TELKOMNIKA Telecommunication, Computing, Electronics and Control
Vol. 24, No. 4, August 2026, pp. 1187∼1196
ISSN: 1693-6930, DOI: 10.12928/TELKOMNIKA.v24i4.27683

Methods of finding the maximum common transitive subgraph: experimental comparison

Oleg Sychev, Anton Chupinin
Department of Software for Automated Systems, Faculty of Electronics and Computer Engineering, Volgograd State Technical University, Volgograd, Russia

Article Info

Article history:

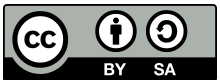
Received Nov 20, 2025
Revised Mar 19, 2026
Accepted Mar 29, 2026

Keywords:

Backtracking
Branch matching
Directed acyclic graph
Dynamic programming
Heuristic method
Maximum common transitive graph

ABSTRACT

The problem of finding a maximum common subgraph (MCS) in a graph has broad applications in practical domains. However, certain scenarios require subgraphs with special properties, such as transitivity, that must be kept during building the subgraph. We formally define the concept of a transitive subgraph, investigate its properties. We study four different algorithms for finding the maximum common transitive subgraph (MCTS), compiled a list of tests aim at comparing graphs after making various changes and evaluated their accuracy and efficiency on a set of test cases. Benchmarking on 64 tests ranks the algorithms by scalability and accuracy: branch matching is the most scalable (> 1000 vertices) and accurate (F1: 0.9907). MCS tree search is viable for graphs of up to ~ 250 vertices (F1: 0.9752). Backtracking is limited to < 30 vertices (accuracy: 0.5625), and brute-force is only feasible for graphs with ≤ 10 vertices, despite its high accuracy (0.9375). We discuss the advantages and disadvantages of each method, the test cases where each method demonstrates a non-optimal MCTS, identify the classes on which the methods work correctly and found that the branch matching method based on the longest common subsequence (LCS) algorithm performed the best.



Corresponding Author:

Oleg Sychev
Department of Software for Automated Systems, Faculty of Electronics and Computer Engineering
Volgograd State Technical University
28 Lenina Prospect, 400005 Volgograd, Russia
Email: oasychev@gmail.com

1. INTRODUCTION

The problem of finding a maximum common subgraph (MCS) has a wide range of applications. The MCS algorithm can be a powerful tool in molecular analysis [1]-[3], chemistry [4], [5], reconfiguration planning of modular robotic systems [6], structural health monitoring [7] tracking [8] and general graph similarity learning [9]-[11].

The problem of finding the MCS is nondeterministic polynomial-time hard (NP-hard) in the general case. However, there are several approximation methods that can find not necessarily optimal but close to optimal solutions [12]. In addition, there are special classes of graphs for which the problem of finding the MCS can be solved in polynomial time [13]. For example, for graphs with unique node labels, the MCS can be found unambiguously using the intersections of the vertex and edge sets. For trees, an algorithm based on dynamic programming is used, which relies on bipartite matching of the vertices of the two trees compared [14].

There is also an approach using quadratic unconstrained binary optimization (QUBO) objective functions for finding the MCS . Parallel [15] and genetic [16] approaches have also been used for finding MCS. Additionally, machine learning (ML) is often used in the MCS search problem [17]-[20]. There are also approaches and attempts to integrate large language models (LLMs) into graph-related tasks [21].

There also exist specialized MCS with additional properties [22]. One of the less-studied cases of the MCS problem is the common subgraph that preserves transitive dependencies between nodes when deleting nodes that do not fit the conditions of the common subgraph. Preserving transitive dependencies in a directed acyclic graph (DAG) allows one to retain relationships like "X precedes Y" even when an intermediate node cannot be included in the common subgraph. One of practical cases of preserving transitivity when building the MCS is the comparison of software repositories. Repositories that use branching can be understood as DAG storing the transitive relationships of precedence between revisions. The previous studies have demonstrated efficiency of using the longest common subsequences (LCS) algorithm in teaching programming and foreign languages [23], [24].

Git is an important tool in modern software development, and learning how to use version control tool is a crucial step in training highly skilled information technology (IT) specialists . Numerous studies show that using specialized automatic tutors in student education can significantly accelerate the learning process [25], [26]. But implementing an effective tutor requires the ability to find the maximum common transitive subgraph (MCTS) to identify misplaced, missing, and extraneous nodes (change sets) in the student's answer [27].

