Revisiting Code Similarity through Optimal Module Alignment

Anonymous ACL submission

Abstract

Code retrieval for programs remains challenging due to structural misalignment of fine-grained similarity metrics. In this paper, we propose the Length-Aware Optimal Module Matching (LOMM) framework that aligns code modules across programs and aggregates similarity scores in a global optimal manner. The proposed method is model-agnostic and can be applied to both embedding-based retrieval and structural metrics.

We evaluate our approach on two datasets, including LongCCD, a new long-code retrieval dataset designed with a large corpus-to-query ratio to encourage model-specific optimal matching. Across diverse embedding models, Our method consistently improves retrieval performance, yielding relative gains of approximately 15–20% in NDCG@5 compared to monolithic baselines.

1 Introduction

Code-to-code retrieval plays a central role in modern code search systems, enabling tasks such as code reuse, refactoring, plagiarism detection, and cross-language retrieval. Most existing approaches rely heavily on pretrained code embeddings, which represent an entire program as a single, fixed-length vector. While such representations are effective for capturing surface-level semantic similarity, they tend to abstract away the internal structure of programs. In particular, embeddings alone may overlook crucial structural aspects of code, such as the abstract syntax tree (AST) (Wang et al., 2022a; Zhang et al., 2025), control flow, and compositional relationships between code components, all of which are essential for accurately modeling program semantics.

We propose an order-agnostic bipartite module matching framework called the Length-Aware Optimal Module Matching (LOMM) framework that explicitly accounts for code structure by decomposing programs into a set of modules and aligning them via an optimal bipartite matching algorithm (Kuhn, 1955). Figure 1 illustrates the proposed LOMM framework. Instead of comparing programs as monolithic entities, our method performs module-level alignment, allowing fine-grained similarities between corresponding components to be identified and aggregated. This formulation naturally accommodates differences in code length and modular organization, which are common in real-world codebases and long-code scenarios.

A key advantage of LOMM is its model-agnostic nature. It can seamlessly incorporate both embedding-based similarity measures and structure-aware metrics, such as *Tree Similarity of Edit Distance* (TSED) (Song et al., 2024). As a result, the proposed approach preserves rich structural information while still benefiting from the expressive power of learned representations. This flexibility makes the framework applicable across a wide range of code embedding models without requiring retraining or architectural modifications.

Our contributions are threefold: (i) an optimal bipartite module alignment framework for code-to-code retrieval; (ii) the introduction of LongCCD, a long-code retrieval dataset with a corpus-heavy design to emphasize top-rank performance; and (iii) extensive evaluation across diverse embedding models, demonstrating consistent retrieval improvements of up to approximately 15% in NDCG@1, while preserving structural code semantics and enabling efficient module-level computation for expensive similarity metrics such as TSED.

2 Related Work

2.1 Code Similarity Computation

Code similarity and clone detection aim to identify semantically or syntactically similar code snippets. Traditional approaches often rely on syntax-based metrics, token matching, or tree-based represen-

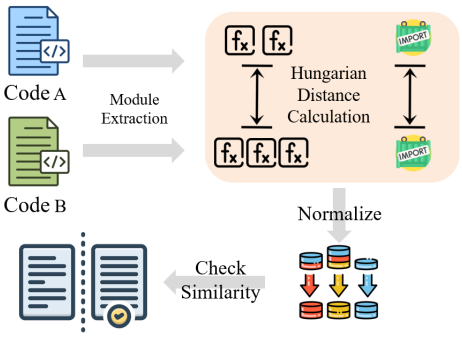

Figure 1: Overview of the proposed LOMM framework

tations such as ASTs and program dependence graphs (Sneha et al., 2025). They tend to be computationally expensive for large codebases (Fan et al., 2024) and may struggle to capture semantic equivalence when functionally equivalent programs differ significantly in structure.

These challenges motivate us to investigate similarity measures that are both structurally-aware and computationally tractable.

2.2 Code Search and Retrieval

Code search and retrieval address the following problem: given a query code snippet, retrieve the most relevant code from a large corpus.

