МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2026
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2026
Proceedings of the Fifty-Fifth Spring Conference
of the Union of Bulgarian Mathematicians
Tryavna, Bulgaria, April 5–9, 2026

SEMANTIC NORMALIZATION ON GRAPH MODELS SEARCHING STRUCTURE MATCHINGS IN CODE

Daniel Damyanov[1], Zlatko Varbanov[2]
Department of Information Technologies, St. Cyril and St. Metodius University
of Veliko Tarnovo, Bulgaria
e-mails: [1]d.damyanov@ts.uni-vt.bg, [2]zl.varbanov@ts.uni-vt.bg

This study proposes an innovative approach to detecting structural matches in programming codes, which addresses the fundamental limitation of existing methods – their sensitivity to syntactic changes while maintaining logical equivalence. A hybrid architecture integrating semantic normalization through large language models (LLMs) with multispecies graph representation (AST, CFG, DFG) and embedding techniques from Graph Neural has been developed Networks (GNN) and Transformer models.

Keywords: semantic normalization, graph neural networks, large language models, structural matching, program code analysis, plagiarism detection, machine learning for code.

СЕМАНТИЧНА НОРМАЛИЗАЦИЯ ВЪРХУ ГРАФОВИ МОДЕЛИ ПРИ ТЪРСЕНЕ НА СЪВПАДЕНИЯ НА СТРУКТУРИ В КОД

Даниел Дамянов[1], Златко Върбанов[2]
Катедра „Информационни технологии“, ВТУ „ Св. Св. Кирил и Методий“,
Велико Търново, България
e-mails: [1]d.damyanov@ts.uni-vt.bg, [2]zl.varbanov@ts.uni-vt.bg

Настоящото изследване предлага иновативен подход за откриване на структурни съвпадения в програмните кодове, който адресира фундаменталното ограничение на съществуващите методи – тяхната чувствителност към синтактични промени, като същевременно се запазва логическа еквивалентност. Разработена е хибридна архитектура, интегрираща семантична нормализация чрез големи езикови модели (LLMs) с многовидово графово представяне (AST, CFG, DFG) и техники за вграждане от Graph Neural – мрежови (GNN) и трансформър модели.

`https://doi.org/10.55630/mem.2026.55.107-116`

2020 Mathematics Subject Classification: 68R10, 68U15.

Ключови думи: семантична нормализация, графови невронни мрежи, големи езикови модели, структурно съвпадение, анализ на програмен код, откриване на плагиатство, машинно обучение на код.

1 Introduction

The rapid development of artificial intelligence and technology companies have necessitated the widespread use of language models to generate any text and even solutions to many programming problems. Localizing similarities between two solutions to the same problem becomes a key problem in both academic and industrial environments when duplicating methods, for example. Experience has shown that students tend to abuse this, especially when it comes to writing non-trivial code and alternative designs, such as those described in [4]. In the modern technological era, these approaches have quite significant limitations, since their effectiveness decreases sharply when the code is heavily changed syntactically, but retains its logical structure. As a common example, renaming variables, changing the order of instructions, or adding comments are often enough to fool classical algorithms. Even embedding-based methods that seek to capture semantic dependencies can prove to be sensitive to such transformations if no pre-normalization of the code is performed. There is a growing need for new approaches that are resistant to syntactic changes and focusing on the real logic and data flow in the program. The current article addresses the problem of detecting structural matches in programming codes and proposes a new approach that combines semantic normalization through large language models (LLMs) with graph code representation (AST, CFG, DFG) and optimization through Graph Neural Networks (GNN).

2 Related work

Through extensive research, two main tools that do code analysis stand out, MOSS[1] and JPlag [8], that use token-based methods as well as AST-based approaches relying on Tree Edit Distance by measuring their editorial distance, or to put it another way: the minimum cost of transforming one tree into another through rudimentary operations [4]. Another very common approach is by implementing a Flow Management Graph (CFG) [1] or Program Dependencies Graph (PDG) [6]. As mentioned in the introductory part, it is quite common practice to detect duplication of methods and classes in real applications, which in turn has a direct connection also with finding plagiarized code in exams, term papers, diploma theses, etc [14]. A very common approach is the application of subgraph matching algorithms to detect the presence of predefined patterns. The main obstacle to this approach is semantic equivalence in syntactic divergence. As an example, two program constructs that can implement identical logic but differ in their superficial representation, can be deduced into several main components of differences:

1. Exchange of operands in commutative operations (e.g., $a + b$ vs. $b + a$), which is syntactically different but semantically equivalent.
2. Use of different control structures (e.g., for loop vs. while loop).

