, are provided and discussed in Appendix A.17.

6.2.1 PERFORMANCE ON STONY BROOK CODEBASE

The first set of experiments focused on retrieving known algorithm implementations from the Stony Brook Algorithm Repository, a structured dataset with documented implementations of classical algorithms. This setting allows us to measure how well *PC2SC* retrieves algorithms whose presence in the codebase is already known, providing a controlled evaluation of retrieval effectiveness.

Our results indicate that *PC2SC* successfully retrieved correct implementations for 98.5% of queries within the top 25 ranked results across Java, C, and mixed-language settings. Java implementations achieved the highest retrieval accuracy, likely due to CodeBERT's pretraining on Java, enhancing its ability to model source code features effectively. In contrast, C implementations exhibited slightly lower performance, suggesting that real-world C implementations often introduce structural optimizations that deviate from textbook-style pseudo-code.

In the single project search experiment, where retrieval is limited to a single repository, *PC2SC* correctly identified the target algorithm in the top-ranked position for 86.2% of Java queries and 50% of C queries. The slightly lower performance in C projects suggests that algorithm implementations in C often span multiple functions or involve low-level optimizations, making direct matching more challenging. However, in the multi-project search experiment, where retrieval was extended across multiple repositories, *PC2SC* maintained high retrieval accuracy, retrieving correct implementations for 96.9% of Java queries and 66.7% of C queries within the top 25 results. This suggests that control flow and mathematical similarity remain strong indicators of algorithm presence even when code is scattered across multiple projects.

A key observation is that *PC2SC* consistently outperforms deep learning-based baselines (DCS and OCoR) across all settings. Compared to DCS, which relies on neural models trained on code-comment pairs, and OCoR, which uses question-code similarity from StackOverflow, *PC2SC* 's incorporation of graph-based control flow learning and mathematical encoding provides a more effective way to capture algorithmic semantics. Quantitatively, *PC2SC* outperformed DCS by 167% and OCoR by 137% in Mean Reciprocal Rank (MRR) for Java code and achieved 46% and 86% improvements in C, respectively. The mixed-language setting, where models were trained on a combination of Java and C projects, further confirmed *PC2SC* 's robustness, with performance gains of 64% over DCS and 72% over OCoR.

6.2.2 PERFORMANCE ON GITHUB CODEBASE

To assess *PC2SC*'s applicability in real-world open-source projects, we conducted additional experiments on a set of popular GitHub repositories. Unlike the structured Stony Brook dataset, GitHub repositories contain diverse implementations, making retrieval more challenging due to differences in coding style, documentation quality, and naming conventions.

Despite these challenges, *PC2SC* demonstrated strong generalization capabilities. It retrieved correct implementations for 82.7% of Java queries and 82.7% of C queries within the top 25 ranked results. These results suggest that *PC2SC* remains effective even when dealing with unstructured, real-world codebases where function names and documentation are less standardized.

One of the most striking findings is the performance gap between *PC2SC* and deep learning-based baselines. On the GitHub dataset, *PC2SC* outperformed DCS by 413% and OCoR by 339% (Java MRR) and achieved 184% and 141% improvements in C, respectively. This highlights the limitations of text-based retrieval models, which struggle when function names and documentation vary significantly from textbook pseudo-code. In contrast, *PC2SC*'s control flow graph representation and mathematical encoding enable it to retrieve semantically similar implementations despite naming differences and optimizations.

However, we also observed cases where retrieval was more difficult. Some algorithms, particularly those that rely heavily on external libraries or inline optimizations, proved challenging for *PC2SC*. In these instances, pseudo-code descriptions failed to capture the full complexity of real-world im-

plementations, leading to mismatches. Nonetheless, even in these cases, *PC2SC*'s top-ranked results often contained partially correct implementations, which could still assist developers in locating relevant code fragments.

6.2.3 Feature Contribution Analysis

To better understand the contributions of *PC2SC*'s core components, we conducted an ablation study that isolates the effects of control flow, mathematical encoding, and natural language representations. We compared three different configurations of our model to evaluate the impact of each feature. The CB (CodeBERT only) configuration relies solely on source code and natural language features, excluding structural information. The CB+G (CodeBERT + GAE) configuration enhances retrieval by incorporating control flow graph encoding, capturing the program's execution structure. The Full Model (*PC2SC*) integrates all three features: control flow, mathematical expressions, and natural language semantics, ensuring the most comprehensive representation for retrieval.

The results confirm that each feature plays a crucial role in improving retrieval accuracy. Control flow encoding significantly improves retrieval for C code, where textual representations alone are insufficient. Mathematical encoding enhances retrieval accuracy for numerical algorithms, where distinct computational operations must be matched precisely. Across all settings, the full *PC2SC* model consistently outperformed both CB and CB+G, indicating that combining all three feature types is necessary for robust algorithm retrieval.

6.3 Qualitative Examples of Retrieved Code

To illustrate how *PC2SC* retrieves algorithm implementations, Figures 5, 6, and 7 in the Appendix provide examples of correctly retrieved code snippets from both structured repositories and unstructured GitHub datasets. These examples highlight key aspects of *PC2SC*'s retrieval mechanism.

First, pseudo-code and actual implementations often exhibit structural variations due to optimizations and coding conventions. However, despite these differences, control flow alignment remains a dominant factor in retrieval, ensuring that algorithmic logic is preserved even when syntax varies. Second, mathematical expressions and function mappings significantly enhance retrieval accuracy, particularly in cases where source code variables differ in naming conventions from those in pseudo-code. By encoding math operations explicitly, *PC2SC* ensures semantic consistency across different representations of the same computation. Third, semantic alignment between pseudo-code descriptions and source code comments plays a crucial role in refining search results. The joint encoding of natural language descriptions and control flow structures allows *PC2SC* to bridge the gap between high-level algorithmic descriptions and implementation details.

A more detailed discussion of these retrieval examples, including failure cases and challenges encountered in mapping pseudo-code to source code, is provided in Appendix A.18.

7 Conclusions

We presented *PC2SC*, a framework for mapping pseudo-code to source code and retrieving algorithm implementations from real-world open-source projects. *PC2SC* integrates automatic algorithm analysis, representation learning, and vector-based retrieval to encode and match control flow, mathematical expressions, and natural language semantics. Our evaluation shows that *PC2SC* effectively retrieves algorithm implementations in both C and Java, identifying 74 implementations from 87 queries in GitHub and 64 out of 65 in the Stony Brook dataset, with over 90% ranked in the top 10. These results highlight its accuracy, scalability, and robustness in bridging the gap between algorithmic descriptions and real-world code. Future work includes expanding *PC2SC* to support additional languages (e.g., Python, C++, Rust), integrating execution traces for context-aware retrieval, and exploring pseudo-code generation to create a bidirectional mapping between algorithm descriptions and implementations.

8 Acknowledgement

The work is supported by the US National Science Foundation under grant 2313054.

