Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1140

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

A Multi-Indicator AST-Based Approach for Code Similarity Detection in Programming Education

Rizky Pamuji[1*]✉, R Ati Sukmawati[1], Novan Alkaf Bahraini Saputra[1], Rita Zubaidah[1], Muhammad Nur Fadhillah[1]
[1]Computer Education, Universitas Lambung Mangkurat, Banjarmasin, Indonesia
**Corresponding Author: rizky.pmuji@ulm.ac.id*

Article Information

Article history:

No. 1140
Rec. April 20, 2026
Rev. June 17, 2026
Acc. June 25, 2026
Pub. June 26, 2026
Page. 1648 – 1663

Keywords:

- *Programming*
- *Code*
- *Similarity*
- *Abstract Syntax Tree*
- *Detection*

ABSTRACT

Evaluation of programming assignments in higher education often faces challenges in ensuring code originality, particularly when students apply cosmetic modifications such as renaming identifiers or altering formatting and comments. This study proposes a multi-indicator code similarity detection system based on Abstract Syntax Tree (AST) analysis to support more objective assessment of programming tasks. The proposed approach analyzes structural, logical, and stylistic aspects of Python programs, combining AST-based analysis with string similarity techniques for comment evaluation. A configurable weighting mechanism is introduced to allow flexible adjustment of similarity assessment according to different evaluation objectives. Experimental results on student programming assignments demonstrate that the system effectively distinguishes varying levels of code similarity and provides consistent similarity measurements. In addition, block-level analysis enables more fine-grained identification of similar code segments. The findings indicate that the proposed method is robust against cosmetic code modifications and can support more systematic and transparent programming assessment in educational contexts.

How to Cite:

Pramuji, R., & et al. (2026). A Multi-Indicator AST-Based Approach for Code Similarity Detection in Programming Education. Jurnal Teknologi Informasi Dan Pendidikan, 19(2), 1648-1663. https://doi.org/10.24036/jtip.v19i2.1140

1. INTRODUCTION

The evaluation of programming assignments in higher education plays a crucial role in ensuring that students achieve the expected learning outcomes. However, this process often faces several challenges, including the time-consuming nature of manual evaluation, inconsistencies in assessment, and the limited availability of supporting tools for detecting code originality [1], [2].

In practice, students may copy code from various sources or share solutions with their peers. They may then apply minor modifications, such as renaming identifiers (e.g., variables, methods, or classes), adjusting formatting, or altering comments. Although these changes make programs appear different at the surface level, the underlying structure and logic often remain largely unchanged [2]. Such practices may negatively impact the learning process by reducing students' engagement in developing problem-solving skills and increasing the risk of plagiarism. Therefore, reliable and automated methods for detecting source code similarity are required to support fair and objective assessment [1].

Existing code similarity detection approaches, particularly those based on textual or token-level representations, rely heavily on surface-level features. As a result, they are highly sensitive to cosmetic modifications that do not alter the underlying program logic. Consequently, these methods often fail to accurately capture structural and behavioral similarities between programs [2], [3].

As an alternative, Abstract Syntax Tree (AST) provides a structural representation of source code based on the grammar of a programming language [4]. In this representation, programs are modeled as hierarchical tree structures, where nodes correspond to syntactic constructs such as loops, conditional statements, function definitions, and assignment operations. By abstracting away superficial elements such as formatting and minor syntactic variations, AST enables similarity analysis that focuses on the structural and logical organization of programs rather than their textual appearance [5].

In this study, code similarity detection is performed using a many-to-many comparison approach, where multiple student programs are compared pairwise to identify code pairs with high similarity [6]. Each source code file is parsed into an AST, and a set of structural features is extracted to represent key aspects of the program, including control structures, execution order, hierarchical depth, logical modifications, and other syntactic patterns. These features are then used to compute similarity scores that reflect structural and logical similarities, even in the presence of cosmetic modifications [3], [5].

While AST-based analysis effectively captures structural and logical similarities [6], it does not directly represent the semantic content of comments embedded in the source code. Therefore, this study complements AST-based similarity detection with a string-based approach for analyzing comments. Comment texts are extracted and compared using string-matching techniques to measure their similarity, allowing the system to distinguish similarities arising from program structure and logic from those originating in documentation or explanatory text [7].

Nevertheless, existing AST-based approaches often rely on limited indicators and fixed similarity models, which restrict their adaptability to diverse programming evaluation contexts [8]. Such limitations reduce their effectiveness in capturing the multi-dimensional nature of code similarity and limit their applicability in educational settings where assessment criteria may vary [3], [5].

To address these limitations, this study proposes a configurable multi-indicator code similarity detection framework based on AST analysis. The proposed method decomposes code similarity into multiple dimensions, including structural, behavioral, and stylistic aspects, through the integration of seven similarity indicators. A configurable weighted aggregation scheme is employed to control the contribution of each indicator, enabling the system to emphasize aspects that are more relevant to specific evaluation objectives [9], [10].