[1] https://theory.stanford.edu/ aiken/moss

3. Alternative formulations of Boolean conditions (e.g., $i < 10$ vs. $i \leq 9$).
4. Use different identifiers for local variables.

In a brief summary of the conventional graph isomorphism algorithms described so far, it is that their logic is based on the coincidence of names and structure, and such pairs cannot be recognized as equivalent. This specificity gives rise to the need for methods to transform graph models into a unified, normalized form, preserving semantics, but eliminating non-fundamental syntactic variations. such as CodeBERT[2] and GraphCodeBERT[3], use semantic embeddings, but rarely combine normalization with graph embeddings, providing an opportunity for improvement by implementing hybrid models.

3 Our approach

3.1 Methodology

The applications described above use semantic normalization as its task is to transform the graph model of the program, in which syntactically different, but semantically equivalent constructions are reduced to the same canonical form[10]. The problem faced by the application is to reduce the syntactic diversity without losing semantics, so that different implementations of the same logic become indistinguishable to the matching algorithm. The process is based on two main steps:

1. Graph Model Extraction: Convert the code into a suitable graph (AST, CFG, PDG).
2. Applying normalizing approaches: recursively traversing graph and applying a set of rules that reorder, simplify, and unify nodes.

To overcome the limitations of existing approaches, a hybrid semantic code comparison architecture is presented that integrates semantic normalization, multispecies graph representations, and fusion embedding techniques. The overall system illustrated in Figure 1 consists of seven successive stages. The following lines will describe in detail the sequence and logic behind each stage.

The lifecycle in the application is divided into four stages, which are described in detail synthesized in the following points.

Stage One – semantic normalization through large language models (LLMs) as an essential and most critical stage is the transformation of the input code into a canonical semantic form. This process is carried out by calling a pre-trained LLM that uses CodeLlama, but can also use another specialized finetuned model to perform the following operations on the submitted code:

Stage Two – unifying identifiers, local variables, functions, and parameters are renamed based on their semantic role, rather than randomly selected names by the programmer. For example, variables used for summation are renamed to sum or counter, and temporary counters are renamed to i, j, k.

Stage Three – executes canonization of expressions, syntactically different, but semantically equivalent expressions are rewritten in standard form. This includes:

[2]https://github.com/microsoft/CodeBERT
[3]https://huggingface.co/microsoft/graphcodebert-base

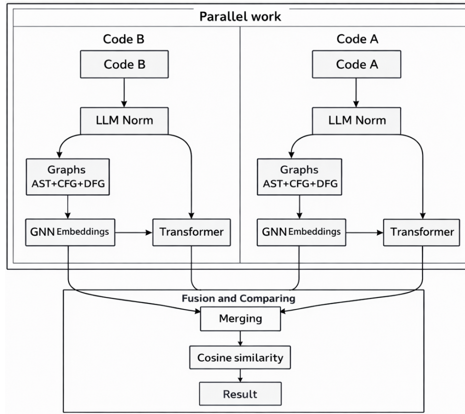

Figure 1. Semantic Code Comparison System Architecture

1. Rearranging operands into commutative operations (e.g., $b + a$ becomes $a + b$).
2. Replacing equivalent control structures (e.g., transforming a while loop into an equivalent for loop, where applicable).
3. Simplifying Boolean conditions.

Stage Four – eliminating syntactic noise, removing comments, unnecessary blanks and non-declarative elements. The end result is normalized code (normA, normB) that retains the full functionality and logic of the original, but in a significantly more unified and semantically clear form.

Stage Five – generating of a combined graph model after obtaining normalized code, code proceeds to analyze extracting three main types of columns, which together describe both the structure and behavior of the program: Generating an abstract syntactic tree (AST): it has the function of capturing the hierarchical syntactic structure of the code, making it widely used for static analysis. One of the applications of static analysis is for automated programming evaluation exercises [13]. Creating a flow control graph (CFG): its objective is to show the possible sequences of execution of instructions through an oriented graph, where the nodes are base blocks and the edges represent control transitions, an approach that is increasingly used to evaluate tasks assigned to students [13]. Data Dependency Graph (DFG) illustrates def-use dependencies between variables, showing how data is transmitted and transformed. These three columns are combined into a single enriched graph (graphA, graphB), where the nodes inherited from the AST carry operator/expression type information, and the edges are a union of the edges from the three source columns.