REFERENCES

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2018.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1186–1197. IEEE, 2021.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 964–974, 2019.

Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pp. 931–942, 2014. ISSN 02705257. doi: 10.1145/2568225.2568287. URL http://dl.acm.org/citation.cfm?doid=2568225.2568287.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

Yufei Ding, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. TOP : A Framework for Enabling Algorithmic Optimizations for Distance-Related Problems. *Proceedings of the VLDB Endowment*, 8(10):1046–1057, 2015. ISSN 21508097. URL http://dl.acm.org/citation.cfm?id=2794367.2794374.

Alexander Elyasov, W. Prasetya, Jurriaan Hage, Urko Rueda, Tanja E J Vos, and Nelly Condori-Fernández. AB=A: execution equivalence as a new type of testing oracle. *Sac*, pp. 1559–1566, 2015. doi: 10.1145/2695664.2695877. URL http://dl.acm.org/citation.cfm?doid=2695664.2695877.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pp. 385–396, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380397. URL https://doi.org/10.1145/3377811.3380397.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944. IEEE, 2018.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. Graphcodebert: Pre-training code representations with data flow. *ArXiv*, abs/2009.08366, 2021.

Rajarshi Haldar, Lingfei Wu, JinJun Xiong, and Julia Hockenmaier. A multi-perspective architecture for semantic code search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8563–8568, 2020.

Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 5954–5971, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.482. URL https://aclanthology.org/2021.emnlp-main.482/.

Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pp. 946–957, 2018.

Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019a. URL https://proceedings.neurips.cc/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019b.

Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 249–260. IEEE, 2017.

Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X Liu, Chunming Wu, and Shouling Ji. Deep graph matching and searching for semantic code retrieval. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 15(5):1–21, 2021.

F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 260–270, Nov 2015. doi: 10.1109/ASE.2015.42.

George Mathew, Chris Parnin, and Kathryn T Stolee. Slacc: Simion-based language agnostic code clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pp. 210–221, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380407. URL https://doi.org/10.1145/3377811.3380407.

Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Trans. Softw. Eng. Methodol.*, 22(4):37:1–37:30, October 2013. ISSN 1049-331X. doi: 10.1145/2522920.2522930. URL http://doi.acm.org/10.1145/2522920.2522930.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Micheline Bénédicte Moumoula, Abdoul Kader Kabore, Jacques Klein, and Tegawendé Bissyande. Large language models for cross-language code clone detection. *arXiv preprint arXiv:2408.04430*, 2024.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pp. 574–584. IEEE Press, 2015. ISBN 9781509000241. doi: 10.1109/ASE.2015.36. URL https://doi.org/10.1109/ASE.2015.36.

Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. *Pldi*, pp. 43–54, 2015. ISSN 03621340. doi: 10.1145/2737924.2737988. URL http://dl.acm.org/citation.cfm?id=2737924.2737988.

Steven S Skiena. *The algorithm design manual*, volume 2. Springer, 1998.

Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428301. URL https://doi.org/10.1145/3428301.

Levent Toksoz, Mukund Srinath, Gang Tan, and C Lee Giles. Pseudoseer: a search engine for pseudocode. *arXiv preprint arXiv:2411.12649*, 2024.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. Multi-modal attention network learning for semantic source code retrieval. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 13–25. IEEE Computer Society, 2019.

Ke Wang. Learning scalable and precise representation of program semantics. *CoRR*, abs/1905.05251, 2019. URL http://arxiv.org/abs/1905.05251.

Ke Wang and Zhendong Su. Learning blended, precise semantic program embeddings. *CoRR*, abs/1907.02136, 2019. URL http://arxiv.org/abs/1907.02136.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. Codemvp: Learning to represent source code from multiple views with contrastive pre-training, 2022. URL https://arxiv.org/abs/2205.02029.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 87–98. IEEE, 2016.

Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. Survey of code search based on deep learning. *ACM Trans. Softw. Eng. Methodol.*, 33(2), December 2023. ISSN 1049-331X. doi: 10.1145/3628161. URL https://doi.org/10.1145/3628161.

Zhiliang Xu. Pseudo Code Tutorial. https://www3.nd.edu/~zxu2/acms40390F11/Pseudo_Code_Summary.pdf, 2017.

Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Classification model for code clones based on machine learning. *Empirical Software Engineering*, 20(4):1095–1125, 2015.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale, 2024a. URL https://arxiv.org/abs/2402.01935.

Shaojie Zhang, Yiwei Ding, Enrui Hu, Yue Yu, and Yu Zhang. Enhancing code representation learning for code search with abstract code semantics. In *2024 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2024b. doi: 10.1109/IJCNN60899.2024.10650119.

Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pp. 141–151, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236068. URL https://doi.org/10.1145/3236024.3236068.

Dengyong Zhou, Olivier Bousquet, Thomas Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. *Advances in neural information processing systems*, 16, 2003.

Qihao Zhu, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. Ocor: An overlapping-aware code retriever. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 883–894. IEEE, 2020.

A APPENDIX

A.1 RELATED WORK

With recent advancements in deep learning, extensive research has been conducted in the area of code search, commonly called neural search. We categorize the related work into three sections: (1) code search using natural language queries, (2) code search using code snippets, and (3) code representation learning.

A.2 CODE SEARCH USING NATURAL LANGUAGE QUERIES

Natural language-based code search enables users to retrieve code snippets that match a given query written in natural language. Early approaches designed ranking functions based on handcrafted features Lv et al. (2015); Mcmillan et al. (2013). More recent research has formulated the problem as a cross-modality search task, where the goal is to map both the natural language query and source code into a shared semantic space Haldar et al. (2020); Ling et al. (2021); Gu et al. (2018); Zhu et al. (2020); Bui et al. (2021). One of the first neural approaches, Deep Code Search (DCS) Gu et al. (2018), employed LSTMs and Multi-Layer Perceptrons to encode both modalities, treating code snippets and natural language descriptions as sequential data. Later works explored improved encoding models, including Transformer-based pre-trained models Cambronero et al. (2019); Husain et al. (2019); Feng et al. (2020).

Recent advancements further incorporate structured code representations to enhance retrieval accuracy. For instance, some works have leveraged Abstract Syntax Trees (ASTs) and AST-based graphs Haldar et al. (2020); Ling et al. (2021); Guo et al. (2021), while others have used Control Flow Graphs (CFGs) Wan et al. (2019). A recent survey on deep learning-based code search Xie et al. (2023) provides a comprehensive overview of the field, categorizing models into three main components: query semantics modeling, code semantics modeling, and similarity computation. The study highlights how pre-trained models such as CodeBERT and GraphCodeBERT have improved retrieval performance by leveraging large-scale pretraining on code datasets.

A.3 CODE SEARCH USING CODE SNIPPETS

Code search using code snippets as queries focuses on retrieving semantically similar code fragments or detecting code clones. Traditional approaches rely on lexical and syntactic similarity metrics White et al. (2016); Yang et al. (2015); Li et al. (2017). FaCoy Kim et al. (2018) enhances retrieval by using query alternation, while DeepSim Zhao & Huang (2018) extracts data flow and control flow features to detect code similarities. SLACC Mathew et al. (2020) employs dynamic analysis to find semantically equivalent implementations across different languages, such as Python and Java.

