A Structural Approach to Program Similarity Analysis

Miller Trujillo, Silvia Takahashi and Nicolás Cardozo*

Systems and Computing Engineering Department, Universidad de los Andes - Bogotá, Colombia

Abstract

Analyzing program similarity is useful in automated assessment grading, plagiarism detection, or proving refactor equivalence. The precondition of existing approaches to program similarity is that the programs to compare are expected to be similar, as for example in the case of program evolution. We note that existing similarity approaches are not appropriate for analyzing programs that may use different implementations to solve a problem. Moreover, existing approaches still are not able to measure the actual differences between program implementations. We propose an improvement to existing similarity techniques using the control flow graph representation of programs. In our approach, we exploit the input graphs' structure and avoid costly subgraph isomorphism comparisons, to reach a metric to measure similarity between programs. Furthermore, we obtain a better similarity detection when comparing similar and different programs, with respect to the state-of-the-art. To validate our approach, we use a new corpus of competitive programming problems to discover similarity between solutions submitted by contestants. Our results show that our approach correctly detects similarity between different program implementations with a precision improvement of up to 77% in some cases, and marks as different those implementations to unrelated programs, with a performance comparable to existing approaches.

Keywords

Algorithmic diversity, Program similarity, Code clones

1. Introduction

In software engineering, *program similarity* refers to how closely related is the behavior and structure of two programs. Analyzing similarity between programs is useful in application domains like automated assessment grading in programming courses, plagiarism detection, and refactor equivalence. Current approaches to analyze similarity between programs focus on semantic similarity (*i.e.,* behavior), comparing programs that are known, or expected, to be similar in advance (*e.g.,* different versions of a program). That is, when analyzing the similarity between different versions of a program, most of the structure of the program is expected to remain the same, with only a few lines of code changing. For example, automated grading systems expect students' programs to be similar to one of the model programs provided by course instructors. This, however, rises questions about similarity between programs describing the same behavioral purpose but using completely different algorithmic ideas and structures. This question is of particular interest for algorithmic diversity in evolutionary environments [1]. When comparing programs that are actually similar, we answer questions like: Is program A similar to program B? In some cases, this question is more restrictive and only concerned with checking semantic equivalence between two programs, and not similarity.

In this paper, we study program similarity between any two programs. We are interested in programs that correspond to the same functional behavior –that is, programs that respond the same output to the same input, but that are built with a different structure. As programs with the same response to a same input can still differ, similarity requires a deeper perspective: evaluating programs' structure. Take for example the case of search algorithms, is QuickSort most similar to MergeSort or HeapSort? Even though the three sorting algorithms have the same output for the same input, their time complexity and abstract idea differ. The structure of the programs is also different. Therefore, we propose an algorithm

SQAMIA 2024: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 9–11, 2024, Novi Sad, Serbia

*Corresponding author.

ma.trujillo10@uniandes.edu.co (M. Trujillo); s.takahas@uniandes.edu.co (S. Takahashi); n.cardozo@uniandes.edu.co (N. Cardozo)

0000-0002-1094-9952 (N. Cardozo)

that takes into account programs' structure to assess similarity. We define two programs to be similar if they define similar control structures, have a similar usage of variables and data structures, and follow the same order of the program statements. It is not the same to sort an array at the very beginning of a program, or to sort it at the end of the program. Program pre-conditions exist if we sort at the beginning; we could write completely different programs from that point on.

The motivation for measuring program similarity comes from algorithmic diversity [1], which is concerned with improving systems' performance by utilizing multiple different algorithm implementations for a program feature at the same time. Example domains in which algorithmic diversity can present improvements are: security, N-version programming, collective systems like smart camera networks, or load balancers. While there are areas in which using different algorithmic solutions for a given problem can be beneficial, there is still a lack of measuring tools to dictate how different the actual implementations are. This is precisely one of the major drawbacks for techniques like N-version programming, and a contribution of our work.