This flexibility provides practical benefits in educational contexts, as instructors can adjust indicator weights according to their assessment priorities. For example, greater emphasis can be placed on structural and logical similarity when investigating potential plagiarism, while stylistic aspects such as formatting or variable naming can be assigned lower importance.

In addition, comment-level similarity is incorporated using string matching, while similarity analysis is performed at both file-level and code-block level. These capabilities enable a more comprehensive and fine-grained evaluation of code similarity. The proposed system provides similarity evidence to assist plagiarism investigation, while final judgments remain subject to human evaluation.

Therefore, the main contribution of this study is the development of a configurable AST-based similarity framework that combines seven similarity indicators through weighted aggregation, enabling flexible code similarity evaluation across different educational assessment objectives.

2. RESEARCH METHOD

This study employs a structural code similarity detection approach based on the Abstract Syntax Tree (AST), complemented by a string-based similarity analysis for source code comments [11], [12]. The proposed method is designed to compare multiple programming assignments in a many-to-many comparison setting, enabling the systematic identification of similarity patterns among student submissions.

As shown in Figure 1, the similarity detection process consists of several main stages: source code preprocessing, AST construction, feature extraction, similarity computation for multiple indicators, and weighted aggregation of similarity scores.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

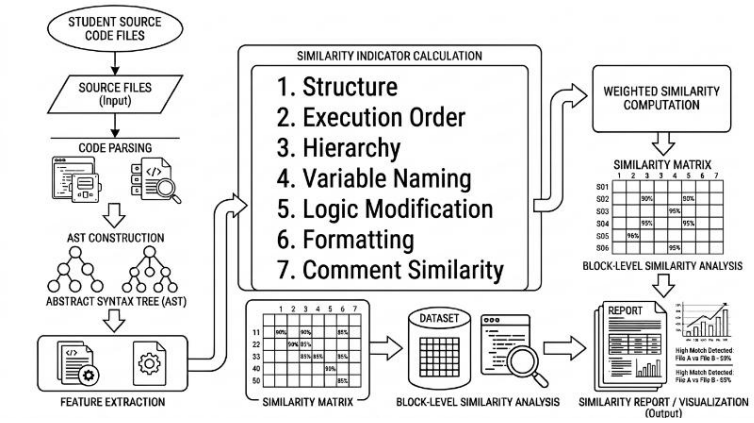


Figure 1. System Architecture / Workflow

2.1. Dataset and Comparison Scheme

The dataset consists of 30 Python source code files collected from student programming assignments. The programs represent three sorting algorithms, namely Bubble Sort, Selection Sort, and Merge Sort, with ten submissions for each algorithm. Similarity analysis is performed separately for each algorithm dataset to ensure that comparisons are conducted among programs implementing the same programming task. Since the source codes were developed independently by students, variations naturally occur in program structure, variable naming, formatting style, commenting practices, and implementation strategies. This diversity provides a realistic evaluation environment for assessing the robustness of the proposed similarity detection framework. All submissions are written in Python to ensure consistency in syntax and facilitate uniform AST analysis.

To evaluate code similarity, a many-to-many comparison scheme is applied, in which each source code file is compared against every other file within the same algorithm dataset. Given a total of N=10 source code files per dataset, this approach produces $\frac{N(N-1)}{2} = 45$ unique code pairs for similarity analysis. Since three algorithm datasets are included in this study, the complete evaluation involves a total of 135 pairwise code comparisons.

This exhaustive comparison strategy enables the identification of different levels of similarity, ranging from nearly identical implementations to independently developed solutions with substantial structural variations. It is particularly effective for detecting potential plagiarism cases, where students may introduce superficial changes such as variable renaming, formatting variations, or minor logic modifications while preserving the core algorithmic structure [13].

2.2. Abstract Syntax Tree Construction

Each source code file is first parsed into an Abstract Syntax Tree (AST) using the Python AST module. The AST provides a hierarchical representation of the source code,

where nodes correspond to syntactic constructs such as function definitions, loops, conditional statements, and assignments [14].

This tree-based representation abstracts away surface-level variations such as formatting and indentation, while preserving the essential structural and logical components of the program [14]. Consequently, unlike raw source code comparison, the AST enables a more robust analysis by focusing on the underlying program structure rather than its textual representation [15].

Although certain lexical elements such as variable names are not central to the AST structure, they are still extracted and analyzed separately as part of specific similarity indicators. The constructed AST serves as the primary foundation for extracting features related to structure, execution order, and hierarchical relationships, which are further utilized in the proposed multi-indicator similarity measurement.