Our work focuses on pseudo-code-based code search, which differs from natural language and code snippet-based searches. Pseudo-code provides a structured representation of algorithms, describing their control flow and computations while remaining language-independent. Unlike raw code snippets, pseudo-code abstracts away implementation details, making it a suitable input for cross-language code retrieval. Prior works have explored pseudo-code generation from source code Oda et al. (2015) and vice versa Kulal et al. (2019b), but to the best of our knowledge, ours is the first approach to analyze and search for algorithm implementations using pseudo-code automatically.

A.4 CODE REPRESENTATION LEARNING

Advances in code representation learning have played a crucial role in improving code search techniques. Inspired by word embedding techniques such as Word2Vec Mikolov et al. (2013), early works like Code2Vec Alon et al. (2019) and Code2Seq Alon et al. (2018) encode code sequences based on AST traversal paths. More recently, Transformer-based pre-trained models have demonstrated superior performance. CodeBERT Feng et al. (2020) and GraphCodeBERT Guo et al. (2021) apply masked language modeling and contrastive learning to learn contextual code representations.

A notable recent contribution, CodeSage Zhang et al. (2024a), introduced a two-stage pretraining framework for code representation learning, significantly improving downstream tasks like code summarization and retrieval. Similarly, CODE-MVP Wang et al. (2022) employs contrastive learning on multiple code representations, including ASTs, CFGs, and raw code, to improve semantic

similarity detection. Another work, ContraCode Jain et al. (2021), improves code embeddings by generating functionally equivalent variants of programs as adversarial examples, making models more robust to syntactic changes.

Recent research has also focused on enhancing representation learning for code search. A study presented at IJCNN 2024 Zhang et al. (2024b) proposes leveraging abstract code semantics to improve code retrieval models, demonstrating how self-distillation techniques can refine representations. Additionally, research in contrastive pretraining has improved identifying functionally similar code snippets across different programming languages Xie et al. (2023).

Our approach integrates these advancements by employing Graph Autoencoder (GAE)-based embeddings for p-code and source code, leveraging control flow, mathematical expressions, and natural language descriptions. Unlike prior methods focusing solely on natural language queries or raw code snippets, our method introduces a structured pseudo-code representation to bridge the gap between algorithmic descriptions and real-world implementations.

A.5 Example: Mapping Pseudo-Code to Source Code

To illustrate how *PC2SC* retrieves algorithm implementations, we consider a matrix multiplication example. The pseudo-code for matrix multiplication, shown in Figure 2a, serves as a query, while the retrieved implementation from an open-source Java project is displayed in Figure 2b. Despite differences in structure and optimizations, *PC2SC* successfully identifies the relevant implementation by analyzing control flow, mathematical operations, and variable dependencies.

A key retrieval aspect is control flow alignment between pseudo-code and source code. The loop structures at lines 4, 5, and 7 in the pseudo-code correspond to lines 1, 4, and 9 in the implementation, preserving the iterative computation logic. Additionally, the summation operation at line 8 in the pseudo-code maps to line 13 in the implementation, confirming functional equivalence. However, real-world implementations often incorporate optimizations, such as loop unrolling at lines 12–20, which reduces iteration overhead and improves performance. Further, cache efficiency is enhanced by storing matrix elements in a contiguous array (block at line 5), optimizing memory access patterns.

This example demonstrates *PC2SC*'s robustness in handling structural variations, enabling it to retrieve functionally equivalent implementations even when optimizations are applied. By leveraging graph-based control flow analysis, mathematical expression matching, and semantic feature extraction, *PC2SC* bridges the gap between abstract algorithm descriptions and real-world code implementations.

A.6 P-Language Grammar and Parsing Rules

The p-language is designed to encode algorithmic structures explicitly, ensuring consistent representation of control flow, mathematical expressions, and natural language descriptions. The formal grammar of p-language, depicted in Figure 3, defines how algorithms are parsed into structured representations. Each algorithm is expressed as a function containing a sequence of statements. Statements are categorized into simple statements, such as function calls, return statements, and expressions, and compound statements, such as loops and conditionals. The grammar also includes explicit markers for mathematical expressions enclosed in '$' symbols and natural language descriptions enclosed in '@' symbols, allowing structured extraction of algorithm semantics.

By formalizing pseudo-code in this manner, *PC2SC* ensures that key algorithmic components, including control flow constructs, mathematical operations, and textual descriptions, are preserved and comparable to source code during retrieval.

A.7 ICFG Construction and Node Representation

Interprocedural Control Flow Graphs (ICFGs) serve as the core representation for both p-code and source code, enabling structured comparison during retrieval; unlike traditional CFGs, which represent low-level program execution paths, p-code ICFGs abstract algorithmic control flow while omitting language-specific implementation details. Each node in the ICFG corresponds to a specific computational step, including branching conditions, loop structures, function calls, and mathematical expressions. Edges represent execution flow between these components, ensuring an accurate algorithm logic model.

```
MATRIX-MULTIPLY(A,B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

(a) Input: pseudo-code as a query