Recent work leverages pretrained embeddings from LLMs to represent code as dense vectors (Feng et al., 2020). However, monolithic code embeddings often overlook fine-grained structural correspondences, limiting their effectiveness on long or modular code. This highlights the need for similarity computation methods that can exploit fine-grained correspondences of modules while remaining computationally efficient at scale.

3 Problem Definition

Given two codes C_A and C_B, our goal is to compute a semantic similarity score $Sim(C_A, C_B) \in [0, 1]$. Code often consists of two distinct structural components: (1) Order-agnostic modules ($\mathcal{M}$): Independent units such as functions, classes, where the relative declaration order does not strictly alter the program's functionality. (2) Order-sensitive statements ($\mathcal{S}$): Sequential statements such as variable declarations where the execution order is semantically critical.

We define the problem as finding an optimal alignment between the sets of modules $\mathcal{M}_A$ and $\mathcal{M}_B$ while simultaneously comparing the ordered statements $\mathcal{S}_A$ and $\mathcal{S}_B$, using a pre-trained LLM to bridge the syntactic gap.

4 Proposed Method

We propose an *Length-Aware Optimal Module Matching (LOMM)* framework. This method decomposes code into modular and sequential components via AST parsing, computes local semantic distances using LLM embeddings, and aggregates these distances based on token-length weights to ensure that longer, more information-dense modules contribute more to the final score.

4.1 Structural Decomposition and Grouping

First, we parse each code using `tree-sitter` to extract the AST. Then, we categorize AST nodes into two groups:

First, a module set $M = \{m_1, m_2, \ldots, m_k\}$ contains independent definitions such as functions, classes, and imports. These are grouped by their node type (e.g., separate groups for methods and classes). Second, a remainder sequence R is a single concatenated string of all order-sensitive nodes (e.g., identifier declaration, global variables) that are not part of the module set.

For every component x, we obtain a vector representation $\mathbf{e}_x$ using an LLM. Simultaneously, we calculate its token length $|x|$ using the tokenizer. The semantic distance between two components x and y is defined as the cosine distance:

$$d(x, y) = 1 - \frac{\mathbf{e}_x \cdot \mathbf{e}_y}{\|\mathbf{e}_x\|\|\mathbf{e}_y\|}. \quad (1)$$

4.2 Length-Aware Optimal Module Bipartite Matching (LOMM) Framework

For each module type (e.g., `import_statement` in C++ and `function_definition` in C++ and Python), we construct a bipartite graph between the modules of C_A and C_B. Let $M_A = \{a_1, \ldots, a_n\}$ and $M_B = \{b_1, \ldots, b_m\}$ be the sets of modules of a specific type. We define a cost matrix $C \in \mathbb{R}^{n \times m}$ where $C_{ij} = d(a_i, b_j)$.

We employ the Hungarian algorithm (Kuhn, 1955) to find the optimal assignment π that minimizes the total distance. Unmatched modules are treated as having a distance of 1.0 (maximum semantic dissimilarity).

Unlike standard averaging, we weigh the contribution of each matching based on the information content, approximated by the sum of token lengths of the involved components. The weighted distance

contribution for the matched modules is:

$$D_{\text{modules}} = \sum_{(i,j)\in\pi} d(a_i, b_j) \cdot (|a_i|+|b_j|). \quad (2)$$

For any unmatched module u, the penalty is its full token length:

$$D_{\text{unmatched}} = \sum_{u\in\text{unmatched}} |u|. \quad (3)$$

The order-sensitive remainders R_A and R_B are compared directly as single units:

$$D_{\text{remainder}} = d(R_A, R_B) \cdot (|R_A|+|R_B|). \quad (4)$$

If one side lacks a remainder, it is treated as an unmatched component.