2.3. Feature Extraction and Indicator-Based Similarity

After the AST is constructed, structural features are extracted from each tree to represent various aspects of the program, including control-flow structures, hierarchical depth, execution order, and logical operations. Each feature is quantified into a numerical representation to enable similarity computation. The similarity between corresponding features extracted from two programs is computed using a normalized distance-based similarity measure [16]. For each indicator i, the similarity score between program A and program B is defined as:

$$S_i(A,B) = \begin{cases} 1, & \text{if } F_i(A)=0, F_i(B)=0 \\ 1 - \frac{|F_i(A)-F_i(B)|}{\max(F_i(A),F_i(B))}, & otherwise \end{cases} \quad \textbf{(1)}$$

Where:

- $S_i(A,B)$= denotes the similarity score for indicator i between program A and program B;
- $F_i(A), F_i(B)$= represent the feature values of indicator i for programs A and B, respectively.

Each feature F_i is represented as a numerical value derived from the AST, such as node counts, depth measurements, or frequency of specific constructs. The resulting similarity score ranges from 0 to 1, where higher values indicate a greater degree of similarity.

Similarity computation is performed by comparing these feature representations across different ASTs. This AST-based analysis primarily captures similarities related to program structure, execution flow, and logical organization, while remaining robust to superficial changes such as formatting variations and minor code modifications.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

In addition to file-level similarity, the proposed system also performs block-level similarity analysis. Each AST is decomposed into meaningful structural components, such as function bodies, loop constructs, conditional branches, and assignment blocks. Similarity is computed not only between complete programs but also between corresponding code blocks across different source files. This approach enables the detection of partial similarities, where only specific segments of code exhibit high similarity, even when the overall programs differ significantly.

The final similarity score is obtained by aggregating all indicator scores using a weighted scheme, as described in the next section.

2.4. Weighted Similarity Aggregation

This section describes the aggregation mechanism used to combine similarity scores obtained from multiple indicators. Each indicator contributes to the final similarity score with a different level of importance, which is controlled through a weighting scheme.

The weighted similarity approach allows the system to emphasize indicators that are more representative of the underlying program logic, while reducing the influence of indicators that can be easily modified without changing program behavior. This design improves the robustness of the similarity detection process against superficial code transformations [17].

The weighting configuration used in this study is based on educational assessment considerations and expert judgment. Indicators that reflect fundamental program characteristics, such as structure, execution order, and hierarchy, are assigned higher weights because they are less susceptible to cosmetic modifications. In contrast, indicators such as variable naming, formatting, and comments receive lower weights because they can be altered without significantly affecting program functionality.

The final similarity score is computed by aggregating all indicator-based similarity values using a weighted model, as formally defined in the subsequent section.

2.5. Similarity Indicators Definition

This study employs seven similarity indicators to capture different dimensions of source code similarity, including structural, behavioral, and stylistic aspects. Each indicator is designed to measure a specific characteristic of the program, enabling a comprehensive similarity evaluation. The indicators are defined as follows:

1) Structure Similarity: Measures the similarity of program structures based on the presence and frequency of syntactic constructs in the AST, such as loops, conditional statements, and assignments.
2) Execution Order Similarity: Evaluates the similarity of execution flow by analyzing the order of nodes obtained from AST traversal (e.g., depth-first traversal).

3) Hierarchy Similarity: Captures the similarity of hierarchical organization by comparing the depth and nesting levels of AST nodes.
4) Variable Naming Similarity: Measures similarity based on identifier names used in the program, typically using string-based comparison techniques.
5) Comment Similarity: Evaluates the similarity of code comments using string matching, reflecting similarities in documentation and explanatory text.
6) Formatting Similarity: Measures similarity based on code formatting patterns, such as indentation, spacing, and line structure.
7) Logical Modification Similarity: Captures similarities in logical operations and expressions, including arithmetic and conditional expressions, even when minor modifications are applied

2.6. Comment Similarity Using String Matching

While AST-based analysis effectively captures structural and logical similarities, it does not directly represent the semantic content of comments embedded within the source code. Therefore, this study incorporates a string-based similarity approach to analyze code comments. We determined that Comments is not part of the source code but it affect similarity score if calculated it in line with the code.

Comments are extracted separately from each source code file and compared using string-matching techniques to compute comment similarity scores. This process enables the detection of similarities in documentation and explanatory text, which may indicate shared understanding, collaboration, or the reuse of descriptive content among students.

The similarity between comment texts is computed using the sequence matching algorithm implemented through the *SequenceMatcher* class provided by Python's *difflib* library. The *SequenceMatcher* class evaluates similarity based on the proportion of matching character sequences between two text strings.