```
for (int iBlock = 0; iBlock < blockRows; ++iBlock){
  final int pStart = iBlock * BLOCK_SIZE;
  final int pEnd = FastMath.min(pStart + BLOCK_SIZE,rows);
  for (int jBlock = 0; jBlock < blockColumns; ++jBlock) {
    final double[] block = blocks[iBlock * blockColumns+...;
    final int qStart = jBlock * BLOCK_SIZE;
    final int qEnd = FastMath.min(qStart+BLOCK_SIZE,...);
    int k = 0;
    for (int p = pStart; p < pEnd; ++p) {
        double sum = 0;
        int q = qStart;
        while (q < qEnd - 3) {
            sum += block[k] * v[q] +
                       block[k + 1] * v[q + 1] +
                       block[k + 2] * v[q + 2] +
                       block[k + 3] * v[q + 3];
            k += 4;
            q += 4;}
        while (q < qEnd) sum += block[k++] * v[q++];
        out[p] += sum;}}
  return out;}
```

(b) Output: code fragment that implements the algorithm

Figure 2: Example: Matrix Multiplication

// function

func: NAME parameters suite

parameters: '(' para (, para)* ')'

para: NAME | expr

suite: '{' stmt+ '}'

// statement

stmt: simple_stmt | compound_stmt

compound_stmt: if_stmt | while_stmt | repeat_stmt | for_stmt

simple_stmt: expr | call_stmt | return_stmt

expr: natural_language | math

math: '$' MATH EXPRESSION '$'

natural_language: '@' DESCRIPTION '@'

// branch, loop and call

if_stmt: 'if' test suite ('elseif' test suite)* ['else' suite]

while_stmt: 'while' test suite

for_stmt: 'for'—'for each' expr ['to'|'downto' expr] suite

repeat_stmt: 'repeat' suite 'until' test

test: and_test | not_test | or_test

return_stmt: 'return' [expr | call_stmt]

call_stmt: NAME parameters

Figure 3: The grammar of the *p-language*

Key differences between p-code and source code ICFGs include algorithm-centric representation, where p-code ICFGs focus on algorithmic control flow rather than implementation details; mathematical expression nodes, where instead of raw operations, mathematical expressions form explicit graph nodes, facilitating direct comparison with source code computations, and interprocedural analysis, where function calls in p-code are resolved to their corresponding algorithm steps, creating a structured call hierarchy.

By encoding algorithms in ICFGs, *PC2SC* establishes a graph-based retrieval approach, enabling structural matching beyond simple textual or token-based similarity.

A.8 Label Propagation for Pseudo-Code Classification

We employ a semi-supervised label propagation approach to accurately distinguish between mathematical expressions and natural language descriptions in pseudo-code. The classification process begins with a small set of labeled examples, where code comments are treated as natural language descriptions and executable statements are labeled as mathematical expressions. The remaining unlabeled pseudo-code statements are then classified using a graph-based propagation algorithm.

A semantic similarity graph is constructed, where each node represents a pseudo-code statement, and edges capture closeness between statements based on syntax and word embeddings. Labels are iteratively propagated through this graph, ensuring that similar statements receive consistent classifications. This method allows us to automatically infer correct labels for unlabeled pseudo-code, preserving algorithmic semantics while reducing manual annotation effort.

A.9 ICFG Construction and Mathematical Feature Encoding

We incorporate mathematical feature encoding to differentiate algorithms effectively, categorizing operations into semantically meaningful groups. Operators are classified into seven categories, treating equivalent operations as functionally identical.

Addition and subtraction include '+' and '-'. Multiplication and division include '*' and '/'. Memory access operators include de-referencing using '[' and ']'. Modular arithmetic is represented by the modulus operator '%'. Bitwise operators include '¡¡' and '¿¿' shift operations. Logical operators include '&&', '!', and '||'. Relational operators include '¿=', '¡=', '¡', '¿', '!=', and '=='.

By normalizing mathematical representations, *PC2SC* ensures that semantically equivalent computations are recognized as similar, even if they differ in notation or implementation style. To differentiate algorithms, we categorize mathematical expressions based on operator types, grouping equivalent operations:

- Addition/Subtraction ($+$, $-$)
- Multiplication/Division ($\star$, $/$)
- Dereference Operators ([,])

- Modular Arithmetic (%)
- Bitwise Operators (<<, >>)
- Logical Operators (&&, !, ||)
- Relational Operators (>=, <=, <, >, !=, ==)

A.10 Graph Autoencoder (GAE) Training and Loss Function

To learn robust embeddings for ICFGs, we train a Graph Autoencoder (GAE) using a Graph Convolutional Network (GCN) encoder. The GCN propagates structural and feature information through graph layers, producing embeddings that capture both local and global control flow relationships. The encoder update rule is defined as:

$$\boldsymbol{Z}^{(\ell+1)} = ReLU\left(\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{X}^{(\ell)}\boldsymbol{W}^{(\ell)}\right), \tag{4}$$

where $\boldsymbol{A}$ is the adjacency matrix, $\boldsymbol{D}$ is the degree matrix, $\boldsymbol{X}^{(\ell)}$ is the node feature matrix, and $\boldsymbol{W}^{(\ell)}$ is the trainable weight matrix.

The decoder reconstructs the graph structure using a dot-product similarity function to predict node relationships:

$$P(\hat{\boldsymbol{A}}_{i,j}|z_i, z_j) = \sigma(z_i^T z_j), \tag{5}$$

where $\sigma(\cdot)$ is the sigmoid function, and z_i and z_j are node embeddings. The model is trained using a binary cross-entropy loss function, optimizing reconstruction accuracy.

$$\boldsymbol{L} = \sum_{i\in V, j\in V} (-A_{i,j}\log\hat{A}_{i,j} - (1-A_{i,j})\log(1-\hat{A}_{i,j})). \tag{6}$$

Training Setup and Hyperparameters. The Graph Autoencoder (GAE) model used for embedding generation is trained using an 80%-20% train-validation split. The model employs a two-layer Graph Convolutional Network (GCN) encoder, each producing 512-dimensional embeddings. Training is optimized using the Adam optimizer with a learning rate of $\eta = 0.01$ and a batch size of 2048. To prevent overfitting, early stopping is applied based on the validation loss.

A.11 Vector-Based Retrieval with FAISS

To efficiently match p-code nodes to source code, we use FAISS for approximate nearest neighbor retrieval. FAISS enables scalable similarity search through the hierarchical navigable small world (HNSW) graphs for efficient indexing, product quantization techniques for fast high-dimensional vector comparisons and inverted file indexing to accelerate nearest-neighbor searches.

Given a query embedding, FAISS retrieves the top-k most similar source code nodes, ensuring fast and scalable search over large repositories.

A.12 Hierarchical Clustering for Code Fragment Construction

After retrieving candidate nodes, we group them into coherent code fragments using agglomerative hierarchical clustering. Initially, each retrieved node is treated as an individual cluster. Clusters are then iteratively merged based on the shortest path proximity in the ICFG:

$$distance(C_1, C_2) = \max\{sp(x, y) \mid x \in C_1, y \in C_2\}. \tag{7}$$

Clusters remain separate if their shortest path distance exceeds a predefined threshold, ensuring only functionally related nodes are grouped.

A.13 Ranking Code Fragments

Retrieved code fragments are ranked using a confidence score:

$$\gamma = QC + \frac{1}{distance'}, \tag{8}$$

where QC represents the number of matched query nodes, and $distance'$ is the maximum shortest path distance within the cluster. Fragments with higher query coverage and stronger structural cohesion receive higher scores, prioritizing the most relevant implementations.

A.14 Dataset Selection and Processing

The dataset used in this study consists of 103 pseudo-code descriptions sourced from algorithm textbooks. The corresponding source code is drawn from two primary repositories: the Stony Brook Algorithm Repository, containing 67 C and 27 Java projects, and a curated set of 20 GitHub repositories, comprising 10 Java and 10 C projects. GitHub repositories were selected based on multiple criteria to ensure high-quality code representation. Only original repositories (not forks) were considered to maintain authenticity. Active maintenance was a key factor, with preference given to repositories with recent commits. Additionally, projects with at least 1000 GitHub stars were included to ensure relevance. Finally, all selected repositories were required to be compatible with Atlas for Interprocedural Control Flow Graph (ICFG) extraction.

A.15 Implementation Details

The GAE model was configured with two GCN layers, each producing 512-dimensional embeddings. The training was conducted using the Adam optimizer with a learning rate of $\eta = 0.01$ and a batch size of 2048. The dataset was split into 80% training and 20% validation, with early stopping applied based on validation loss to prevent overfitting.