In this work, we propose a variant of the MCS problem that accounts for transitive relationships in DAGs. We define the MCTS, compare several methods for finding it, and analyze the comparison results. The paper is structured as follows. Section 2 provides the key definitions necessary to understand the subject. Section 3 describes the prospective methods of finding the MCTS: brute-force search of all possible variants, an adaptation of dynamic programming method for finding the maximum common subtree, a heuristic method based on matching each branch of graph G_1 with the closest branch of graph G_2 and calculating their LCS and backtracking method for unique node label. Section 4 outlines the core test cases and provides a rationale for the selected test cases. Section 5 presents the experimental results, and section 6 contains the study conclusions.

2. KEY DEFINITIONS

In this section, the basic concepts and terminology used throughout the paper are introduced.

Definition 1. *A graph is denoted as $G = (V, E)$, where V is the set of vertices (nodes) and $E \subseteq V \times V$ is the set of edges (connections between vertices).*

Definition 2. *A DAG is a directed graph that does not contain directed cycles.*

In other words, starting from a vertex v and following the direction of the arcs, it is impossible to return to the original vertex v.

Definition 3. *Let $G = (V, E)$ be a directed graph. The complete transitive removal of the vertex v is the operation that for all distinct vertices $u_1, \ldots, u_n \in V$ and $w_1, \ldots, w_m \in V$ satisfying:*

$$\forall i, j : (u_i, v) \in E \wedge (v, w_j) \in E \tag{1}$$

produces a graph $G' = (V', E')$ where:

$$V' = V \setminus \{v\}, \quad E' = (E \setminus \{(u_i, v), (v, w_j) \mid \forall i, j\}) \cup \{(u_i, w_j) \mid \forall i, j\} \tag{2}$$

That is, when a vertex v is removed, for each of its ancestors, an edge is created connecting it to each of vertex v's descendants.

Definition 4. *A graph $G_1 = (V_1, E_1)$ is called a transitive subgraph of $G = (V, E)$ if there exists a graph G_1' isomorphic to G_1 that can be obtained from G by applying a finite number (possibly zero) of transitive removals of the vertex.*

Definition 5. *A graph G^* is called the MCTS of graphs G_1 and G_2 if: (1) G^* is a transitive subgraph of both G_1 and G_2; (2) There exists no other common transitive subgraph G' of G_1 and G_2 with $|V(G')| > |V(G^*)|$.*

Figure 1 shows the MCTS of the two graphs.

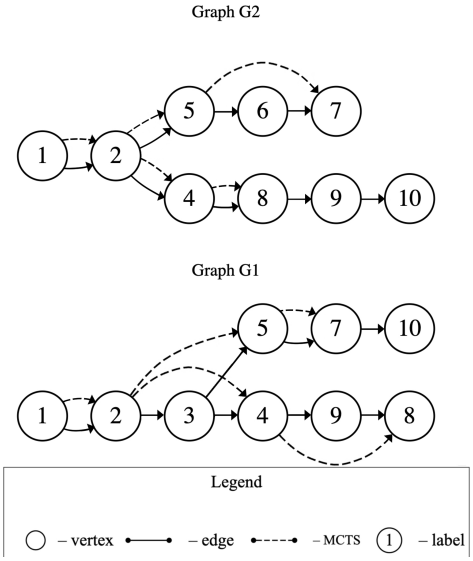


Figure 1. MCTS

3. METHOD

3.1. Brute-force search

The brute-force method for finding the MCTS is based on analyzing all possible mappings between the vertices of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Assuming $|V_1| \leq |V_2|$, the algorithm examines all possible bijections between subsets $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$, where $|V_2'| = |V_1'|$. At the same time, the graph structure is preserved: if $v_1, u_1 \in V_1$ and $v_2, u_2 \in V_2$, then under the bijection mapping v_1 to v_2 and u_1 to u_2, the vertex relations must be maintained: if u_1 is a descendant of v_1, then u_2 must be a descendant of v_2. The goal is to select the mapping with the maximum number of correct vertex bijections.

First, the method selects the k vertices of G_1 and matches them to k vertices of G_2. To search and verify the number of valid bijective vertex mappings for a complete graph matching, it is necessary to traverse all vertices of both graphs using depth-first search (DFS) or breadth-first search (BFS) and to store all transitive connections for each graph before comparing them. Note that, for labeled graphs, the number of possible mappings can be reduced by requiring label equality for the corresponding vertices, which can significantly speed up the solution in practical applications.

The brute-force method guarantees an exact solution for finding the MCTS, as it exhaustively enumerates all possible subgraphs and selects the optimal one, but due to the complexity of $O(n! \cdot 2^n)$, where n is the number of vertices in the largest graph, its use is limited to small graphs. The key advantage of exhaustive search is its absolute accuracy, though its exponential complexity limits practical applications. For labeled graphs, optimization is possible through vertex-label matching, which reduces the search space. Despite its computational limitations, this method remains the gold standard for verifying approximation algorithms.