The comment similarity score is defined as follows:

$$S_{comment(A,B)} = \frac{2 \times M}{|A| + |B|} \quad \textbf{(2)}$$

- $S_{comment}(A, B)$ = represents the similarity between comment texts $A$and B
- M = represents the number of matching characters identified through sequence matching
- $|\ A\ |$and $|\ B\ |$ = denote the length of the compared comment strings

This formulation is consistent with the similarity ratio produced by the *SequenceMatcher* algorithm. The resulting similarity value ranges between 0 and 1, where a value closer to 1 indicates higher similarity between comment contents. This approach is suitable for short textual content such as source code comments, where exact tokenization may not always produce reliable similarity measurements

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

2.7. Combined Similarity Evaluation

The final similarity score for each pair of source code files is derived by combining AST-based similarity indicators with comment similarity results. AST-based indicators represent similarities in program structure, logic, execution behavior, and cosmetic transformations, while string-based similarity is applied exclusively to comments. This separation of similarity domains allows flexible weighting and interpretation of results depending on the evaluation objectives. The system outputs similarity scores along with detailed information about code pairs that exceed a predefined similarity threshold, thereby supporting fine-grained analysis at both the file and code-block levels.

The final similarity score for each pair of source code files is computed by combining multiple similarity indicators derived from Abstract Syntax Tree (AST) analysis and comment similarity analysis. In this study, seven indicators are used to represent different dimensions of code similarity, namely (1) Structure similarity, (2) Execution order similarity, (3) Hierarchy similarity, (4) Variable naming similarity, (5) Comment similarity, (6) Formatting similarity, and (7) Logical modification similarity.

These indicators capture structural, behavioral, and stylistic characteristics of source code. AST-based indicators primarily measure structural and logical similarities between programs, while comment similarity is computed using a string-matching approach. The overall similarity between two programs A and B is computed using a weighted multi-indicator model as expressed in Equation (3).

$$S_{total}\ (A,B) = \sum_{i=1}^{n} = W_i \cdot S_i(A,B) \quad \textbf{(3)}$$

- S_{total} = Total similarity between program A *and* B
- $S_i(A,B)$ = Represent the similarity score of indicators i
- W_i = Denotes the weight assigned to indicator i
- n = The total number of indicators

In this study, $n = 7$ indicators, the weighting values satisfy the following constraint with the condition in Equation (4).

$$\sum_{i=1}^{n} = W_i\ = 1 \quad \textbf{(4)}$$

For the experimental scenario presented in this study, the indicator weights are defined as shown in Equation (5), where each similarity component corresponds to one of the defined indicators.

$$S_{total=}\ w_1 S_{structure} + w_2 S_{execution} + w_3 S_{hierarchy} + w_4 S_{variable} + w_5 S_{comment} + w_6 S_{format} + w_7 S_{logic} \quad \textbf{(5)}$$

Following the computation of the weighted similarity score, a similarity threshold is applied to identify potentially similar source code pairs. In this study, a threshold value of 0.65 is adopted. The threshold was determined based on pedagogical considerations and the experience of programming instructors involved in teaching introductory programming courses. Since students are assigned the same programming tasks and are exposed to similar algorithmic concepts, a certain degree of similarity is naturally expected among independent submissions. Based on instructional experience, similarity scores below 0.65 generally reflect common solution patterns resulting from shared learning materials and problem-solving approaches. In contrast, similarity scores equal to or greater than 0.65 indicate stronger agreement across multiple structural, behavioral, and stylistic indicators, suggesting a substantially higher degree of code similarity. Therefore, the threshold serves as a practical decision boundary for identifying potentially similar source code pairs within an educational assessment context. The threshold is not intended to represent a universal value and may be adjusted according to the characteristics of the dataset and the assessment objectives defined by the instructor.

2.8. Similarity Detection Algorithm

The overall process of similarity detection is summarized in Algorithm 1. The algorithm processes a set of source code files, constructs their Abstract Syntax Trees (AST), extracts structural features, and computes similarity scores based on multiple indicators.

```
Input: Set of source code files F
Output: Similarity matrix M
  1. for each file f in F do
  2. AST_f ← ParseToAST(f)
  3. Features_f ← ExtractStructuralFeatures(AST_f)
  4. Comments_f ← ExtractComments(f)
  5. end for
  6. for each pair (fi, fj) in F do
  7. S_structure ← CompareStructure(Features_fi, Features_fj)
  8. S_execution ← CompareExecutionOrder(AST_fi, AST_fj)
  9. S_hierarchy ← CompareHierarchy(AST_fi, AST_fj)
  10. S_variable ← CompareVariableNames(fi, fj)
  11. S_logic ← CompareLogicalOperations(AST_fi, AST_fj)
  12. S_format ← CompareFormatting(fi, fj)
  13. S_comment ← StringSimilarity(Comments_fi, Comments_fj)
  14. S_total ← WeightedSum(S_i, w_i)
  15. Store S_total in matrix M
  16. end for
  17. return M
```

Figure 2. Pseudocode of the proposed similarity detection system

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

3. RESULTS AND DISCUSSION

3.1. Dataset Description

The dataset used in this study consists of 30 Python source code files collected from programming assignments submitted by undergraduate students enrolled in an introductory programming course. The programs represent three sorting algorithms, namely Bubble Sort, Selection Sort, and Merge Sort, with ten independently developed submissions for each algorithm.