Our proposal (Section 2) tackles the problem of program similarity comparing the structure of programs, within a same domain, that are meant to be different beforehand [2]. Specifically, we use the Control Flow Graph (CFG) [3] program representation and add information about the graph structure as input for our comparison [4]. Furthermore, we avoid complicated subgraph isomorphism detection [5] by normalizing the similarity value of the CFG nodes to the total number of nodes in the graph for each program. To validate our approach (Section 3), we created a corpus of 566 C++ programs extracted from the competitive programming platform Codeforces. Our corpus consists of multiple implementations of five different programming problems. Our results (Section 4) show a clear clustering of the different solutions to each of the problems, demonstrating the effectiveness of our approach in detecting similar and disimilar implementations across different multi-function programs. The results present a similar performance with respect to existing approaches, showing a precision improvement of up to 77%. Additionally, our approach is effective in detecting disimilar implementations as such, providing a balanced metric to program similarity.

Concretely, the main contributions of our paper are:

- Definition of a new similarity metric and algorithm between different programs.
- Creation of a corpus to evaluate similarity between any two programs, rather than comparing program versions differentiating in small deltas.
- Evaluation of the proposed metric with respect to the state-of-the-art in node base similarity analysis.

2. Similarity Among Programs

2.1. Motivating Example

To motivate the concept of similarity between programs, and to assess the perspectives of functional behavior and structural similarity, we consider as example single function algorithms in three concrete cases: linear search (Figure 1 left), max (Figure 2 left), and binary search (Figure 3 left).

```
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int n = 5, k = 4;
    int ans = -1;
    for (int i = 0; i < n; ++i)
        if (a[i] == k) {  ans = i;  }
    return 0;
}
```

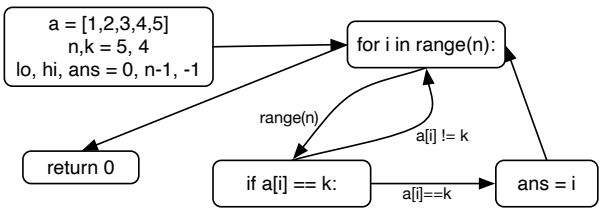


Figure 1: Linear search C++ implementation with its CFG

```
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int n = 5;
    int ans = -1;
    for (int i = 0; i < n; ++i)
      if (ans == -1 || a[i] >= a[ans]) { ans
            = i; }
    return 0;
 }
```

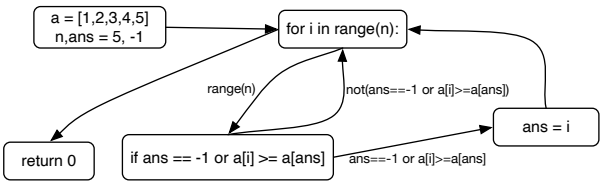


Figure 2: Max C++ implementation with its CFG

```
int main() {
    int a[] = {1, 2, 3, 4, 5};
    int k = 4, n = 5;
    int lo = 0, hi = n - 1, ans = -1;
    while(lo < hi) {
       int mid= (lo + hi)/2;
       if(k <= a[mid])  { hi = mid; }
       else  { lo=mid+1; }
    }
    if(a[lo] == k) {  ans = lo;  }
    return 0;
}
```

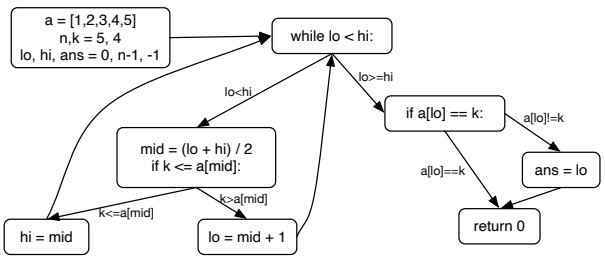


Figure 3: Binary search C++ implementation with its CFG

All programs receive a number array as parameter and iterate over it to compute the corresponding operation. The main functionality of the programs is characterized by a loop with an internal conditional statement, as seen in the code snippets. To assure the similarity analysis is agnostic to specific syntax elements, the loops and conditionals use different syntax (*e.g.,* **`for`** vs. **`while`** loops). The CFGs corresponding to each of the algorithms are on the right-hand side of Figures 1 to 3.