4.3 Final Similarity Score

The total weighted distance is the sum of modules, unmatched, and remainder distances. We normalize this by the total token count of both code snippets to obtain a normalized distance, which is then converted to a similarity score:

$$\text{Sim}(C_A, C_B) = 1 - \frac{D_{\text{modules}} + D_{\text{unmatched}} + D_{\text{remainder}}}{\sum_{x\in C_A\cup C_B} L(x)} \quad (5)$$

5 Experiments

5.1 Datasets

We employ two datasets that differ in programming language, code length, and structural complexity.

- CodeTransOcean-DL: For python, we use CodeTransOcean-DL from the COIR benchmark suite (Li et al., 2025).
- LongCCD: For C++, we introduced our benchmark LongCCD originated form CodeContest+ (Wang et al., 2025). LongCCD (short for Long Code Clone Detection) is designed to evaluate retrieval performance on *long and structurally complex programs*, where global embeddings often struggle to preserve fine-grained semantics.

5.2 Evaluation Metrics

We evaluate retrieval performance using Normalized Discounted Cumulative Gain (NDCG) at different cutoff levels, depending on the benchmark.

For LongCCD, we report NDCG@1 and NDCG@5. We emphasize NDCG@1 to assess whether the top-ranked result corresponds to the most semantically appropriate match. Since different retrieval models exhibit different semantic behaviors, focusing on the top-ranked result provides a clearer comparison of their ability to identify the single best candidate, rather than multiple acceptable alternatives. NDCG@5 is additionally reported to evaluate ranking stability beyond the first position.

For CodeTransOcean-DL, we follow the evaluation protocol of the original COIR benchmark and report NDCG@10. To ensure fair comparison and consistency with prior work, we adopt the same evaluation metric while additionally reporting NDCG@5 for finer-grained analysis.

5.3 Baseline Methods

We evaluate a diverse set of code embedding models, including `All-MiniLM-L6-v2`, `GTE-Base` (Zhang et al., 2024), `BGE-Base` (Xiao et al., 2023), `E5-Base` (Wang et al., 2022b), `Contriever` (Lei et al., 2023), `bge-m3` (Chen et al., 2024), and `OpenAI-text-embedding-3-large`,

These models span a wide range of architectures and training objectives, including general-purpose text embeddings, code-specialized embeddings, and retrieval-optimized models.

We also employ the Tree Similarity of Edit Distance (TSED) (Song et al., 2024) as a baseline method for computing the similarity of codes.

5.4 Results and Analysis

Table 1 presents code retrieval performance on the proposed datasets CodeTransOcean-DL and LongCCD. We compare monolithic code retrieval baselines to those with the LOMM framework, across multiple embedding models.

Effectiveness of LOMM Framework On both CodeTransOcean-DL and LongCCD, Hungarian-based aggregation consistently improves retrieval accuracy over monolithic representations for most embedding models. The improvements are particularly pronounced in NDCG@5 and NDCG@10 on CodeTransOcean-DL, indicating that decomposing code into modular units and aggregating them via bipartite matching enables more accurate structural alignment. In contrast, monolithic embeddings often fail to capture correspondences between large code snippets.

Model-Agnostic Improvement The performance gains from LOMM are observed across

Table 1: Code retrieval performance across datasets, embedding models, and similarity computation methods.