3.2. Adapting the maximum common subtree search method using dynamic programming to solve the problem of computing the maximum common transitive subtree.

For solving problems related to finding the maximum common subtree of two trees [14], there exists a specific algorithm that employs dynamic programming and a maximum-weight matching algorithm. Adapting this method to find the MCTS could allow solving a wide range of problems with a lower time complexity that brute-force search, allowing the analysis of larger graphs.

The algorithm uses an approach that determines the value of a parent vertex through the values of its child vertices. Three main cases need to be considered: matching vertices, matching one vertex with the descendants of the other, and matching the second vertex with the descendants of the first. If a vertex u from graph G_1 can be matched with a vertex v from graph G_2, then a set of pairs of vertex matching from graph

G_1 to graph G_2 should be created for these vertices, and the pair (u, v) should be added. Next, it is necessary to iterate through all descendants of u and v and construct a weight matrix for each pair of descendants of the vertices u and v. Then, using an algorithm to find the maximum weight matching, we determine the best pairs among the descendants. For the cases of comparing vertices with descendants, it is necessary to select a pair where the weight of the pair is maximal. Then, for the current pair of vertices, it is necessary to choose the best result among the three presented. The complete algorithm is shown in Algorithm 1.

Algorithm 1 Maximum common transitive subtree

```
procedure MaxTransitSubTree(u, v, G1, G2, dp)
    left ← ∅                                              ▷ Case 1: match u and v
    center ← ∅                                            ▷ Case 2: skip v, recurse on children of u
    right ← ∅                                             ▷ Case 3: skip u, recurse on children of v
    if u = v then                                         ▷ Only match if labels agree
        left ← left ∪ {(u, v)}
        Initialize matrix M
        for all u' ∈ Children(G1, u) do
            for all v' ∈ Children(G2, v) do
                subgraph ← MaxTransitSubTree(u', v', G1, G2, dp)
                Add subgraph to M[u', v']
            end for
        end for
        match ← MaxWeightMatching(M, dp)                  ▷ Find best child pairing
        left ← left ∪ {match}
    end if
    for all u' ∈ Children(G1, u) do
        center ← max(center, MaxTransitSubTree(u', v, G1, G2, dp))
    end for
    for all v' ∈ Children(G2, v) do
        right ← max(right, MaxTransitSubTree(u, v', G1, G2, dp))
    end for
    dp[u, v] ← max(left, center, right)                   ▷ Choose largest set
    return dp[u, v]
end procedure
```

Since the Hungarian algorithm (with time complexity $O(n^3)$) is used to find the maximum weight matching, the total time complexity of the algorithm is $O(n^5)$, where n is the number of vertices in the graph.

3.3. Heuristic method based on matching branches of two graphs

This heuristic method is based on the idea that different branches in a Git repository have a diverse set of changes. Each change can be viewed as a label on the vertices (commits). It follows that, when two graphs have similar characteristics to a Git repository, it is necessary to match each branch of graph G_1 to the branch of graph G_2 that is most similar, such that each branch of G_1 is matched with only one branch from G_2, and each branch of G_2 is matched with only one branch from G_1. Subsequently, for each such pair, it is necessary to find the LCS of vertices, considering that some vertices may contain multiple labels, and there may be cases of incomplete label correspondence for vertices that we include in the LCS. To this end, it will be necessary to use a modified version of the LCS algorithm such that, in addition to maximizing the length of the subsequence, it minimizes the number of mismatched labels.

$$L[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ \max(L[i-1,j], L[i,j-1], L[i-1,j-1] + v[i][j]), & \text{if } x_i \text{ can compare } y_j, \\ \max(L[i-1,j], L[i,j-1]), & \text{otherwise.} \end{cases} \tag{3}$$

Let $B(G)$ denote the list of all branches in graph G. Then, the number of matched branch pairs between graphs G_1 and G_2 is equal to $\min\{|B(G_1)|, |B(G_2)|\}$. The time complexity of the LCS algorithm is $O(n \cdot m)$, where n and m are the lengths of the sequences; in our case, this is the number of vertices in the branch. Considering all of this, the overall complexity of the heuristic method is $O(\min\{|B(G_1)|, |B(G_2)|\} \cdot n \cdot m)$, which is significantly lower than the time complexities of the algorithms that were described above.

Some vertices may belong to several branches at once, causing a single vertex to participate multiple times in the LCS search operations (complexity $O(n^2)$), which implies that the algorithm's complexity is $O(n^2 \cdot k)$, where n is the number of vertices and k is the number of branches.