These examples raise the question of which of these algorithms are more cloesely related to the others (if any)? Linear search should be more similar to binary search, as these two algorithms belong to the same domain, and their end result is the same to the same given input. However, linear search is structurally closer to max, as their CFGs are closer to each other than to binary search. We can conclude that focusing on just one dimension of similarity analysis can lead to erroneous assessments. As a consequence, with our proposal we measure the similarity of programs that are functionally equivalent but that have different structures to identify diversity or different versions, as in the case between linear and binary search.

2.2. Program similarity

We now turn our attention to the problem of assessing the similarity between two programs, A and B. As discussed previously, different factors dictate the way programs may compare with each other, beyond their functional equivalence. These are: (1) programs' structure, (2) statements' order, (3) control structures used, and (4) data structures used.

We build on the approach taken by the LAV program similarity evaluation [6] for our similarity analysis. The LAV algorithm analyzes similarity between programs using their CFG representation, according to the information stored in the CFG's nodes (*i.e.,* sequences of LLVM instructions). To calculate the similarity between two given CFGs, the algorithm uses the neighbor matching method [7]. The neighbor matching method defines similarity for a CFG, $G = \langle V, E \rangle$, based on the similarity scores of its individual nodes. Two nodes, $i \in V_A$ and $j \in V_B$, abstracted from programs A and B respectively, are defined as similar, if the neighbors of i can be matched to neighbors of j. Therefore, the neighbor matching method relies in the assignment problem. The goal of the assignment problem is to find a matching of elements of CFGs G_A to G_B with the highest weight. A *matching* of nodes

for programs A and B is a set of pairs $M = \{\langle i, j\rangle \mid i \in V_A, j \in V_B\}$ such that no element of one set (say V_A) is paired with more than one element of the other set (V_B). For the matching M, we have two enumeration functions $f : \{1, 2, ..., k\} \to V_A$ and $g : \{1, 2, ..., k\} \to V_B$, such that $M = \{\langle f(l), g(l)\rangle \mid l = 1, 2, ..., k\}$ where $k = |M|$. The similarity of nodes i and j is defined as the average of the similarity of the matched in-nodes (*i.e.,* in-neighbors) and the matched out-nodes (*i.e.,* out-neighbors) for i and j, as in Equation (1),

$$x_{ij}^{k+1} \leftarrow \frac{S_{in}^{k+1}(i,j) + S_{out}^{k+1}(i,j)}{2} \tag{1}$$

$$S_{in}^{k+1}(i,j) \leftarrow \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x_{f_{ij}^{in}(l) g_{ij}^{in}(l)}^{k} \tag{2}$$

$$S_{out}^{k+1}(i,j) \leftarrow \frac{1}{m_{out}} \sum_{l=1}^{n_{in}} x_{f_{ij}^{out}(l) g_{ij}^{out}(l)}^{k} \tag{3}$$

where $m_{in} = max(id(i), id(j))$, $m_{out} = max(od(i), od(j))$ with $id(i)$ the in-degree of node i, and $od(i)$ the out-degree of node i. The functions f_{ij}^{in} and g_{ij}^{in} are the enumeration functions of the optimal matching of in-neighbors for nodes i and j with weight function $w(a, b) = x_{ab}^{k}$. Analogously f_{ij}^{out} and g_{ij}^{out} are the enumeration functions of the optimal matchings of out-neighbors for nodes i and j. The algorithm terminates the comparison between the CFGs when $max_{ij}|x_{ij}^{k} - x_{ij}^{k-1}| < \epsilon$, or after reaching a given number of iterations.

The similarity of the CFGs (*i.e.,* $S_{in}(i, j)$ and $S_{out}(i, j)$) is computed as the weight of the optimal matching of nodes divided by the number of matched nodes [7].