For search and retrieval, the FAISS (Facebook AI Similarity Search) library was used to perform approximate nearest-neighbor searches efficiently. The retrieval process was based on cosine similarity, ensuring robust matching between pseudo-code and source-code representations. The system retrieved the top 100 most relevant results per query, allowing for high recall while maintaining computational efficiency.

A.15.1 Baseline Implementations and Training

To compare the effectiveness of *PC2SC*, we implemented and trained two state-of-the-art code search baselines: Deep Code Search (DCS) and OCoR. The DCS model was configured to extract method names, API sequences, and comments from source code to improve retrieval performance. OCoR, on the other hand, converted pseudo-code descriptions into comment-code pairs, enabling retrieval based on textual and semantic similarity.

All methods were evaluated using two primary metrics: Mean Reciprocal Rank (MRR), which measures how well the ranking system prioritizes correct results, and First Rank (F-Rank), which indicates the position of the first correct result in the ranked output. These metrics provide a comprehensive assessment of retrieval effectiveness and ranking accuracy.

A.16 Experimental Workflow

The experimental setup follows a structured workflow to evaluate *PC2SC* across different datasets and programming languages. Figure 4 illustrates the organization of evaluations, detailing the number of queried algorithms and the corresponding dataset partitions. The workflow includes data preprocessing, pseudo-code transformation into p-code, ICFG generation, model training, retrieval evaluation, and performance comparison against baseline methods. Following this structured approach ensures a rigorous and reproducible evaluation of *PC2SC* 's effectiveness in retrieving algorithm implementations from real-world codebases.

A.17 Detailed Experimental Results

This section compares *PC2SC* with the baselines discussing the full numerical results, including ranking performance.

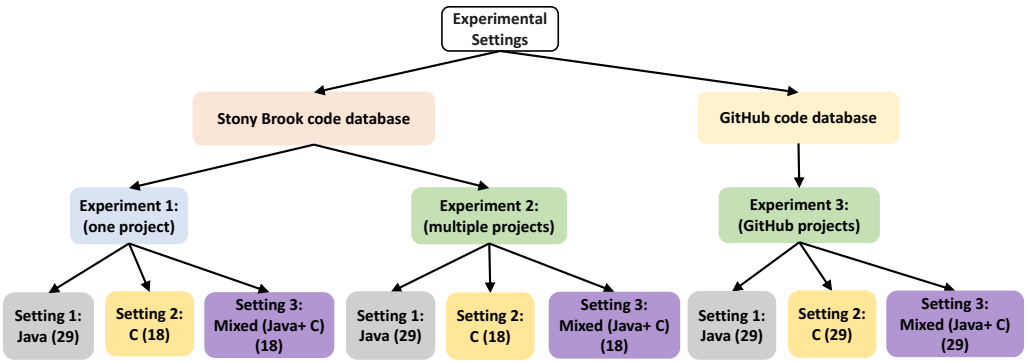

Figure 4: Experiment Workflow: Organization of evaluations. Numbers indicate the number of queried algorithms.

A.17.1 STONY BROOK CODE DATABASE

Tables 2, 3 and 4 display our results for the settings of Java, C, and mixed languages, respectively, on the Stony Brook code database. Each table lists the results for *single project* and *multiple projects* (described in Section 6.1.1). Under *Query* in the tables, we listed the algorithms with known implementations in the *Stony Brook Algorithm Repository*. Specifically, 29 and 18 algorithms are implemented in the Stony Brook code database for Java and C language, respectively. There are 18 algorithms that have both C and Java implementations (Table 4). Columns DCS and OCoR reported our baseline results, and Columns *CB+G* and *CB* provided results for our ablation studies (details in Section 6.1.4).

Table 2: The F-rank results for experiments on *Stony Brook code database* for the setting of Java language. For *single project*, the results show the mean over all the ground truth projects implementing the algorithm. The best-performing MRR result is shown in bold.

		single project					multiple projects				
No.	**Query**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**
1	any-segments-intersect	1	3.5	4.5	1.5	1.5	1	4	4	1	2
2	approx-vertex-cover	5	5	7	6	9	11	7	14	12	13
3	breadth-first-search	2.25	6	3	5	11.5	9	13	10	9	21
4	compute-transition-function	1	9	5	1	2	1	11	9	1	1
5	extend-shortest-paths	1.33	3.33	2.33	1.33	1.33	1	3	2	1	1
6	finite-automaton-matcher	2	12	6	2	2	5	15	9	6	12
7	floyd-warshall	1	12.33	4	1.33	3	1	21	14	2	4
8	graham-scan	1.5	1.5	2.5	2	3.5	1	2	3	3	5
9	hopcroft-karp	1.5	4	8.5	2.5	4	7	7	10	9	15
10	insertionsort	1	3	5	1	1	8	4	7	8	16
11	johnson	1.33	10.33	1.33	1.33	1.33	1	12	2	3	5
12	kruskal	1	17.75	14.75	1.25	1.5	1	23	18	2	9
13	lcs-length	1	3	1	2	10	1	5	1	1	2
14	lu-decomposition	1	1	2	1	2	16	2	6	16	18
15	maybe-mst	1.2	11.4	10.2	1.2	1.4	1	17	15	3	17
16	modular-linear-equation-solver	1	3	3	1	1	1	5	6	2	13
17	mst	1	1.2	1.2	1.2	1.4	1	1	1	1	1
18	mst-reduce	1.2	6	2.2	1.2	1.4	1	14	4	1	1
19	naïve-string-matcher	1	4	3	1	1	12	6	4	12	12
20	optimal-bst	1.25	9.25	12	4	18.5	5	14	17	5	28
21	pivot	1	2	3	1	1	1	3	7	2	10
22	prim	1	2.25	3.25	2.25	3.5	1	3	3	1	5
23	print_all_pair_shortest_paths	1.5	4	9.5	3.5	6	2	7	10	3	6
24	quicksort	1	5	4	4	8	49	51	55	51	63
25	rabin-karp-matcher	1	5	3	1	1	4	8	5	5	6
26	radixsort	1	3	2	2	6	3	5	3	4	14
27	recursive-activity-selector	1.5	1.5	1.5	1.5	1.5	1	1	1	2	8
28	slow_all_pair_shortest_pair	4.5	7.25	9	4.5	7.25	8	11	13	8	11
29	matrix_multiplication	1.33	2.33	2	1.33	1.33	3	5	8	6	13
	MRR	**0.819**	0.307	0.346	0.65	0.517	**0.606**	0.232	0.263	0.432	0.256

Experiment 1 (single project): In Tables 2 and 3, under *single project*, we observe that *PC2SC* found the implementation for 29 out of 29 queries from Java projects and 17 out of 18 queries from C projects in *Stony Brook code database*. Among these, we found implementation for 25 out of 29 queries from Java projects and 9 out of 18 queries from C projects as the first hit. Table 4 shows that *PC2SC* can have the *GAE* models trained based on the mixed languages of Java and C language

Table 3: The F-rank results for experiments on *Stony Brook code database* for the setting of C language. For *single project*, the results show the mean over all the ground truth projects implementing the algorithm. The best-performing MRR result is shown in bold.

		single project					multiple projects				
No.	**Query**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**
1	any-segments-intersect	1	3	2	1	1	16	18	29	16	16
2	approx-vertex-cover	1	1	3	5	32	1	1	3	6	41
3	breadth-first-search	14	6	15	17	32	23	8	19	28	>100
4	extend-shortest-paths	1.33	3	2.33	1.33	1.33	1	4	2	1	3
5	graham-scan	19	15.5	10.5	19.5	50	>100	15	15	>100	>100
6	hopcroft-karp	3	1	10	7	11	31	10	41	33	>100
7	insertionsort	1.5	1.5	1.5	1.5	1.5	1	19	24	1	1
8	kruskal	6.33	7	7.33	6.33	6.33	40	39	>100	40	>100
9	lcs-length	1	1	1	1	1	1	1	1	1	1
10	maybe-mst	7.2	14.4	16.2	8.2	8.6	17	21	26	19	>100
11	mst	1.4	7.2	8	1.4	2.2	7	10	14	9	9
12	mst-reduce	11.6	4.4	2.2	12	18.8	16	12	9	18	20
13	optimal-bst	1	4	6	2	2	11	14	14	12	13
14	prim	2	2.8	2.2	2.2	2.4	49	48	41	49	>100
15	quicksort	>100	>100	>100	>100	>100	>100	>100	>100	>100	>100
16	radixsort	1.5	2.5	4	1.5	1.5	1	3	5	1	2
17	recursive-activity-selector	1	11	13.5	20	21.5	31	43	47	35	>100
18	matrix_multiplication	4	6	7	4	9	10	12	15	12	14
	MRR	**0.522**	0.357	0.281	0.472	0.34	**0.316**	0.192	0.149	0.265	0.184

Table 4: The F-rank results for experiments on *Stony Brook code database* for the setting of mixed language. For *single project*, the results show the mean over all the ground truth projects implementing the algorithm. The best-performing MRR result is shown in bold.

		single project					multiple projects				
No.	**Query**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**
1	any-segments-intersect	10	12.33	16	10.33	12.33	12	16	19	14	18
2	approx-vertex-cover	1.5	1.5	3	1.5	2.5	1	3	4	2	5
3	breadth-first-search	6	6.2	4.2	6.2	9.4	12	13	6	13	16
4	extend-shortest-paths	1.33	4.33	2.16	3.16	5.33	1	5	2	3	5
5	graham-scan	9	7.25	9.25	9.25	10.25	14	12	16	14	16
6	hopcroft-karp	3.33	1.33	17	3.33	3.33	15	5	29	15	19
7	insertionsort	1.33	2	3.33	3	4.33	2	4	3	3	7
8	kruskal	1.28	6.42	7.14	2.14	3	2	13	12	2	3
9	lcs-length	1.5	3.5	1.5	3.5	5	1	3	1	3	4
10	maybe-mst	1.3	12.5	14.4	1.4	2.3	10	15	17	11	13
11	mst	1.4	2.3	1.4	1.4	3.2	3	3	2	4	7
12	mst-reduce	8.1	9.2	9.3	8.4	11.2	9	9	9	9	13
13	optimal-bst	2.2	6.4	8.2	3.4	4.2	6	12	14	6	7
14	prim	2.22	3	2.22	3.22	3.33	14	21	19	14	14
15	quicksort	10.33	13.33	12	12.33	15	56	59	57	57	71
16	radixsort	1.33	1.33	1.33	2.33	5	9	7	5	9	9
17	recursive-activity-selector	1.25	9.25	11	2.25	2.5	4	13	17	6	9
18	matrix_multiplication	2	2.25	2.25	2.25	2.5	6	8	10	8	11
	MRR	**0.498**	0.304	0.289	0.349	0.243	**0.313**	0.146	0.203	0.189	0.122

and retrieve the implementations from both Java and C projects. In this setting, *PC2SC* found the implementations for 9 out of 18 queries as the first hit. Overall, across all the settings in Tables 2, 3, and 4, except for one query, *PC2SC* found all the implementations of known queries in the top 25 code snippets. In terms of MRR, *PC2SC* performed the best on Java, followed by the mixed language model, followed by C. This is likely caused by the fact that CodeBERT is pre-trained on Java, so it can model Java features better.

PC2SC outperforms the baselines *DCS* and *OCoR* for all the queries in Table 2, for most queries in Table 3 except for queries No.3, No.5, No.6, and No.12, and for most queries in Table 4 except for queries No.5 and No.6. Among these queries, *DCS* performed the best for queries No.3 and No.6 in Table 3 and No.5 and No.6 queries in Table 4 while *OCoR* performed the best on the remaining. In these cases, the value of the F-rank is lower for *PC2SC* because *PC2SC* has more candidate code fragments to rank—-the baselines report methods and *PC2SC* reports code fragments; in *PC2SC*, one method can have several code fragments in the ranking. In return, *PC2SC* can more accurately locate the algorithm implementations within a function or across multiple functions.