3.4. Backtracking method tree for unique node label

It is worth noting that primarily for educational purposes, to teach students how to work with version control system (VCS), repositories are used where diffs in commits are unique. These repositories can be viewed as trees with unique node labels, where each vertex is assigned exactly one label, implying that for every vertex $v \in G_1$, there is a unique corresponding vertex $u \in G_2$.

For each vertex of each tree, we compute the list of all its ancestors. Then, we perform a DFS algorithm of the first tree G_1, proceeding from the root to the leaves. During this traversal, for each vertex we examine the list of its ancestors in the second tree G_2 and the corresponding branches in G_1.There is no need to consider branches all of whose vertices belong to another branch; formally, we disregard any branch B such that $\exists B' \neq B$ with $V(B) \subseteq V(B')$.

Two branches are *conflicting* if they contain a pair of conflicting vertices. Vertices u and v are conflicting if u is not an ancestor of v in G_1, while u' is an ancestor of v' in G_2. Formally, (u, v) are conflicting $\iff \left(u \not\preceq_{G_1} v\right) \wedge \left(u' \preceq_{G_2} v'\right)$ where $\preceq_G$ denotes the ancestor relation in tree G.

During the backtracking phase once we have obtained the list of active branches containing v for each vertex v, we handle the case where v has two or more direct children. If $\deg^+(v) \geq 2$, (i.e., v has direct children $u_1, u_2, \ldots, u_n$ with $n \geq 2$), we must select a set of non-conflicting branches – each containing one of the children $u_1, u_2, \ldots, u_n$ – and attempt to merge them into a single branch, which is added to the global branch list. For this algorithm, the solution is the largest branch by cardinality. The time complexity of the backtracking method is $O(n!)$, where n is the number of branches in the graph.

4. TEST SAMPLE

When creating the tests necessary to verify the correctness of the methods for finding the maximum transitive subgraph, we based on the properties of possible practical usage: comparing two Git repositories of the same project when learning to use Git. The number of vertices for the majority of test graphs is 7-12 vertices (up to 20 vertices), which is sufficient for testing basic graph transformation cases.

We considered different types of changes in the graph (which are interpreted as errors). One class of errors different placement of vertex labels (i.e., a label being removed from a vertex, an extra label being added to a vertex, or the label moved to another vertex). The latter case can be interpreted as a combination of two operations: deleting the label from one vertex and adding it to another. For the errors of this type, a linear graph (also called a bamboo) is sufficient.

The other kind of differences is the differences in the order of vertices. Examples include deleting an leaf vertex, deleting a vertex from the middle of the graph (which has descendant vertices), adding a vertex to the middle, adding a vertex to the end and changing the vertex placement. There are also more specific changes, such as merging several vertices into one (removing a vertex and adding all its labels to another vertex) and splitting a single vertex into a few vertices. For those changes, directed trees with 2–3 branches are used. This is important for testing the methods' ability to correctly identify changes in the branching structure. It is also important to verify if the method correctly handles the merging of multiple branches into one and accurately processes other changes when the branches are merged. We also made complex tests for combinations of the graph changes described above.

Table 1 shows the main parameters that were combined to create different tests and their values; Table 2 shows a part of the test sample using the short names of parameters.

Table 1. Test parameters

Test parameter	Short name	Values
Graph type	type	DAG, directed tree
Number of vertices	vertices	1, 3-8, 12-15, 16-20
Number of branches	branches	1, 2-3
Number of vertex labels	labels	0, 1, 2 or more
Number of changes	diff	0, 1, 2 or more

Table 2. Test cases

No.	Name	G_1 type	G_1 vertices	G_1 labels	G_1 branches	G_2 type	G_2 vertices	G_2 labels	G_2 branches	diff.
1	Equal trees	Tree	10	1-2	1	Tree	10	1-2	1	0
2	Equal DAGs	DAG	14	1	3	DAG	14	1	3	0
3	Vertex deleted in middle	Tree	7	1	3	Tree	6	3	1	1
4	Leaf deleted	Tree	7	1	3	Tree	6	3	1	1
					...					
20	Branch deleted	Tree	9	1	3	Tree	7	2	1	2
21	Reversed branch	Tree	6	1	1	Tree	6	1	6	5
22	DAG branch deleted	DAG	11	1	3	Tree	9	2	1	2
					...					
35	Label added	Tree	10	1	2-3	Tree	10	1	1	1
36	Label missing	Tree	10	1	2-3	Tree	10	1	1	1
					...					
50	One vertex split into several	Tree	6	1-4	1	Tree	7	1-2	1	2
					...					
63	Complex test 5	Tree	8	2-3	2	Tree	8	2-3	2	4
64	Complex test 6	Tree	8	1-2	2	Tree	10	2-3	2	4

This test suite covers most of the error cases that arise when students work independently to master version control systems and learn repository-editing skills using the provided tools.