The original neighbor matching method of LAV [7] analyzes the graphs based only on their topology. Our approach extends this by annotating nodes with valuable information related to the program (LLVM instructions) extracted from the CFG. Furthermore, we use the edit distance between the sequences of instructions of every pair of nodes $i \in V_A$ and $j \in V_B$ [6] to calculate their similarity. The edit distance $d(i, j)$ is the minimal number of insertions, deletions, and substitutions needed to transform one sequence of characters into another. In LAV, the cost of insertion and deletion of an instruction is defined to be 1. The substitution cost depends on whether the instruction is a function call. If the instruction is a function call to different functions, then the substitution cost is 1. If the instruction is a call to the same function, then the cost is 0. If the instruction is not a function call, then the cost is 1, unless the instructions in both nodes are exactly the same. Finally, the similarity of the sequences of instructions of the nodes $i \in V_A$ and $j \in V_B$ is defined as $y_{ij} = 1 - d(i, j)/max(|i|, |j|)$, where $|i|$ is the number of instructions in the sequence [6]. The update rule is modified to calculate the similarity of nodes taking into account nodes' sequences of instructions, as in Equation (4).

$$x_{ij}^{k+1} \leftarrow \sqrt{y_{ij} \cdot \frac{S_{in}^{k+1}(i,j) + S_{out}^{k+1}(i,j)}{2}} \tag{4}$$

We observe that the similarity evaluation described by LAV presents problems when we compare two programs, A and B, in the case program A is contained in program B. In such case, the similarity value of the programs will be unusually high, disregarding the additional instructions from the container program B. This anomaly exacerbates whenever the size of program B is much larger than size of program A. The reason for this anomaly is that the neighbor matching method is designed to determine subgraph isomorphism. However, this property is not necessarily desired when analyzing programs. For instance, an automated grading system using this approach could be fooled by an empty program as it is contained by every other program. Therefore, to strengthen the neighbor matching method for similarity analysis, we improve the algorithm by taking into account the topological context of the graphs. Additionally, we normalize the output of the node similarity algorithm based on the maximum number of nodes between the CFGs of the two programs under analysis.

To counter the aforementioned problem and to take into account information about the graphs' topology, we modify the similarity metric between nodes, so it also contains information about the structure of the nodes' neighborhood (a level beyond of the current approach). Therefore, given two CFGs G_A and G_B, we compute a similarity matrix Z, where the position z_{ij} indicates the similarity between the nodes $i \in V_A$ and $j \in V_B$. The similarity index z_{ij} is calculated based on the Local Relative Entropy (LRE) index [8]. In contrast to the neighbor matching method described previously, our approach considers the similarity based only on the neighborhood structure –that is, comparing the degrees of a node's neighborhood without considering the value of the nodes itself.

To calculate the LRE we need to: (1) Find local networks $L_i(N, D)$, and their probability sets $P(i)$. (2) Calculate the relative entropy between all node pairs based on the probability sets for each node. (3) Calculate the similarity of each pair of nodes based on their relative entropy [8]. The local network of a node i is defined as $L_i(N, D)$ where N is the set of nodes in the local network, composed of the node i and its neighbors, and D the set of degrees of N. The total degree of a local network is defined by Equation (5), where $D(k)$ is the degree of the k neighbor of node i.

$$D_L(i) = \sum_{k=1}^{|D|} D(k) \tag{5}$$

The probability set $P(i)$ for node i describes the ratio between the degree of a node k in the local network and the total degree of the local network of node i. The probability set must have the same magnitude for all nodes. All probability sets have a fixed size equal to the largest degree in the graph (D_{max}) and the node itself, $|P(i)| = D_{max} + 1$, as in Equation (6).

$$P(i) = \{p(i,1), p(i,2), ..., p(i,k), ..., p(i, D_{max}+1)\} \tag{6}$$

where the probability for node k relative to node i is

$$p(i,k) = \begin{cases} \dfrac{D(k)}{D_L(i)}, & k \leq degree(i)+1 \\ 0, & k > degree(i)+1 \end{cases} \tag{7}$$

To compute LRE, the probability sets are sorted in decreasing order ($P'(i) = sorted(P(i), dec)$). Now, the relative entropy between two nodes is defined as in Equation (8).

$$D_{KL}(P^{'}(i)||P^{'}(j)) = \sum_{k=1}^{min(|D_i|,|D_j|)+1} p^{'}(i,k) ln \frac{p^{'}(i,k)}{p^{'}(j,k)} \tag{8}$$

Based on the relative entropy for each pair of nodes, the relevance matrix R is created, such that $r_{ij} = D_{KL}(P^{'}(i)||P^{'}(j)) + D_{KL}(P^{'}(j)||P^{'}(i))$. Finally, the LRE similarity score is defined in Equation (9).

$$z_{ij} = 1 - \frac{r_{ij}}{max(R)} \tag{9}$$

Even though, LRE is described for nodes within the same graph, we used it for calculating similarity between nodes across different graphs. This is part of our contribution to the algorithm to calculate the LRE and similarity for pairs of nodes $i \in V_A$ and $j \in V_B$.