PC2SC achieved significantly higher MRR than both baselines. Specifically, *PC2SC* outperformed *DCS* by a margin of 167% and *OCoR* by a margin of 137% for the setting of Java language. For the setting of C language, *PC2SC* outperformed *DCS* by a margin of 46% and *OCoR* by a margin of 86%. For the setting of mixed language, *PC2SC* outperformed *DCS* by a margin of 64% and *OCoR* by a margin of 72%.

Table 5: The F-rank results for experiments on *GitHub code database* for the setting of Java language. The best performing MRR result is shown in bold.

		GitHub projects (Java)				
No.	**Query**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**
1	any-segments-intersect	>100	>100	>100	>100	>100
2	approx-vertex-cover	1	5	3	2	4
3	breadth-first-search	1	7	5	1	2
4	compute-transition-function	2	4	5	4	13
5	extend-shortest-paths	4	21	12	5	10
6	finite-automaton-matcher	1	9	13	2	73
7	floyd-warshall	8	10	9	8	12
8	graham-scan	1	9	5	1	4
9	hopcroft-karp	1	7	5	3	8
10	insertionsort	1	2	3	1	2
11	johnson	>100	>100	>100	>100	>100
12	kruskal	9	15	12	10	11
13	lcs-length	1	16	7	1	2
14	lu-decomposition	10	9	7	12	>100
15	maybe-mst	1	8	7	1	2
16	modular-linear-equation-solver	13	18	21	13	>100
17	mst	1	16	12	2	4
18	mst-reduce	>100	>100	>100	>100	>100
19	naïve-string-matcher	1	11	21	2	3
20	optimal-bst	>100	>100	>100	>100	>100
21	pivot	>100	>100	>100	>100	>100
22	prim	1	6	6	4	9
23	print_all_pair_shortest_paths	1	6	11	1	1
24	quicksort	1	11	16	2	4
25	rabin-karp-matcher	1	3	2	1	2
26	radixsort	16	21	17	20	60
27	recursive-activity-selector	3	12	5	5	11
28	slow_all_pair_shortest_pair	1	14	9	1	2
29	matrix_multiplication	5	10	6	6	6
	MRR	**0.58**	0.113	0.132	0.427	0.217

Table 6: The F-rank results for experiments on *GitHub code database* for the setting of C and mixed languages. The best performing MRR result is shown in bold.

		GitHub projects (C)					GitHub projects (mixed)				
No.	**Query**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**	***PC2SC***	**DCS**	**OCoR**	**CB+G**	**CB**
1	any-segments-intersect	>100	>100	>100	>100	>100	>100	>100	>100	>100	>100
2	approx-vertex-cover	10	13	13	12	18	3	8	5	9	12
3	breadth-first-search	3	>100	2	5	16	2	34	47	3	15
4	compute-transition-function	10	18	21	10	19	4	24	25	6	15
5	extend-shortest-paths	1	4	4	1	2	2	19	9	2	3
6	finite-automaton-matcher	2	12	8	3	6	2	10	11	4	16
7	floyd-warshall	4	14	12	6	13	6	22	20	10	18
8	graham-scan	5	13	7	8	28	4	10	6	6	26
9	hopcroft-karp	1	13	12	1	3	1	15	18	4	8
10	insertionsort	6	1	2	8	20	4	4	5	5	16
11	johnson	18	>100	>100	24	32	26	>100	>100	42	72
12	kruskal	41	12	16	42	48	12	14	13	15	18
13	lcs-length	>100	>100	4	>100	>100	2	62	8	2	4
14	lu-decomposition	8	9	8	9	13	9	9	8	13	18
15	maybe-mst	3	14	10	4	20	2	16	12	4	15
16	modular-linear-equation-solver	>100	>100	>100	>100	>100	15	21	23	18	>100
17	mst	3	12	9	4	21	2	12	14	3	12
18	mst-reduce	>100	>100	>100	>100	>100	>100	>100	>100	>100	>100
19	naïve-string-matcher	2	9	6	2	4	2	11	12	4	5
20	optimal-bst	>100	>100	>100	>100	>100	>100	>100	>100	>100	>100
21	pivot	23	38	42	26	>100	36	54	65	43	>100
22	prim	1	7	16	2	36	1	6	9	3	34
23	print_all_pair_shortest_paths	13	10	12	17	21	4	19	11	4	5
24	quicksort	1	11	9	1	18	1	12	12	1	14
25	rabin-karp-matcher	2	3	2	3	5	2	3	2	2	4
26	radixsort	15	24	26	18	45	18	21	19	21	52
27	recursive-activity-selector	2	9	13	2	16	3	9	7	4	14
28	slow_all_pair_shortest_pair	27	24	21	31	>100	4	16	12	4	6
29	matrix_multiplication	1	6	4	3	28	2	8	6	3	26
	MRR	**0.321**	0.113	0.133	0.249	0.08	**0.345**	0.081	0.098	0.229	0.087

Experiment 2 (multiple projects): In Tables 2, 3, and 4, under *multiple projects*, we can observe that within the top 25 of the ranked lists, *PC2SC* found the implementation for 28 out of 29 queries from Java projects, 12 out of 18 queries from C projects, and 17 out of 18 queries from mixed language projects in Stony Brook code database. *PC2SC* outperformed the baselines *DCS* and *OCoR* for 40 queries out of the 65 queries. *OCoR* won for query No.10 in Table 2 and No.11 and No.16 in Table 4, and *DCS* performed the best for the remaining. In terms of MRR, *PC2SC* outperformed

DCS by a margin of 161% and *OCoR* by a margin of 130% for the setting of Java language. For the setting of C language, *PC2SC* outperformed *DCS* by a margin of 65% and *OCoR* by a margin of 112%. For the setting of mixed language, *PC2SC* outperformed *DCS* by a margin of 114% and *OCoR* by a margin of 54%.

A.17.2 GitHub code database

In Tables 5 and 6, under *GitHub projects*, we show that within the top 25 of the ranked lists, *PC2SC* found the implementation for 24 out of 29 queries from Java projects, for 24 out of 29 queries from C projects, and for 26 out of 29 queries from mixed language projects. Excluding the 3 queries that none of the methods retrieved implementations in top 100 results, *PC2SC* outperformed the baselines *DCS* and *OCoR* for 62 of the 84 queries. *OCoR* performed the best for 3 out of 84 queries, while *DCS* performed the best for 1 out of 84 queries. We observed that the library calls and complex data structure initialization could lead to the mismatch of pseudo code and source code in our approach. In terms of MRR, *PC2SC* outperformed *DCS* by a margin of 413% and *OCoR* by a margin of 339% for the setting of Java language. For the setting of C language, *PC2SC* outperformed *DCS* by a margin of 184% and *OCoR* by a margin of 141%. For the setting of mixed language, *PC2SC* outperformed *DCS* by a margin of 326% and *OCoR* by a margin of 252%.

Overall, we observe that the baselines perform well when the queries and code are more similar in natural languages because the baselines are trained considering the queries to be natural language. However, *PC2SC* considers the control flow, maths, and natural language features; thus, it is more robust and general for all types of pseudo code queries.

A.17.3 Ablation results

Tables 2, 3, 4, 5, and 6 show that *PC2SC* performs better than *CB+G* and *CB*. For example, under *single project*, for 36 out of 65 queries, *PC2SC* outperformed *CB+G*, and for 45 out of 65 queries, *CB+G* is better than *CB*. The results demonstrated that control flow and math information are useful. We also found that for the setting of the Java language, *CB* can find algorithm implementations better than our baselines. For the setting of the C language, CB performed poorly compared to the baselines, as CB uses encoding from CodeBERT, and CodeBERT is not trained in the C language. We observed that *CB+G* is significantly better than *CB* on the setting of C language, indicating that the control flow is very useful for search.

A.18 Examples

In Figures 5, 6, and 7, we show a few examples of algorithm implementations found from the *Stony Brook code database* and *GitHub code database*. The implementations of the *floyd-warshall*, *lcs-length*, and *insertion sort* algorithms are the first ranked in *PC2SC* output list. The figures show that pseudo-code and implementations are comparable but not exactly the same. For example, in Figure 5b, lines 2–5 initialize the matrix `sums` (`D` at line 3 in Figure 5a), and lines 7—11 implement lines 4–7 in Figure 5a. When manually confirming these code fragments, we found that the control flow and math operators are important here. The figures also show that mapping the code and natural language to the same vector space is important and helpful for search. For example, pseudo code in Figure 6a initializes matrices using natural language at line 4, and in Figure 6b, source code at lines 3—4 implement line 4 in Figure 6a. Pseudo code in Figure 7a uses "$<$" at line 5, and source code in Figure 7b uses the function "`less`" at line 4. Since we map both code and natural language to the same vector space using CodeBERT, *PC2SC* is able to retrieve such relevant code.

A.19 Dataset Limitations and Generalizability

While our dataset includes a variety of algorithms, it remains biased toward well-documented, classical algorithms from textbooks. Many real-world applications use highly domain-specific algorithms (e.g., in cryptography, computational biology, or machine learning), which are not covered in our dataset. Expanding the dataset to include more diverse algorithm descriptions from research papers, software documentation, and technical blogs could improve generalizability.

Additionally, the open-source repositories we used primarily consist of educational and general-purpose projects. Industry codebases, especially in large-scale commercial software, often contain proprietary optimizations, unconventional naming conventions, and non-standard coding styles that