Model (param.)	Method	CodeTansOcean-DL			LongCCD		
		NDCG@5	NDCG@10	Time (s)	NDCG@1	NDCG@5	Time (s)
`All-MiniLM` (22M)	Monolithic	17.61	27.94	**46.14**	30.00	19.59	**45.00**
	LOMM	**19.62**	**34.42**	71.96	**43.33**	**27.72**	56.98
`GTE-Base` (110M)	Monolithic	19.70	27.27	**36.56**	19.70	27.27	**36.56**
	LOMM	**20.06**	**35.04**	57.61	**46.67**	**28.62**	72.90
`BGE-Base` (110M)	Monolithic	14.61	22.84	**30.41**	26.67	16.77	**51.11**
	LOMM	**18.22**	**33.78**	54.37	**33.33**	**24.39**	66.37
`E5-Base` (110M)	Monolithic	13.48	22.06	**48.96**	**36.67**	**25.78**	**50.96**
	LOMM	**18.43**	**32.25**	75.36	**36.67**	23.06	66.67
`Contriever` (110M)	Monolithic	16.05	22.91	**30.23**	26.67	18.51	**50.37**
	LOMM	**18.80**	**33.48**	53.30	**43.33**	**26.59**	66.77
`BGE-M3` (567M)	Monolithic	16.73	29.55	**59.82**	46.67	30.73	**61.20**
	LOMM	18.97	**34.31**	85.11	**60.00**	**36.56**	87.74
`OpenAI-text-embedding-3-large`	Monolithic	17.39	33.27	**402.47**	63.33	**63.34**	**401.36**
	LOMM	**18.51**	**33.67**	572.47	**66.67**	44.37	1522.00
TSED	Monolithic	21.41	35.90	11,611.57	33.33	34.89	30,744.15
	LOMM	**22.02**	**36.23**	**327.24**	**40.00**	**35.11**	**645.81**

a wide range of embedding models, including lightweight models (e.g., `All-MiniLM-L6-v2`), mid-sized encoders (e.g., `BGE-Base`, `E5-Base`), and larger proprietary embeddings. This indicates that the LOMM framework is largely orthogonal to the choice of embedding model and functions as a complementary structural enhancement rather than a model-specific optimization.

Notably, even models with strong monolithic performance on CodeTransOcean-DL benefit from Hungarian aggregation, suggesting that structural decomposition remains valuable despite improved embedding capacity.

Computational Efficiency LOMM incurs additional computational overhead due to pairwise module matching. When used with embedding-based models, the overhead mainly stems from computing and aggregating module-level similarities, resulting in extra cost compared to monolithic embeddings. Nevertheless, for most embedding-based models, the runtime increase is moderate and often justified by the observed retrieval accuracy gains.

In contrast, for computationally expensive structural metrics such as TSED with $O(n^3)$ complexity, LOMM can improve practical efficiency. By decomposing programs into smaller modules, it reduces the cost of individual similarity computations, while enabling parallel execution and effective caching across queries. As a result, LOMM can substantially reduce end-to-end runtime for TSED, acting as a computational accelerator rather than an overhead for high-cost similarity measures.

Summary In summary, results on both CodeTransOcean-DL and LongCCD demonstrate that LOMM provides a robust and model-agnostic improvement for code retrieval. While LOMM introduces modest overhead when paired with embedding-based models, it can substantially reduce runtime for expensive structural metrics through decomposition, parallelization, and caching. The gains are especially pronounced for long and structurally complex code, validating the effectiveness of the proposed approach beyond monolithic embedding-based retrieval.

6 Conclusions

In this work, we proposed LOMM, an order-agnostic, modular aggregation framework based on optimal bipartite matching. By decomposing code into semantically meaningful units and aligning them explicitly, our approach overcomes key limitations of classical tree edit distance and monolithic embedding-based retrieval.

Extensive experiments on CodeTransOcean-DL and the newly introduced LongCCD dataset demonstrate that LOMM consistently improves retrieval accuracy across diverse embedding models. The gains are particularly pronounced for long and structurally complex code, where global representations often fail to capture semantics.

Limitations

While our proposed LOMM framework demonstrates significant improvements in retrieval accuracy, specifically for long and complex code, it comes with certain limitations.

First, the method introduces *additional computational overhead*. The pairwise similarity calculation between modules and the subsequent Hungarian matching algorithm (typically $O(n^3)$ complexity) increase inference latency. This computational cost can be a bottleneck for real-time retrieval applications requiring millisecond-level responses on massive-scale corpora.

Second, our approach relies on the *availability and correctness of ASTs*. The decomposition process assumes that the input code is syntactically valid. For code snippets with syntax errors, the method may fail to extract meaningful structural modules.

References

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *Preprint*, arXiv:2402.03216.

Dayi Fan, Rubao Lee, and Xiaodong Zhang. 2024. X-ted: Massive parallelization of tree edit distance. *Proc. VLDB Endow.*, 17(7):1683–1696.