Each student was instructed to implement the assigned algorithm without relying on built-in sorting functions. Although the programming task was identical within each algorithm group, variations naturally emerged in code structure, variable naming, logical organization, formatting style, and comment content. These variations reflect realistic conditions in programming education, where students may arrive at similar solutions while employing different implementation strategies.

The selected algorithms share several common AST characteristics, including loop structures, nested loops, conditional statements, assignment operations, element-swapping mechanisms, and code comments. At the same time, they exhibit differences in execution flow and program organization, making them suitable for evaluating the effectiveness of the proposed multi-indicator AST-based similarity detection framework.

This dataset provides a realistic evaluation environment for code similarity analysis, where similarity may arise from shared algorithmic understanding, common instructional guidance, or potential code reuse. Consequently, it is appropriate for assessing the ability of the proposed framework to distinguish structural, behavioral, stylistic, and comment-related similarities among student-developed programs.

3.2. Experiment Scenario

The experiment was conducted by comparing all pairs of student source code files within a single dataset folder using a many-to-many comparison scheme. Each pair of code files was analyzed using the AST-based approach with seven main indicators and their predefined weights, as shown in Table 1:

Table 1. Code Comparison Indicators

Indicator	Weight
Structure	0.25
Execution Order	0.15
Hierarchy	0.10
Variable Naming	0.10
Comments	0.15
Formatting	0.15
Logical Modification	0.10

Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1140

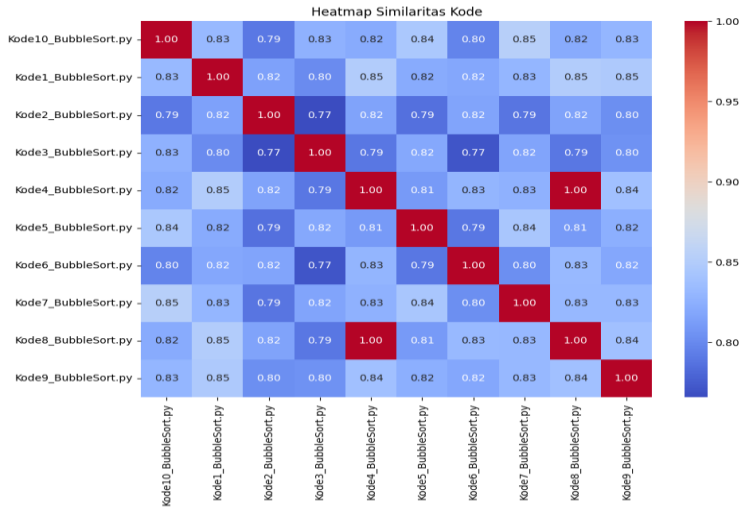
The total weight of all indicators equals 1.00, indicating that each indicator contributes proportionally to the overall similarity calculation. In this scenario, the weighting scheme is not focused on a single aspect but is designed to represent overall code similarity, encompassing structural, logical, and stylistic aspects of the source code.

A similarity threshold value of 0.65 (out of a maximum score of 1.00) was defined to identify code pairs exhibiting significant similarity. Code pairs with similarity scores exceeding this threshold were subsequently analyzed in greater detail at the code-block level.

3.3. System Execution Results

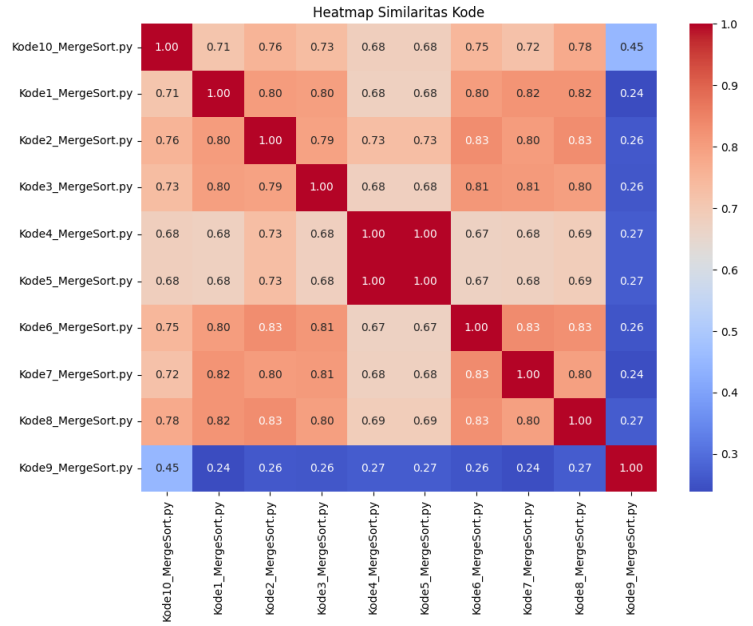
The system execution results in this study are not intended to evaluate the correctness of algorithm implementations, but rather to validate the system's capability to detect code similarity based on program structure, logic, comments, and coding style. The system produces several outputs that represent similarity patterns among student-developed source code files.