```
FLOYD-WARSHALL(W)
n = W.rows
D=W
for k = 1 to n
    for i = 1 to n
        for j =1  to n
            d_ij = min(d_ij , d_ik+d_kj)
return D
```

(a) Pseudo code

```
...
final int[][] sums = new int[vertices.
    size()][vertices.size()];
for (int i = 0; i < sums.length; i++) {
    for (int j = 0; j < sums[i].length;
     j++) {
        sums[i][j] = Integer.MAX_VALUE;
    }}
...
for (int k = 0; k < vertices.size(); k+
    +) {
    for (int i = 0; i < vertices.size()
    ; i++) {
        for (int j = 0; j < vertices.
    size(); j++) {
            ...
            final int summed = (ikCost !=
    Integer.MAX_VALUE && kjCost !=
    Integer.MAX_VALUE) ? (ikCost +
    kjCost) : Integer.MAX_VALUE;
            ...
...
return allShortestPaths;
...
```

(b) Real-World Implementation, a segment from a function

Figure 5: `Floyd Warshall` in *Stony Brook code database*

may affect retrieval performance. Future work could include evaluating *PC2SC* on closed-source codebases through collaborations with software companies.

A.20 Baseline Selection and Alternative Approaches

We selected DCS and OCoR as baselines due to their state-of-the-art performance in neural code search and the availability of reproducible implementations. However, alternative approaches could provide additional insights:

1. Code Clone Detection Tools (e.g., DECKARD, NiCad, AST-based matching): Traditional clone detection methods might be effective at retrieving syntactically or structurally similar functions but would struggle with conceptual algorithm matching when implementations differ significantly in structure.
2. Large Language Models (e.g., GPT-4, Codex, StarCoder): Modern LLMs trained on code might offer zero-shot retrieval capabilities, but their reliance on pre-trained datasets means they may not generalize well to less common or mathematically dense algorithms.
3. Graph-Based Search Models: Future work could explore hybrid models that explicitly learn interprocedural control flow patterns using more advanced graph neural networks (GNNs).

A.21 Evaluation Metrics and Potential Biases

While we used F-rank and MRR as primary evaluation metrics, it is important to note potential biases:

1. Method-Level vs. Fragment-Level Ranking: Baseline methods return entire functions, whereas *PC2SC* retrieves fine-grained code fragments. This means that if an algorithm is spread across multiple functions, baselines may have an unfair advantage in F-rank scores.
2. Ranking-Based Metrics vs. Precision-Based Metrics: Alternative ranking metrics like Normalized Discounted Cumulative Gain (NDCG) could be useful to weigh the relevance of multiple retrieved results rather than just the first correct result.
3. Error Analysis on False Positives and Negatives: A deeper breakdown of misclassified or partially correct results could help understand where retrieval errors occur. For example,