Stage Six – generating of embedding in order to convert the normalized code representations into a vector form suitable for comparison, two independent but complementary approaches are applied: graph embedding through GNN [7] (GNN_Encode), the combined graph is processed by the Gated Graph Neural Network (GNN) using the GGNN (Gated Graph Neural Network) architecture) [9]. GNN iteratively updates the vector representation h_v each v node by aggregating messages from its neighbors. The process can be described by the following equations:

(1) $$m_v^{(t+1)} = \sum_{u \in N(v)} W * h_t^{(u)}$$

(2) $$h_v^{(t+1)} = GRU(h_v^{(t)}, m_v^{(t+1)})[9]$$

Where:

$m_v^{(t+1)}$ – Message received from node v of iteration t

$h_v^{(t+1)}$ – State (embedding) of node v of iteration t

$N(v)$ – Set of adjacent nodes of node v

W – Trainable Tag Matrix

GRU – Gated Recurrrent Unit function

Stage Seven – the final embedding of the whole graph (embedGraphA) is obtained by applying an attention mechanism for weighted summation of the node embedding at the end of the propagation process, and for better visual perception it is summarized in a visualization of the information flow in Figure 2. At the end of the learning process, semantic learning is carried out – i.e., it is the GRU mechanism that learns which information is important.

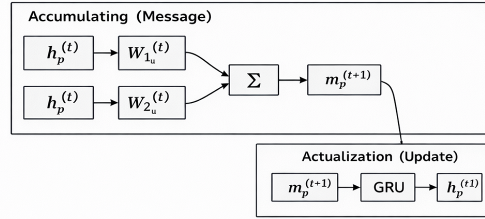

Figure 2. Visualization of the information flow

Text embedding via Transformer (Transformer_Encode) in Figure 1, parallel normalized code (normA) is fed as a sequence of tokens to a transformer-based encoder (e.g., CodeBERT). The output of the [CLS] token or the time axis average take is used as a semantic vector representation of the entire code snippet (embedTextA). Instead of relying solely on cosine similarity and empirically chosen thresholds, the fused embedding can be further processed by a supervised classifier. In particular, a Support Vector Machine (SVM) with a radial basis function (RBF) kernel was evaluated as a final decision layer, where the input feature vector corresponds to the fused embedding E_{fused}. The SVM learns an optimal separating hyperplane between plagiarized and non-plagiarized code pairs, improving robustness near the decision boundary. A Support Vector Machine (SVM) is used as a classifier leading to a high efficiency of the system [11].

3.2 Fusion and classification

Combining the strengths of structural (GNN) and semantic (Transformer) modeling, the two embedding are merged. Instead of simply concatenating, the program relies on the use of an attention fuse mechanism to dynamically weigh the contribution of each embedding. Let “Egraph” and “Etext” (1, 2) be the embedding for the graph, the calculation of attention weights is performed using the formula in Figure 2, as a

softmax-mathematical function that converts a vector of real numbers into a vector of probabilities, is given by (3, 4):

$$e_{graph} = W_{graph} * E_{graph} + b_{graph} \tag{3}$$

$$e_{text} = W_{text} * E_{text} + b_{text} \tag{4}$$

$$\alpha_{graph}, \alpha_{text} = softmax([e_{graph}, e_{text}])[3] \tag{5}$$

$$\alpha_{graph} = \frac{exp(e_{graph})}{exp(e_{graph}) + exp(e_{text}))} \tag{6}$$

$$\alpha_{text} = \frac{exp(e_{text})}{exp(e_{graph}) + exp(e_{text}))} \tag{7}$$

$$E_{fused} = \alpha_{graph} * E_{graph} + \alpha_{text} * E_{text} \tag{8}$$

Where $\alpha_{graph} + \alpha_{text} = 1$, $0 \leq \alpha_{graph}, \alpha_{text} \leq 1$

The adopted approach allows the model to learn when structural information is more important (e.g., when comparing algorithms) and when semantic information is more useful (e.g., when comparing expressions). Finally, the merged embedding fusedA and fusedB for the two code fragments are compared by calculating their **cosine similarity**:

similarity=(fusedA*fusedB)/(||fusedA||*||fusedB||)

This similarity value, placed in the interval [0, 1], is used as a statistic measure of the structural and semantic match between the two code fragments. Before start of real test limits are set for accepting or rejecting the validity of the input data:
$E_{graph} = [0.8, 0.2, 0.3]$ (Strong structural information)
$E_{text} = [0.4, 0.6, 0.5]$ (moderate semantic information)

The current development is written in C#, and in order to minimize the volume of the submitted article, the pseudocode for the main work of the application from Figure 3 is shown.