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

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Harold W Kuhn. 1955. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97.

Yibin Lei, Liang Ding, Yu Cao, Changtong Zan, Andrew Yates, and Dacheng Tao. 2023. Unsupervised dense retrieval with relevance-aware contrastive pre-training. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 10932–10940, Toronto, Canada. Association for Computational Linguistics.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. 2025. CoIR: A comprehensive benchmark for code information retrieval models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 22074–22091, Vienna, Austria. Association for Computational Linguistics.

Soily Ghosh Sneha, Sadia Niha, and Dr. Md. Manzurul Hasan. 2025. Evaluating code clone detection and management: A comprehensive comparison among different techniques and tools along with some effective future directions. In *Proceedings of the 3rd International Conference on Computing Advancements*, ICCA '24, page 207–215, New York, NY, USA. Association for Computing Machinery.

Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé Bissyandé, and Jacques Klein. 2024. Revisiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, page 38–46. Association for Computational Linguistics.

Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022a. Unified abstract syntax tree representation learning for cross-language program classification. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, page 390–400, New York, NY, USA. Association for Computing Machinery.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. 2022b. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. 2025. CodeContests+: High-quality test case generation for competitive programming. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 5576–5600, Suzhou, China. Association for Computational Linguistics.

Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-pack: Packaged resources to advance general chinese embedding. *Preprint*, arXiv:2309.07597.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. 2023. CodeTransOcean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5067–5089, Singapore. Association for Computational Linguistics.

Xin Zhang, Yanzhao Zhang, Dingkun Long, Wen Xie, Ziqi Dai, Jialong Tang, Huan Lin, Baosong Yang, Pengjun Xie, Fei Huang, Meishan Zhang, Wenjie Li, and Min Zhang. 2024. mgte: Generalized long-context text representation and reranking models for multilingual text retrieval. *Preprint*, arXiv:2407.19669.

Yilin Zhang, Xinran Zhao, Zora Zhiruo Wang, Chenyang Yang, Jiayi Wei, and Tongshuang Wu.

2025. cAST: Enhancing code retrieval-augmented generation with structural chunking via abstract syntax tree. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 8106–8116, Suzhou, China. Association for Computational Linguistics.

7 Appendix

A Algorithm Pseudocode

Algorithm 1 Length-Aware Optimal Module Matching

Require: Code C_A, C_B, Embedding Model Φ
Ensure: Similarity Score $S \in [0, 1]$

Step 1: Parse and Embed
$M_A, R_A \leftarrow \text{TreeSitterParse}(C_A)$
$M_B, R_B \leftarrow \text{TreeSitterParse}(C_B)$
Compute embeddings $\mathbf{e}$ and lengths L for all components in M_A, M_B, R_A, R_B using Φ.
Step 2: Initialize Variables
TotalDist $\leftarrow 0$
TotalLen $\leftarrow \sum_{x \in M_A \cup R_A} L(x) + \sum_{y \in M_B \cup R_B} L(y)$
Step 3: Process Modules (by type)
for each type T in Modules **do**
  Get modules $U = M_A[T], V = M_B[T]$
  Construct Cost Matrix $C_{ij} = d(U_i, V_j)$
  $\pi \leftarrow \text{HungarianAlgorithm}(C)$
  for $(i, j) \in \pi$ **do**
    TotalDist $\leftarrow$ TotalDist $+ C_{ij} \times (L(U_i) + L(V_j))$
  end for
  for unmatched $k \in U \cup V$ **do**
    TotalDist $\leftarrow$ TotalDist $+ 1.0 \times L(k)$
  end for
end for
Step 4: Process Remainder
if R_A exists AND R_B exists **then**
  TotalDist $\leftarrow$ TotalDist $+ d(R_A, R_B) \times (L(R_A) + L(R_B))$
else
  TotalDist $\leftarrow$ TotalDist $+ \sum_{r \in \{R_A, R_B\}} L(r)$
end if
Step 5: Normalize
return $1 - (\text{TotalDist}/\text{TotalLen})$

A.1 LongCCD Dataset Construction and Evaluation Rationale

LongCCD is constructed from the CodeContests+(Wang et al., 2025) by selecting C++ programming problems whose solutions exhibit substantial length and structural complexity. Specifically, for each problem, we rank solutions by their average code length and select problems whose average solution length falls beyond the 800th percentile among all problems. This procedure ensures

that LongCCD focuses on genuinely long programs rather than isolated functions or short snippets.