Once we compute the Z matrix, we combine it with the edit distance stored in the Y matrix, such that $y_{ij} := \dfrac{y_{ij} + z_{ij}}{2}$, the average of the edit distance score and the LRE score. Finally, the similarity of the graphs corresponds to the weight of the optimal matching of nodes divided by the number of nodes of the CFG with the largest number of nodes.

Note that using our approach, the similarity of a program compared with itself is not 1, but rather is the highest value of the comparison with all the other programs in the analysis. Values closer to the comparison of a program with itself, mean that the programs are more similar.

3. Evaluation

We evaluate our program similitude analysis in two stages. First, we validate the effectiveness of our approach using algorithms from known domains. These algorithms are well understood so that their semantic behavior is known. Additionally, the programs used are complex enough so that all language features are evaluated. Second, use a new corpus for program similitude analysis, to evaluate our algorithm on larger multi-function codebases. The programs in the corpus are divided in 5 domains, including implementations that are alike and that are structurally different –that is, differ further than small deltas, but still provide the same result to a problem.

3.1. Evaluation Environment

To execute all the evaluation scenarios we use an Intel core i5-5257U processor with 8GB RAM running Ubuntu 18.04.2 LTS. Our evaluation uses version 3.3 of LLVM and C++11. All data used, and the full evaluation results are available in our online appendix https://flaglab.github.io/SimCorp/web/sqamia2024.html, here we focus the attention to the evaluation of our corpus.

3.2. Data Corpus

Our corpus consists of 566 different C++ programs extracted from the Codeforces competitive programming online judge. The extracted programs are solution submissions to five problems (domains). The problems used present different implementation characteristics, ranging from simple straightforward implementations (the difficulty level of the problems is given by their accompanying letter starting with A as the simplest problem), to implementations using multiple functions, requiring to manage complex data structures (*e.g.,* DSU) or advanced algorithms (*e.g.,* dynamic programming (dp), or computational geometry). For each of the problems we extracted up to 50 submissions from the categories: (1) (OK) complete all test cases, (2) (RUNTIME_ERROR) yield a runtime error, and (3) (WRONG_ANSWER) do not solve the problem properly. Table 1 shows the distribution of the data set classified by solution category, each containing the average Lines of Code (LOC) per submission.

Table 1
Corpus of Codeforces programs

Problem	Domain	OK	RUNTIME_ERROR	WRONG_ANSWER
558B	implementation	25 (avg. 28LOC)	44 (avg. 38LOC)	50 (avg. 26LOC)
922E	dp	12 (avg. 27LOC)	47 (avg. 53LOC)	49 (avg. 26LOC)
1142C	geometry	48 (avg. 34LOC)	25 (avg. 130LOC)	46 (avg. 31LOC)
1579A	math, strings	50 (avg. 17LOC)	21 (avg. 35LOC)	41 (avg. 19LOC)
1553G	brute force, constructive algorithms, dsu, hashing, number theory	46 (avg. 55LOC)	12 (avg. 57LOC)	50 (avg. 51LOC)

An important characteristic of the data set is that all OK solutions for a problem are assured to provide the same output to the same input. However, there is a wide disparity on the submissions for the other two categories, RUNTIME_ERROR and WRONG_ANSWER. In these cases, the submitted solutions may variate from empty programs, to programs close to a solution, to programs that solve completely different problems.

3.3. Experiment Design

The similarity evaluation of the programs in our corpus first computes a similarity matrix containing the similarity score for every pair of programs compared, as explained in Section 2.2. We use Principal Component Analysis (PCA) [9] to reduce the dimension of the matrix to two principal components, and plot these components using the PCA index. Furthermore, we use the silhouette coefficient [10] on the

similarity matrix to evaluate the cohesion and separation of the clusters per problem. The silhouette score is bounded between -1 and 1, similar programs have a score close to 1, overlapped clusters have a score close to 0, and dissimilar programs have a negative score.