```
LCS-LENGTH(X,Y)
m = X.length
n = Y.length
let b[1..m,1..n] and c[0..m,0..n]
be new tables
for i = 1 to m
    c[i,0] = 0
for j = 0 to n
    c[0,j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i,j] = c[i-1,j-1] + 1
            b[i,j] = "\nwarrow"
        else if c[i-1,j] \geq c[i,j-1]
            c[i,j] = c[i-1,j]
            b[i,j] = "\uparrow"
        else if c[i,j] = c[i,j-1]
            b[i,j] = "\leftarrow"
return c and b
```

(a) Pseudo code

```
int n = a.length();
int m = b.length();
int S[][] = new int[n+1][m+1];
int R[][] = new int[n+1][m+1];
int ii, jj;
for(ii = 0; ii <= n; ++ii) {
    S[ii][0] = 0;
    R[ii][0] = UP;}
for(jj = 0; jj <= m; ++jj) {
    S[0][jj] = 0;
    R[0][jj] = LEFT;}
for(ii = 1; ii <= n; ++ii) {
    for(jj = 1; jj <= m; ++jj) {
        if( a.charAt(ii-1) == b.
            charAt(jj-1) ) {
            S[ii][jj] = S[ii-1][jj-1]
                + 1;
            R[ii][jj] = UP_AND_LEFT;}
        else {
            S[ii][jj] = S[ii-1][jj-1]
                + 0;
            R[ii][jj] = NEITHER;}
        if( S[ii-1][jj] >= S[ii][jj]
            ) {
            S[ii][jj] = S[ii-1][jj];
            R[ii][jj] = UP;}
        if( S[ii][jj-1] >= S[ii][jj]
            ) {
            S[ii][jj] = S[ii][jj-1];
            R[ii][jj] = LEFT;}}}
...
return new String(lcs);
```

(b) Real-World Implementation, a segment from a function

Figure 6: `LCS length` in *Stony Brook Algorithm Repository*

```
InsertionSort(A)
A[0] = -\infinity
for i = 2 to n
    j = i
    while A[j] < A[j - 1]
        swap(A[j],A[j-1])
        j = j - 1
```

(a) Pseudo code

```
for (int i = 1; i < array.length; i++) {
    T insertValue = array[i];
    int j;
    for (j = i - 1; j >= 0 && less(
    insertValue, array[j]); j--) {
        array[j + 1] = array[j];
    }
    if (j != i - 1) {
        array[j + 1] = insertValue;
    }
}
return array;
```

(b) Real-World Implementation, an entire function

Figure 7: `Insertion sort` in *GitHub code database*

functionally equivalent but structurally different implementations might be unfairly penalized in the ranking evaluation.

A.22 Future Work and Mitigation Strategies

To further improve robustness and mitigate threats to validity, we suggest the following future directions:

1. Expanding the dataset to include more diverse sources, such as research papers, competitive programming datasets, and industry codebases.
2. Exploring hybrid retrieval methods that combine traditional program analysis techniques with deep learning models.
3. Developing better ranking models that account for partial correctness and multiple function implementations, beyond simple first-hit ranking.
4. Comparing *PC2SC* against large-scale code embedding models (e.g., CodeT5, GraphCodeBERT) to assess the effectiveness of learned graph-based representations vs. transformer-based embeddings.

These steps would provide a more comprehensive evaluation framework and further validate the practical utility of *PC2SC* for real-world algorithm retrieval.