5. RESULTS

This section presents the results of testing the listed algorithms on the listed set of tests that simulate the results of performing tasks for students to work with control systems. A comparison of the methods with the performance of the DeepSeek-V3.2 model is also carried out, for which a prompt was specifically engineered to find MCTS. Metrics such as precision, recall, and F1 will be used to evaluate the accuracy of MCTS determination. For the confidence interval of the pass rate metric, the Wilson score interval with a 95% confidence level is used. For this purpose, the following vertices are defined: true positive (TP) – a pair of vertices matched by the algorithm that is present in the reference set. False positive (FP) – a pair of vertices predicted by the algorithm but absent in the reference set. False negative (FN) – a pair of vertices from the reference set not found by the algorithm. True negative (TN) – any other pair of vertices (not predicted by the algorithm and not included in the reference set). All tests were conducted on a computer with an Apple M2 processor and 16 GB of random access memory (RAM).

The DeepSeek model successfully passed 37 tests (pass rate 0.58). However, it frequently made errors when matching graph nodes, indicating certain difficulties in correctly interpreting the structure of graph data or their interconnections. The brute-force method demonstrated a high pass rate (0.94). All failed tests had excessively long execution times, which necessitated the termination of these tests. However, it scales poorly with the graph size. For instance, on a graph with just 11 vertices, this method can take about 2 minutes to find an answer, while other methods solve it in under one second. Furthermore, there is a significant issue with memory consumption. For a graph of 12 vertices, the method already requires more than 5 GB of RAM.

The backtracking method for tree-based unique node labeling showed the lowest pass rate among all algorithms tested (0.56). The primary set of tests where this algorithm incorrectly identifies the maximum transitive subgraph involved graphs with vertices containing multiple labels and graphs with repeated labels. Additionally, all tests in which the graphs were DAGs indicate that this algorithm is unsuitable for checking repositories with multiple branch merges. The method demonstrates acceptable runtime performance for graphs with a number of vertices $n < 30$. However, as n increases beyond this threshold and the graph's branching factor grows, the execution time scales accordingly.

The next algorithm in terms of pass rate (0.83) is the method adapting dynamic programming for finding the maximum transitive subtree. The primary class of problems where this method made errors involves comparing graphs after removing a common parent of multiple branches. In this case, the correct solution would be to match corresponding vertices in the compared graphs, but the method instead proposes discarding smaller branches and retaining only the largest branch in the matching. Performance testing revealed that the

proposed method maintains fast execution on graphs with the number of vertices $n > 100$, which is comparable to the scale of real-world repositories.

The heuristic method based on matching each branch of the graph performed best on this test set, achieving a pass rate of 0.98. It successfully handled tasks involving graphs without branch merges, repositories with merges, graphs with repeated labels, and graphs with unique label sets. It demonstrated high speed in executing test cases. The method only performed poorly in a test where two or more branches of a graph had identical label sets but differed in the order of vertex arrangement, leading to incorrect matching of corresponding branches across different graphs and failure to identify the MCTS. This method proved to be the most performance, maintaining fast execution on graphs with $n > 1000$ vertices.

All methods are in one way or another suited for a specific class of graphs on which they correctly locate the MCTS. When the MCTS search is incorrect, the methods return not the maximum, but one of the common transitive subgraphs. It is possible to use several methods at once to search for the MCTS and choose the largest output. The Table 3 presents a comprehensive comparison of these algorithms. Table 4 and Figure 2 demonstrate the program's execution time as a function of the number of vertices in the graph. Figure 3(a) presents an example of two graphs on which the backtracking method performs an incorrect search; Figure 3(b) shows an example of two graphs where the adapted dynamic programming method conducts an incorrect search; Figure 3(c) illustrates an example of two graphs where the heuristic matching-based method fails to perform a correct search.

Table 3. Test results

No	Method	Passed	Failed	Pass rate	CI 95%	Precision	Recall	F1-score
1	Bruteforce	60	4	0.9375	0.8500–0.9754	1.0000	0.8843	0.9386
2	MCS tree search	53	11	0.8281	0.7178–0.9012	0.9952	0.9560	0.9752
3	Branch matching	63	1	0.9844	0.9167–0.9972	0.9953	0.9861	0.9907
4	Backtracking	36	28	0.5625	0.4409–0.6771	0.9922	0.8796	0.9325
5	deepseek-v3.2	37	27	0.5781	0.4561–0.6913	0.8986	0.8819	0.8902

Table 4. Algorithm performance by problem size (seconds)