We evaluate our algorithm using the LAV similarity analysis [6] as a baseline. We take this baseline for our experiments, as this constitutes the state-of-the-art analysis for node-based similarity, which is closest to our approach. Here we focus on the comparison of functionally equivalent programs that satisfy problems' conditions (OK), which we use to identify different implementation techniques for a specific problem.

Note that all programs in the corpus have a common behavior (*e.g.,* I/O instructions), and therefore have a positive similarity score. Furthermore, as all the submissions that solve a problem (*i.e.,* the OK category) have the same black-box behavior, we expect all solutions to a problem to be clustered. However, while the solutions to a problem behave the same, the algorithms used can have different implementations; therefore, we also expect to find sub-clusters for each problem.

3.4. Results

Our evaluation computes the similarity of the different problems from three perspectives. First, we generate the similarity matrix using the LAV method (labeled ORIGINAL in the figures). Second, we use our node similarity method to compare the submissions (labeled ORIGINAL-NODE-SYM). Third, we normalize the node similarity as described in Section 2.2 (labeled NORMALIZED-NODE-SYM).

When analyzing correct solutions (OK) to the problems, Figure 4a shows some clustering for each problem (identified by shape and marker's color in the figure). The silhouette score is 0.234, but still presents overlapping and scattering between problems 1579A, 922E, and 1553G. Using our algorithm, Figure 4b shows a better clustering, with a silhouette score of 0.204, and a $3.57\times$ improvement over the evaluation of all problems. This suggests that, as problems have behavior equivalence, our approach significantly improves the similarity metric of the evaluated programs.

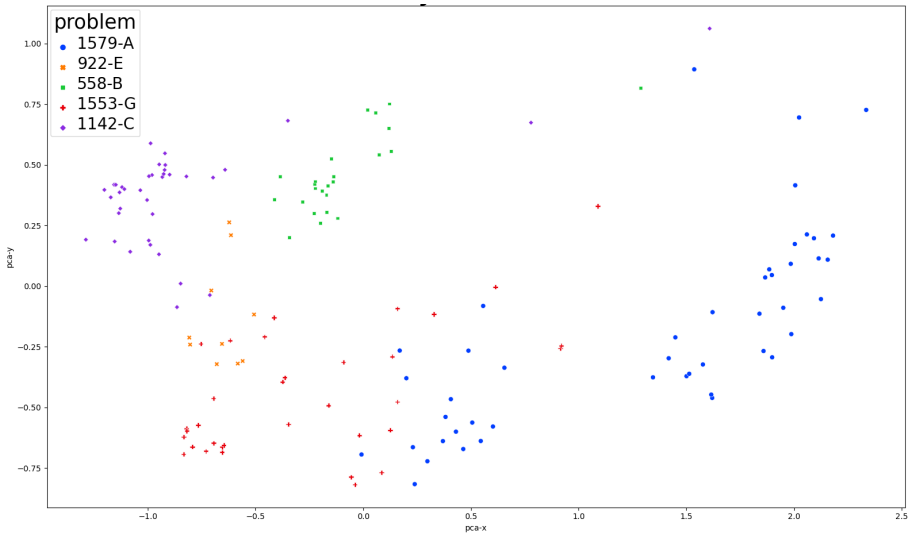
Note program scattering can be attributed to specific algorithms. There might be different solutions for a given problem, therefore these solutions should not be as similar to each other, as other solutions with the same algorithmic principle.

Figure 5a shows the OK submissions for problem 558B using the ORIGINAL method, with two clusters, showing two distinctive solutions to this problem. From the figure we see a tight cluster represented by the solutions using a circle, and a more scattered clustered represented by the solutions using a $\times$. This explains the low silhouette score of 0.276. Figure 5b shows our NORMALIZED method evaluating the same problem. Here we too obtain two distinctive clusters. In this figure we observe that the clusters are less scattered, explained by a silhouette score of 0.483. As there are two common solution patterns for the problem, we confirm our algorithm is effective in finding similar and dissimilar programs for particular problems with common black-box behavior.