For each query program, we include multiple semantically equivalent solutions as positive corpus items, along with both hard negatives and random negatives. Hard negatives are selected based on high Jaccard similarity to the query, making them difficult to distinguish without structural understanding. This design encourages models to rely on semantic and structural cues rather than surface-level similarity.

We adopt NDCG@1 as the primary evaluation metric for LongCCD. Unlike conventional retrieval settings where multiple acceptable results may exist, long-code retrieval often requires identifying the single most semantically aligned program. Moreover, different embedding models exhibit diverse semantic behaviors, making top-ranked accuracy a more reliable indicator than deeper-ranked metrics. NDCG@5 is additionally reported for completeness. Table 2 summarizes the statistics of the LongCCD (C++) dataset used in our experiments.

Design Objective. A key design goal of LongCCD is to ensure that retrieval evaluation remains meaningful across diverse models with different semantic biases. To this end, the dataset is constructed with a relatively small number of queries and a substantially larger corpus. This design increases the diversity of plausible candidates, giving each model the opportunity to retrieve the result that best aligns with its own representation characteristics, rather than being constrained by a narrow candidate set.

Evaluation Metric. Under this setting, NDCG@1 is adopted as the primary metric. When the corpus contains many semantically related candidates, evaluating only deeper-ranked results may obscure whether a model truly identifies its most appropriate match. By focusing on the top-ranked result, NDCG@1 provides a clearer and more discriminative signal of a model's ability to select the best candidate among many alternatives. NDCG@5 is additionally reported to reflect broader retrieval behavior.

A.2 Experimental Setup

All experiments are conducted on a high-performance server equipped with **Intel Xeon Gold 6348R+ CPUs**, **4 × NVIDIA RTX A6000 GPUs**, and **1024 GB RAM**. This configuration is used for all embedding models that support or benefit from GPU acceleration.

Table 2: Statistics of the LongCCD (C++) dataset used in our experiments.

Statistic	Value
Total Queries	30
Total Corpus	390
Corpus per Query	10
Hard Negatives	60
Random Negatives	30

For embedding models that do not utilize GPUs, including *text-embedding-3-large*, experiments are conducted on a separate server equipped with **Intel Xeon Gold 6240R+ CPUs** and **4 × NVIDIA A10 GPUs** and **1024 GB RAM**. Since these models perform inference exclusively on CPUs, GPU availability does not affect their execution.

To ensure fairness and reproducibility, all experiments using the same embedding model are executed under identical hardware configurations. Performance comparisons are therefore conducted only across results obtained within the same execution environment.

A.3 Ablation Study

Table 4 reports an ablation study on LongCCD, analyzing the impact of parallelization, caching, and Hungarian matching.

The results in Table 4 demonstrate that applying the Hungarian algorithm enables Tree Edit Distance (TED) to be computed at the module level rather than on entire monolithic programs. By decomposing code into reusable modules (e.g., functions or classes), the overall computation is split into multiple smaller subproblems, which significantly reduces runtime.

Compared to the monolithic TSED baseline, Hungarian-based decomposition reduces execution time from 30,744 seconds to approximately 650 seconds, corresponding to a reduction of about **97.9%**. Furthermore, module-level decomposition increases reusability across comparisons, especially when combined with caching.

Data Leakage Considerations. While we use the CodeContest+ dataset, it is essentially identical to the original CodeContest dataset, which means that several embedding models may have been highly exposed to substantial portions of the

Table 3: Comparison of the average number of relevant code snippets per query and the average code length (in characters) between queries and corpora across code-to-code retrieval datasets. LongCCD features significantly longer code sequences than other datasets.