One of the primary outputs is a heatmap visualization that illustrates the overall similarity levels among source code files within each algorithm dataset. The results show that every source code file compared with itself produces a similarity score of 1.00, indicating that the normalization and weighting mechanisms operate consistently. In addition, variations in indicator weights and similarity threshold values influence the sensitivity of similarity detection according to the selected configuration.

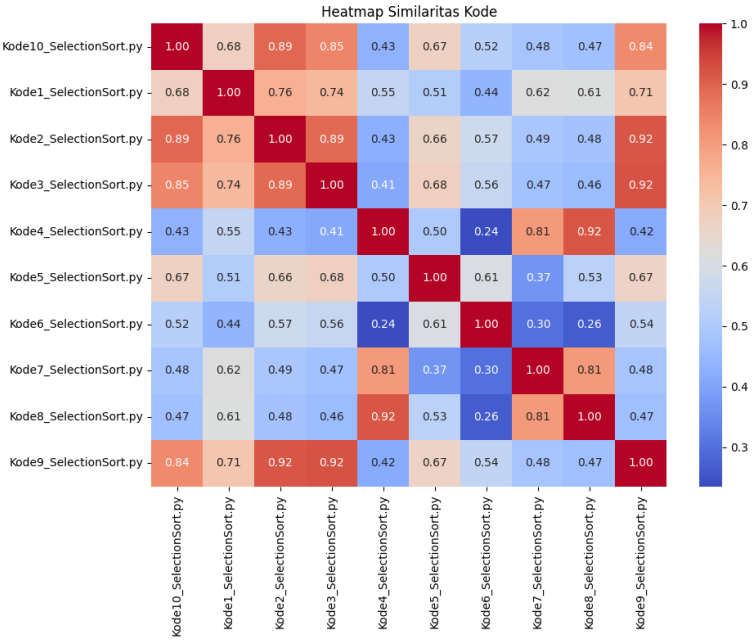


(1)

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id



(2)



(3)

Figure 3. Similarity Result Heatmap Visualization of Code Case of Bubble Sort (1), Merge Sort (2) and Selection Sort (3)

The heatmap visualization also demonstrates that the system can distinguish between identical, highly similar, and moderately similar implementations. This finding

indicates that the proposed multi-indicator weighting scheme generates similarity scores that are discriminative and informative rather than purely binary.

Another output is a list of code blocks whose similarity values exceed the defined threshold of 0.65, presented in both text and spreadsheet formats. This output contains information regarding the compared file pairs, code block types, similarity scores, and representative code excerpts. Consequently, similarity analysis can be performed at a more granular level, enabling the identification of partial similarities that may not always be reflected in overall file-level similarity scores.

File A	File B	Jenis	Similarity	Kode A	Kode B
Kode10_MergeSort.py	Kode10_MergeSort.py	For	1	for i in range(n): indeks_terkeci	for i in range(n): indeks_terkecil = i for j in range(i + 1, n): ...
Kode10_MergeSort.py	Kode10_MergeSort.py	For	1	for data in siswa: print(f"nama:	for data in siswa: print(f"nama: {data['nama']}, nilai: {data['nilai']}")...
Kode10_MergeSort.py	Kode10_MergeSort.py	For	1	for data in siswa: print(f"nama:	for mhs in mahasiswa: print(f"nama: {mhs['nama']}, nilai: {mhs['nilai']}"
Kode10_MergeSort.py	Kode10_MergeSort.py	FunctionDef	1	def merge_sort(data): if len(da	def merge_sort(data): if len(data) > 1: mid = len(data) // 2 ...
Kode10_MergeSort.py	Kode10_MergeSort.py	FunctionDef	0,995774648	def merge_sort(data): if len(da	def merge_sort(data): if len(data) > 1: mid = len(data) // 2 ...
Kode10_MergeSort.py	Kode10_MergeSort.py	FunctionDef	0,995774648	def merge_sort(data): if len(da	def merge_sort(data): if len(data) > 1: mid = len(data) // 2 ...
Kode10_MergeSort.py	Kode10_MergeSort.py	FunctionDef	1	def merge_sort(data): if len(da	def merge_sort(data): if len(data) > 1: mid = len(data) // 2 ...
Kode10_MergeSort.py	Kode10_MergeSort.py	For	1	for mhs in mahasiswa: print(f"n	for data in siswa: print(f"nama: {data['nama']}, nilai: {data['nilai']}")...
Kode10_MergeSort.py	Kode10_MergeSort.py	For	1	for mhs in mahasiswa: print(f"n	for mhs in mahasiswa: print(f"nama: {mhs['nama']}, nilai: {mhs['nilai']}"
Kode10_MergeSort.py	Kode1_MergeSort.py	For	0,959615385	for data in siswa: print(f"nama:	for t in siswa_selectionsort: print (f"| {t[0]} \t| {t[1]} \t|")...
Kode10_MergeSort.py	Kode1_MergeSort.py	For	0,980594406	for data in siswa: print(f"nama:	for b in mahasiswa: print(f"| {b['nama']} \t| {b['nilai']} \t|")...
Kode10_MergeSort.py	Kode1_MergeSort.py	For	0,959615385	for data in siswa: print(f"nama:	for a in mahasiswa: print (f"| {a['nama']} \t| {a['nilai']} \t|")...
Kode10_MergeSort.py	Kode1_MergeSort.py	FunctionDef	0,656443872	def merge_sort(data): if len(da	def selectionsort_siswa (siswa): n= len(siswa) for i in range (n-1): ...