	Number of vertices												
Method	9	10	11	25	30	32	34	50	100	200	250	1000	5000
Branch	0.20	0.20	0.20	0.22	0.23	0.22	0.22	0.23	0.22	0.26	0.29	0.64	8.00
MCS tree search	0.03	0.03	0.03	0.07	0.10	0.11	0.10	0.35	6.00	120.0	360.0	–	–
Backtracking	0.03	0.03	0.03	0.64	8.00	24.00	105.0	–	–	–	–	–	–
Bruteforce	2.00	20.00	99.00	–	–	–	–	–	–	–	–	–	–

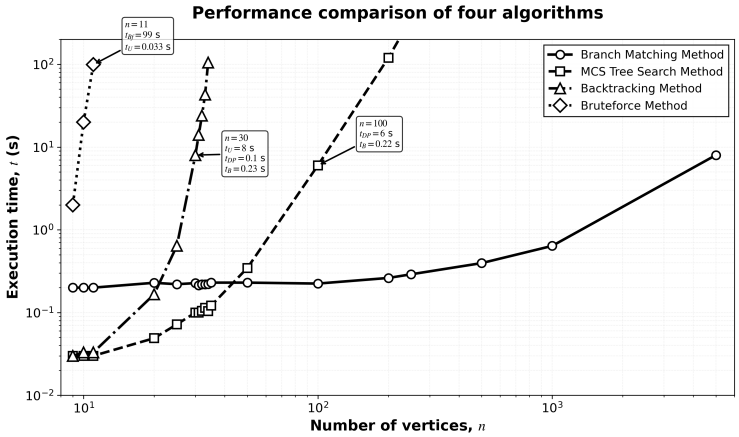


Figure 2. Performance comparison of four algorithms

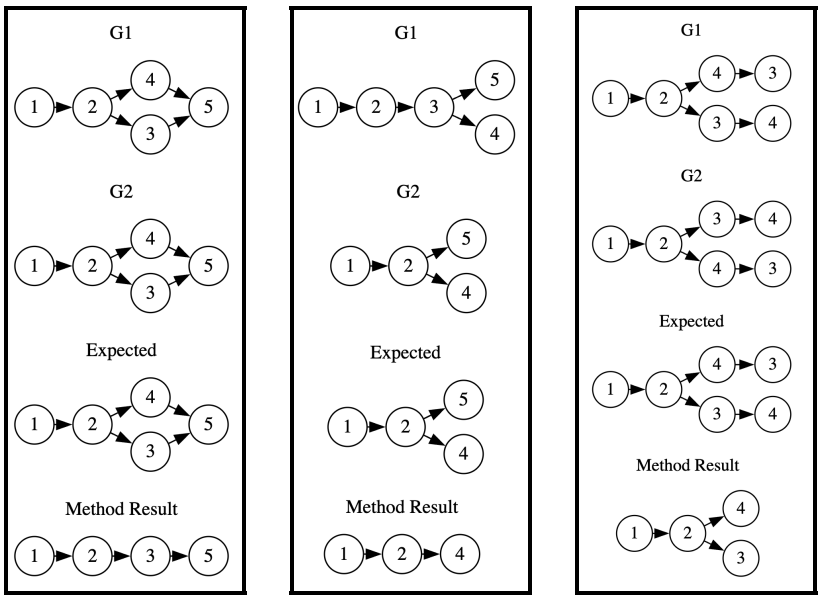


Figure 3. Graphs for which the methods fail: (a) backtracking, (b) MCS tree search, and (c) branch matching

6. CONCLUSION

Finding MCTS is necessary for finding changes in transitive DAGs, which can be used, to verify students' works when learning to use version control systems like Git. We considered four methods of finding MCTS and designed a test sample to study their efficiency when different changes to graphs were made. Experimental results confirmed the theoretical expectation that the worst-performing methods would be the LLM-based method (0.58) and the backtracking method (0.56). The brute-force method achieved perfect accuracy, but only on small graphs $(n < 10)$. The best overall performance was demonstrated by the branch matching method (0.98), which failed only on graphs containing highly similar branches. The MCS tree search method yielded a solid result (0.83), although it underperformed relative to initial expectations. Although none of the methods solved all test cases, we identified distinct graph classes where each method tends to fail. That enables dynamic selection or combination of methods based on graph structure – for example, by multiplexing approaches or running several in parallel and selecting the best result using a "bag of experts" strategy – to further improve overall accuracy. The studied methods will be used in developing an intelligent tutoring system for teaching version control, which is important for training programmers and IT specialists. Finding MCTS between the correct solution repository and the student's repository will allow detection of missing, extraneous, misplaced and wrong commits, which can be show to the student with appropriate messages. That will allow training with feedback without human in the loop, which significantly increases the number of solved tasks and the resulting skills compared to manual verification.