Dataset	Language	Avg. Query Len.	Avg. Corpus Len.
CodeTransOcean-Contest (Yan et al., 2023)	Python, C++	367.80	522.78
CodeTransOcean-DL (Yan et al., 2023)	Python	1867.62	1479.07
CodeSearchNet-ccr (Husain et al., 2020)	Python	367.80	522.78
LongCCD	C++	**4698.70**	**3980.15**

Table 4: Ablation study on LongCCD analyzing the impact of module-level decomposition and Hungarian matching on TSED efficiency. Hungarian-based aggregation enables module-wise TED computation, reducing execution time by approximately 98% compared to the monolithic baseline, while improving retrieval accuracy and reusability. Setting 'Parallelize' includes debugging original TSED's node naming errors.

Setting	NDCG@1	NDCG@5	Time (s)
TSED (Baseline)	34.89	29.69	30744.15
TSED + Parallelization	**40.00**	32.40	1541.96
TSED + Parallelization + Hungarian Matching	**40.00**	**35.11**	**645.81**

data during pre-training. As a result, both datasets are susceptible to unusually high retrieval scores that are unlikely to reflect genuine generalization. To mitigate this potential data leakage issue, we exclude models that exhibit near-saturated performance on LongCCD or CodeContest+, as such results are difficult to interpret as meaningful retrieval ability. This filtering is applied conservatively and only affects evaluations specific to these datasets; all models are still reported on CodeTransOcean-DL. Our primary conclusions regarding the effectiveness of Hungarian-based aggregation remain consistent across all datasets.

B Discussion

B.1 Why Hungarian Aggregation Works

The observed performance gains can be attributed to the ability of Hungarian aggregation to align semantically corresponding code modules rather than entire code snippets. By decomposing code into modular units and performing bipartite matching, the proposed approach effectively relaxes strict one-to-one comparisons imposed by monolithic embeddings. This is particularly important for long code, where irrelevant or auxiliary components may otherwise dominate global representations.

B.2 Model-Agnostic Structural Benefits

An important observation is that Hungarian aggregation improves performance across a wide range of embedding models, from lightweight open-source encoders to large proprietary embeddings. This indicates that the proposed method acts as a structural enhancement that is largely orthogonal to embedding quality. Even strong embedding models benefit from module-level aggregation, suggesting that structural alignment remains valuable despite increased model capacity.

B.3 Data Leakage Considerations

Several recent large-scale embedding models achieve near-saturated performance on LongCCD when used in a monolithic setting. Given that LongCCD is constructed from legacy code contest problems released more than three years ago, such results may reflect potential overlap with pre-training data rather than genuine retrieval capability. To ensure meaningful comparison, we conservatively exclude models whose performance is difficult to interpret due to possible data leakage. Importantly, the relative effectiveness of Hungarian-based aggregation remains consistent across retained models.

B.4 Efficiency Trade-offs

Hungarian aggregation introduces additional computational overhead due to pairwise module matching. However, the runtime increase is moderate for most open-source models and is often justified by substantial improvements in retrieval accuracy.

Table 5: Mapping strategies for AST decomposition across languages. We retain major structural components (e.g., functions, classes) as distinct module groups, while merging granular variants (e.g., specific import types or preprocessor directives) into unified groups to facilitate robust alignment.

Language	Tree-sitter Node Type	Mapped Module Group
C++	`function_definition`	`function_definition`
	`class_specifier`	`class_specifier`
	`struct_specifier`	`struct_specifier`
	`enum_specifier`	`enum_specifier`
	`preproc_*`	`preproc_directive`
Python	`function_definition`	`function_definition`
	`class_definition`	`class_definition`
	`import_statement`	`import_statement`
	`import_from_statement`	
	`future_import_statement`	

Moreover, similarity-based Hungarian aggregation typically incurs comparable runtime to random matching, indicating that improved alignment quality does not significantly increase computational cost.