Figure 4. Excerpt from the results of identifying similar code blocks of Bubble Sort

Experimental evaluation was conducted using three sorting algorithm datasets, namely Bubble Sort, Selection Sort, and Merge Sort. Across all datasets, the proposed framework was able to identify varying levels of similarity among student implementations despite differences in variable naming, formatting style, commenting practices, and program organization. These results indicate that the multi-indicator AST-based approach is capable of capturing multiple dimensions of code similarity and can support educational code evaluation in programming courses.

3.4. Discussions

Based on the experimental results, the system consistently produces a maximum similarity score of 1.00 when a source code file is compared with itself, indicating that the normalization and weighting mechanisms operate consistently and reliably. In addition, variations in indicator weights influence the sensitivity of the system toward different aspects of code similarity, demonstrating the effectiveness of the proposed weighted multi-indicator approach [18], [19].

When greater weights are assigned to structural and logical indicators, the system becomes more sensitive to similarities in program organization and algorithmic flow. Conversely, increasing the weights of comment and formatting indicators enhances sensitivity to similarities in documentation and coding style. These findings indicate that the weighting mechanism provides flexibility in adapting the similarity analysis to different evaluation objectives.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

Code-block-level analysis provides more detailed information than overall file-level similarity scores. By identifying highly similar code blocks, the system can highlight specific program segments that share structural and logical characteristics, even when the overall programs are not identical [20]. This capability is particularly beneficial in educational settings, as it enables instructors to investigate similarities in student solutions more effectively and obtain insights beyond a single similarity score.

The experimental evaluation using student-developed Python programs implementing Bubble Sort, Selection Sort, and Merge Sort demonstrates that the proposed framework can be applied across multiple programming assignments while maintaining consistent similarity measurement. The configurable weighting mechanism further allows instructors to emphasize different dimensions of similarity according to specific learning objectives, such as algorithmic understanding, program structure, or coding style.

Although the proposed framework provides quantitative similarity scores, standard classification-based evaluation metrics such as precision, recall, and F1-score were not included in the current study. Future work will involve expert-labeled datasets to enable systematic quantitative validation and benchmarking against established plagiarism detection tools. Furthermore, the experimental evaluation was limited to three sorting algorithm datasets. Additional validation using larger datasets, more complex programming tasks, and diverse programming paradigms is required to further assess the scalability and generalizability of the proposed framework.

The primary contribution of this work lies in the development of a configurable multi-indicator AST-based similarity assessment framework for educational code evaluation. Unlike approaches that rely solely on overall similarity measurements, the proposed framework enables similarity analysis across multiple dimensions, including structural, behavioral, stylistic, and comment-related aspects. This flexibility allows the framework to support a variety of educational assessment scenarios and provides a foundation for future comparative studies with established code similarity and plagiarism detection tools.

4. CONCLUSION

This study proposed and implemented an AST-based code similarity detection framework using a multi-indicator weighting approach. The framework evaluates code similarity across seven indicators, including program structure, execution order, hierarchy, variable naming, logical modification, formatting, and comments. By combining structural analysis with comment similarity, the system provides a more comprehensive assessment of source code similarity than purely text-based approaches.

Experimental results on student programming assignments involving Bubble Sort, Selection Sort, and Merge Sort demonstrate that the proposed framework produces consistent similarity measurements and is capable of identifying both highly similar

programs and similar code blocks. The configurable weighting mechanism allows the analysis to emphasize different similarity dimensions according to educational evaluation objectives.