FUNDING INFORMATION

The study was carried out with the support of the Center for Digital Scientific and Educational Projects and Developments in the Field of Industrial Artificial Intelligence (C2RED-AI) of Volgograd State Technical University, created as part of the implementation of top-level educational programs in the field of artificial intelligence (Agreement No. 70-2025-000756).

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Oleg Sychev	✓	✓		✓		✓		✓	✓	✓		✓	✓	
Anton Chupinin		✓	✓	✓	✓	✓		✓	✓		✓			

C : **C**onceptualization
M : **M**ethodology
So : **So**ftware
Va : **Va**lidation
Fo : **Fo**rmal Analysis
I : **I**nvestigation
R : **R**esources
D : **D**ata Curation
O : Writing - **O**riginal Draft
E : Writing - Review & **E**diting
Vi : **Vi**sualization
Su : **Su**pervision
P : **P**roject Administration
Fu : **Fu**nding Acquisition

CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

DATA AVAILABILITY

The data supporting the findings of this study are openly available in the GitHub Project Repository at https://github.com/vakhew1900/master-dis/tree/main

REFERENCES

[1] M. Stroet *et al.*, "OfraMP: a fragment-based tool to facilitate the parametrization of large molecules," *Journal of Computer-Aided Molecular Design*, vol. 37, no. 8, pp. 357–371, 2023, doi: 10.1007/s10822-023-00511-7.

[2] L. Schietgat *et al.*, "Automated detection of toxicophores and prediction of mutagenicity using PMCSFG algorithm," *Molecular Informatics*, vol. 42, no. 3, p. 2200232, 2023, doi: 10.1002/minf.202200232.

[3] N. Parisutham, "How do centrality measures help to predict similarity patterns in molecular chemical structural graphs?" *Artificial Intelligence Chemistry*, vol. 1, no. 2, p. 100007, 2023, doi: 10.1016/j.aichem.2023.100007.

[4] T.-L. Phan *et al.*, "Reaction rebalancing: a novel approach to curating reaction databases," *Journal of Cheminformatics*, vol. 16, no. 1, p. 82, 2024, doi: 10.1186/s13321-024-00875-4.

[5] Y. Chang *et al.*, "High-entropy alloy electrocatalysts screened using machine learning informed by quantum-inspired similarity analysis," *Matter*, vol. 7, no. 11, pp. 4099–4113, 2024, doi: 10.1016/j.matt.2024.10.001.

[6] C. Zhang, L. Zhou, and Y. Li, "Pareto optimal reconfiguration planning and distributed parallel motion control of mobile modular robots," *IEEE Transactions on Industrial Electronics*, vol. 71, no. 8, pp. 9255–9264, 2024, doi: 10.1109/TIE.2023.3321997.

[7] J. Gosliga, D. Hester, K. Worden, and A. Bunce, "On population-based structural health monitoring for bridges," *Mechanical Systems and Signal Processing*, vol. 173, p. 108919, 2022, doi: 10.1016/j.ymssp.2022.108919.

[8] U. Ahmed, G. Srivastava, Y. Djenouri, and J. C.-W. Lin, "Knowledge graph based trajectory outlier detection in sustainable smart cities," *Sustainable Cities and Society*, vol. 78, p. 103580, 2022, doi: 10.1016/j.scs.2021.103580.

[9] Z. Lan, B. Hong, Y. Ma, and F. Ma, "More interpretable graph similarity computation via maximum common subgraph inference," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 11, pp. 6588–6599, 2024, doi: 10.1109/TKDE.2024.3387044.

[10] Z. Liu, Y. Chen, N. Liu, J. He, and D. Li, "Graph2region: Efficient graph similarity learning with structure and scale restoration," *IEEE Transactions on Knowledge and Data Engineering*, vol. 37, no. 12, pp. 7213–7225, 2025, doi: 10.1109/TKDE.2025.3617461.

[11] C. Zou, G. Lu, L. Du, X. Zeng, and S. Lin, "Graph similarity learning for cross-level interactions," *Information Processing & Management*, vol. 62, no. 1, p. 103932, 2025, doi: 10.1016/j.ipm.2024.103932.

[12] L. Cardone and S. Quer, "The multi-maximum and quasi-maximum common subgraph problem," *Computation*, vol. 11, no. 4, p. 69, 2023, doi: 10.3390/computation11040069.

[13] D. J. Aldous, "On the largest common subtree of random leaf-labeled binary trees," *SIAM Journal on Discrete Mathematics*, vol. 36, no. 1, pp. 299–314, 2022, doi: 10.1137/20M1347504.

[14] A. Gupta and N. Nishimura, "Finding largest subtrees and smallest supertrees," *Algorithmica*, vol. 21, no. 2, pp. 183–210, 1998, doi: 10.1007/PL00009212.