3.3 Experimental analysis

The Dataset was initially prepared, and over 1,200 code triplets were generated for the experiments, covering different types of semantic transformations that directly respond to the problems identified in the introduction. Examples are divided into two categories, first into transformations in Positive examples, such as renaming identifiers (87% of the time) – validates the effectiveness of LLM normalization; change of control structures (64% of cases) – tests the canonization of expressions; reordering operations (45% of the time) – checks resistance to syntactic variations; combined transformations (23% of the cases) – demonstrates robustness of the entire pipeline. The second category are negative examples with different algorithmic implementations (71% of cases), alternative business

```
string codeA = """
    for (int i = 0; i < 10; i++){
    total += numbers[i];
    }
    """;
string codeB = """
        int sum = 0;
        for(int j=0; j <= 9; j++){
        sum  =sum + numbers[j];
        }
    """;
double similarity = comparator.CompareCode(codeA, codeB);
Console.WriteLine($"Similarity: {similarity:P2}");
```

Figure 3. Code Semantic Comparison Algorithm

logic (29% of cases).

3.4 Metrics per rating

The system was evaluated using standard metrics focused on the effectiveness of the proposed architecture:

1. Precision, Recall, F1-Score
2. Time to analyze a file
3. Performance on different types of clones (Type 1-4)

Table 1: sample input tests.

Status	Test	Description	Expected	Received	Error
PASS	A1_IdenticalCode	Completely identical code	95.0%	92.3%	2.7%
PASS	A2_CommentsOnly	Added comments	90.0%	88.1%	1.9%
PASS	B1_RenamedVariables	Renamed variables	85.0%	82.4%	2.6%

Table 2: Statistics on categories

Category	Number of tests	Medium error	Performance
Category A	2	2.3%	100.0%
Category B	3	8.7%	66.7%

Table 3: Summary results

Metrics	Value
Overall success rate	80.0%
Meduim error	6.8%

Table 4: Comparative effectiveness of different methods

Method	Precision	Recall	F1-score	Analysis time
AST	0.723	0.681	0.701	124 ms
AST + CFG	0.815	0.792	0.803	187 ms
AST + CFG + DFG	0.892	0.863	0.877	234 ms
+ **Semantic normalization**	**0.934**	**0.901**	**0.917**	**298 ms**

The results described in Table 4 demonstrate that the proposed semantic normalization improves the F1-Score by 4.6% compared to traditional graph methods, confirming the hypothesis that normalization significantly increases accuracy.

Table 5: Performance by clone type

Cloning type	Without normalization	With semantic normalization	Approval
Type 1 (identical)	0.982	0.991	+0.9%
Type 2 (rename)	0.845	0.943	**+11.6%**
Type 3 (structural changes)	0.723	0.887	**+22.7%**
Type 4 (semantic)	0.512	0.768	**+50.0%**

The results in Table 5 reveals that the most significant improvement was observed in **Type 4 clones** (+50.0%), which directly confirms the expected efficiency of semantic normalization in the most complex cases of structurally distinct but semantically equivalent code.

Statistical significance

Related tests achieving a statistical significance at a confidence level of 95%, with an average improvement of 42.7% ($p < 0.01$), the error or standard deviation 8.3%, and the confidence interval: [38.2%, 47.2%]

3.5 Case studies and practical applications

The experimental data carried out have identified specific cases where the proposed approach significantly exceeds traditional methods of deception when writing code:

Name change + method rearrangement → 98% similarity after normalization

This case demonstrates the effectiveness of LLM normalization in combined syntactic changes

Comment translation + formatting change → 87% similarity

Confirms resistance to non-fundamental syntactic variations

Swapping for loop with while + changing conditions → 79% similarity

Illustrates successful canonicalization of control structures

The system and architecture designed in this way can be successfully applied to detect plagiarism in student assignments, automatic evaluation of program exercises, analysis of the evolution on a code basis.

3.6 Limitations and future work

Identified limitations through conducted experiments confirmed some of the theoretical limitations of the approach:

Time complexity: Increase by 25-30% due to normalization

Parser dependent: The quality of analysis depends on the accuracy of the syntactic parser

Complex semantic transformations: Still a challenge

Future improvements are based on the experimental results, identifying the following directions for future development:

Integration with ML-based approaches for better recognition of semantic similarities