The proposed framework can support instructors in detecting code similarity, evaluating programming assignments, and investigating potential plagiarism cases. Future work should focus on larger and more diverse datasets, comparison with established plagiarism detection tools, quantitative validation using standard evaluation metrics, and the integration of semantic analysis for source code comments.

REFERENCES

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *SIGMOD 2003*, ACM, Jun. 2003, pp. 76–85. doi: 10.1145/872757.872770.

[2] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding Plagiarisms among a Set of Programs with JPlag," *Journal of Universal Computer Science*, vol. 8, pp. 1016–1038, Nov. 2002, [Online]. Available: http://www.jplag.de.

[3] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: techniques, applications, and challenges," 2023, doi: 10.1016/j.jss.2023.111796.

[4] G. Lukácsy and P. Szeredi, "Plagiarism detection in source programs using structural similarities," *Acta Cybernetica*, vol. 19, no. 1, pp. 191–216, 2009, doi: 10.14232/actacyb.19.1.2009.13.

[5] G. Lee, J. Kim, M. S. Choi, R. Y. Jang, and R. Lee, "Review of Code Similarity and Plagiarism Detection Research Studies," *Applied Sciences (Switzerland)*, vol. 13, no. 20, Oct. 2023, doi: 10.3390/app132011358.

[6] A. Sheneamer, S. Roy, and J. Kalita, "An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion," *IEEE Access*, vol. 9, pp. 84828–84844, 2021, doi: 10.1109/ACCESS.2021.3079156.

[7] Y. Song, C. Lothritz, D. Tang, T. F. Bissyandé, and J. Klein, "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)*, Jun. 2024. doi: 10.18653/v1/2024.acl-short.3.

[8] A. Sheneamer and J. Kalita, "A Survey of Software Clone Detection Techniques," *Int. J. Comput. Appl.*, vol. 137, no. 10, pp. 975–8887, 2016, doi: 10.5120/ijca2016908896.

[9] Y. Mohammed Khazaal and Y. Hammo, "Survey on Software Code Clone Detection," *TECHNIUM*, vol. 4, no. 3, pp. 28–36, 2022, doi: 10.47577/technium.v4i3.6361.

[10] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.

[11] E. Hosam, M. Hadhoud, A. Atiya, and M. Fayek, "Classification feature sets for source code plagiarism detection in Java," *Journal of Engineering and Applied Science*, vol. 69, no. 1, Dec. 2022, doi: 10.1186/s44147-022-00155-8.

[12] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree," in *Conference: 2020 IEEE 27th International*

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

Conference on Software Analysis, Evolution and Reengineering (SANER), Feb. 2020. doi: doi.org/10.48550/arXiv.2002.08653.

[13] M. A. Umar, "A Study of Software Testing: Categories, Levels, Techniques, and Types," Jun. 29, 2020. doi: 10.36227/techrxiv.12578714.v1.

[14] Y. and S. W. and F. S. and W. C. and Z. D. and J. H. Wu, "Fine-Grained Code Clone Detection by Keywords-Based Connection of Program Dependency Graph," *IEEE Trans. Reliab.*, pp. 1–15, 2025.

[15] Gergely Lucacsy and Peter Szeredi, "Plagiarism Detection in Source Programs Using Structural Similarities," *Acta Cybern*, vol. 19, pp. 191–216, 2009.

[16] H. E. Wahanani, M. H. Prami Swari, and F. A. Akbar, "Case based Reasoning Prediksi Waktu Studi Mahasiswa Menggunakan Metode Euclidean Distance dan Normalisasi Min-Max," *Jurnal Teknologi Informasi dan Ilmu Komputer*, vol. 7, no. 6, p. 1279, Dec. 2020, doi: 10.25126/jtiik.2020763880.

[17] S. Kalhor and M. R. Keyvanpour, "Weighted Content Similarity Feature for Software Architecture Anti-Patterns Prediction," *International Journal of Web Research (IJWR)*, vol. 8, no. 3, pp. 33–43, 2025, [Online]. Available: https://ijwr.usc.ac.ir/article_226898.html

[18] B. Kim, K. Lim, S.-J. Cho, and M. Park, "RomaDroid: A Robust and Efficient Technique for Detecting Android App Clones Using a Tree Structure and Components of Each App's Manifest File," *IEEE Access*, vol. 7, pp. 72182–72196, 2019, doi: 10.1109/ACCESS.2019.2920314.

[19] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009, doi: 10.1016/j.scico.2009.02.007.

[20] Y. Wang, D. Liu, and M. Hou, "Study of Clone Code Detection Method," in *Proceedings of the 3rd International Conference on Computer Engineering, Information Science & Application Technology (ICCIA 2019)*, Paris, France: Atlantis Press, 2019. doi: 10.2991/iccia-19.2019.64.