[15] V. Vasilchikov, "Recursive-parallel algorithm for solving the maximum common subgraph problem," *Automatic Control and Computer Sciences*, vol. 58, no. 7, pp. 827–835, 2024, doi: 10.3103/S0146411624700287.

[16] C. Valenti, "A genetic approach to the maximum common subgraph problem," in *Proceedings of the 20th International Conference on Computer Systems and technologies*, 2019, pp. 98–104, doi: 10.1145/3345252.3345272.

[17] S. Quer, T. Madeo, A. Calabrese, G. Squillero, and E. Carraro, "Node embedding and cosine similarity for efficient maximum common subgraph discovery," *Applied Sciences*, vol. 15, no. 16, p. 8920, 2025, doi: 10.3390/app15168920.

[18] I. Roy, S. Chakrabarti, and A. De, "Maximum common subgraph guided graph retrieval: late and early interaction networks," *Advances in Neural Information Processing Systems*, vol. 35, pp. 32 112–32 126, 2022.

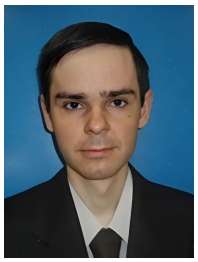
[19] K. S. Yow, N. Liao, S. Luo, and R. Cheng, "Machine learning for subgraph extraction: Methods, applications and challenges," *Proceedings of the VLDB Endowment*, vol. 16, no. 12, pp. 3864–3867, 2023, doi: 10.14778/3611540.3611571.

[20] Z. Yan, C. Ding, L. Ma, L. Cao, and H. You, "Rotated graph similarity computation via graph transformer networks," *Neurocomputing*, vol. 658, p. 131474, 2025, doi: 10.1016/j.neucom.2025.131474.

[21] X. Ren, J. Tang, D. Yin, N. Chawla, and C. Huang, "A survey of large language models for graphs," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6616–6626, doi: 10.1145/3637528.3671460.

[22] E. de Gastines and A. Knippel, "Formulations for the maximum common edge subgraph problem," *Discrete Applied Mathematics*, vol. 346, pp. 115–130, 2024, doi: 10.1016/j.dam.2023.11.044.
[23] O. Sychev, "Questions for teaching phrase building with automatic feedback," *Software Impacts*, vol. 15, p. 100461, Mar. 2023, doi: 10.1016/j.simpa.2022.100461.
[24] E. Novozhenina, O. Sychev, O. Toporkova, and O. Evtushenko, "Teaching english word order with correctwriting software," in *Computational Science and Its Applications – ICCSA 2021*. Springer International Publishing, 2021, p. 681–692, doi: 10.1007/978-3-030-86970-0_47.
[25] A. W. F. Kouam, "The effectiveness of intelligent tutoring systems in supporting students with varying levels of programming experience," *Discover Education*, vol. 3, no. 1, p. 278, 2024, doi: 10.1007/s44217-024-00385-3.
[26] A. Lieb and T. Goel, "Student interaction with NewtBot: An LLM-as-tutor Chatbot for Secondary Physics Education," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '24. New York, NY, USA: Association for Computing Machinery, 2024, doi: 10.1145/3613905.3647957.
[27] O. Sychev and D. Mamontov, "Automatic error detection and hint generation in the teaching of formal languages syntax using correctwriting question type for moodle lms," in *2018 3rd Russian-Pacific Conference on Computer Technology and Applications (RPC)*, 2018, pp. 1–4, doi: 10.1109/RPC.2018.8482125.

BIOGRAPHIES OF AUTHORS



Oleg Sychev received the M.Sc. degree in Computer Science and Engineering from Volgograd State Technical University, Russia. He is a Associate Professor of Software Engineering Department, Volgograd State Technical University, Volgograd, Russia. He is currently working on the design and implementation of intelligent tutoring systems capable of presenting worked examples, determining semantic errors, and providing explanatory feedback on why an answer is incorrect—specifically detailing what subject-domain rules are broken.His research interests involves developing systems to ask follow-up questions to stimulate student thinking and automatically classify generated questions to minimize direct human involvement in the creation of assessments, all based on a single subject-domain model. His professional experience spans 23 years, including 22 years in scientific and pedagogical work. He can be contacted at email: oasychev@gmail.com.

Anton Chupinin received the B.Eng. degree in Software Engineering from Volgograd State Technical University (VSTU), Volgograd, Russia, in 2024. He is currently a graduate student of the M.Eng. degree in the same field. He is also a backend developer in the Volgablob company. His research interests involves developing intelligent tutoring systems for teaching computer science. He can be contacted at email: antchupinin@gmail.com.