Support for additional programming languages

Performance optimization through parallel processing of graph components

4 Conclusion

The research and subsequent development of a detection application present a different approach to semantic comparison of program codes, which largely solves the fundamental problem of existing methods, namely their sensitivity to syntactic changes while preserv-

ing the logical structure. Through a combination of semantic normalization with large LLM language models, multispecies graph representation (AST, CFG, DFG) and hybrid embedding merger (GNN + Transformer), the developed system achieves a significant improvement in the accuracy of detecting structural matches. The experimental results demonstrate a convincing effectiveness of the proposed approach. The system reaches an F1-Score of 0.917, which represents an improvement of 4.6% over traditional graph methods and over 21% compared to the basic AST-based approaches. A significant improvement was observed in Type 4 semantic clones (+50.0%), which unequivocally confirms the method's ability to recognize structurally distinct but logically equivalent conversions. The architecture shown successfully addresses the key challenges identified in the literature: resistance to renaming identifiers, canonicalization of control structures, and elimination of syntactic noise. One of the essential processes in the application, in which data is pre-prepared by merging embeddings, allows dynamic balancing between structural (GNN) and semantic (Transformer) information, significantly optimizing the comparison process for different types of code transformations. Although the noted limitations are mainly related to the time complexity and dependence on the accuracy of the syntactic parser used, the proposed approach offers significant advantages for academic and industrial practice. The system can be used in the detection of plagiarism in a learning environment, the automatic evaluation of programming exercises and the analysis of the evolution of codebases. Future research may focus on performance optimization through parallel processing, expanding support for additional programming languages, and integrating with more advanced ML-based semantic analysis techniques. The proven effectiveness of semantic normalization opens up new opportunities to improve existing code analysis systems and offers a solid foundation for developing smarter programmatic analysis tools in the age of artificial intelligence.

References

[1] Allen F., Control flow analysis. *ACM SIGPLAN Notices*, Volume 5, Issue 7, (1970), pp. 1-19, https://doi.org/10.1145/390013.808479

[2] Bahdanau D., Cho. K., Benigo Y., *Neural Machine Translation by Jointly Learning to Align and Translate*, (2014), https://doi.org/10.48550/arXiv.1409.0473.

[3] Bishop, C. M., *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer, (2006), ISBN 978-1-4939-3843-8

[4] Donchev, I., An alternative to virtual functions in modern C++. *Application in training, Mathematics, Computer Science and Education*, Vol. 8, Issue 2, (2025), pp. 144-159

[5] Dulucq S., Touzet H., Analysis of Tree Edit Distance Algorithms, Combinatorial Pattern Matching, *14th Annual Symposium*, CPM 2003, Morelia, Michocán, Mexico, June 25-27, (2003), pp. 83-95, DOI: 10.1007/3-540-44888-87

[6] Halder R., Cortesi A., Abstract program slicing on dependence condition graphs, *Science of Computer Programming*, 78(9), (2013), pp. 1240-1263, DOI: 10.1016/j.scico.2012.05.007

[7] John K., Potika K. (2025). Diverse GNN Encoder-Decoder for Graph Anomaly Detection, *IEEE Conference on Artificial Intelligence (CAI)*, Santa Clara, CA, USA: IEEE, (2025), DOI: 10.1109/CAI64502.2025.00021, pp. 89-94

[8] Prechelt L, Malpohl G, Philippsen M., Finding Plagiarisms among a Set of Programs with JPlag, *JUCS – Journal of Universal Computer Science*, 8(11), (2002), pp. 1016-1038, https://doi.org/10.3217/jucs-008-11-1016

[9] Li Y., Tarlow D., Brockschmidt M., Zemel R., *Gated Graph Sequence Neural Networks*, (2017), https://doi.org/10.48550/arXiv.1511.05493

[10] Lubin J., Ferguson J., Ye K., Yim J., Chasins E. S., Equivalence by Canonicalization for Synthesis-Backed Refactoring. *Proceedings of the ACM on Programming Languages*, Volume 8, Issue PLDI, (2004), 1879-2004.

[11] Petrov M., A model of a two-stage classification system for glial tumors in magnetic resonance imaging, *International Conference AUTOMATICS AND INFORMATICS'2023*, Varna, October 05 – 07, (2023), DOI: 10.1109/ICAI58806.2023.10339009

[12] Putra I. S., Rukmono S., Perdana S. R., Abstract Syntax Tree (AST) and Control Flow Graph (CFG) Construction of Notasi Algoritmik, *International Conference on Data and Software Engineering (ICoDSE)*, Bandung, Indonesia: IEEE, (2021), DOI: 10.1109/ICoDSE53690.2021.9648437

[13] Sendjaja K., Rukmono. A. S., Perdana S. R., Evaluating Control-Flow Graph Similarity for Grading Programming Exercises, *International Conference on Data and Software Engineering (ICoDSE)*, Bandung Indonesia: IEEE, (2021), pp.1-6, DOI: 10.1109/ICoDSE53690.2021.9648464

[14] Wang W., Li G., Ma B., Xia X., Jin Z., Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree, *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (2020), pp. 261-271, DOI: 10.1109/SANER48275.2020.9054857