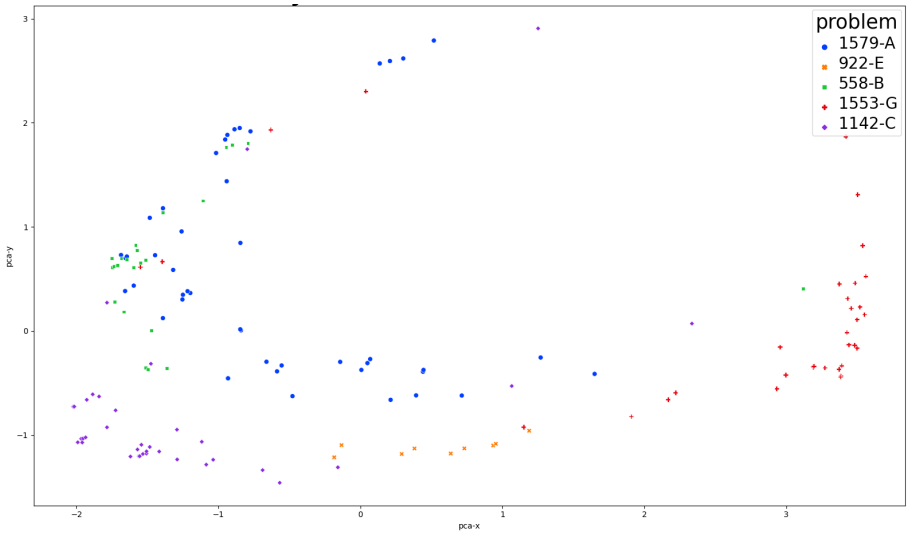
We use our approach to evaluate submission types for all the problems in the corpus. Table 2 presents the silhouette scores for all the programs in our corpus, with the best scores in bold. It is important to note, that the silhouette score for a specific problem, represents the ability of the algorithm to detect different implementations for the same problem. Not all problems contain different implementations within the same type of submission. Identifying such property is valuable for applications domains like diversity or N-version programming.

3.5. Discussion

From the results we conclude our approach is appropriate to: (1) Identify features common to different problems, for example, in the case of SimCorp, in which we identify similarities across all analyzed submissions in the way the input and output of the problem are processed (independent of the specific instructions used). (2) Detect differences in the algorithms behind different programs, even when they are functionally equivalent. This is shown in the clusters for the correct (OK) solutions to a problem in our corpus.



(a) ORIGINAL algorithm for OK submissions



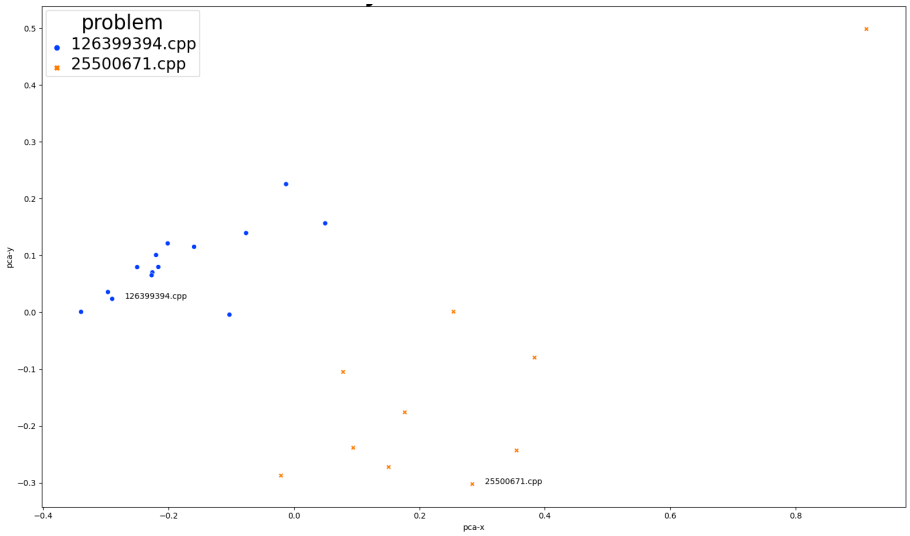
(b) NORMALIZED-NODE-SYM algorithm for OK submissions

Figure 4: PCA-x and PCA-y components for all problems

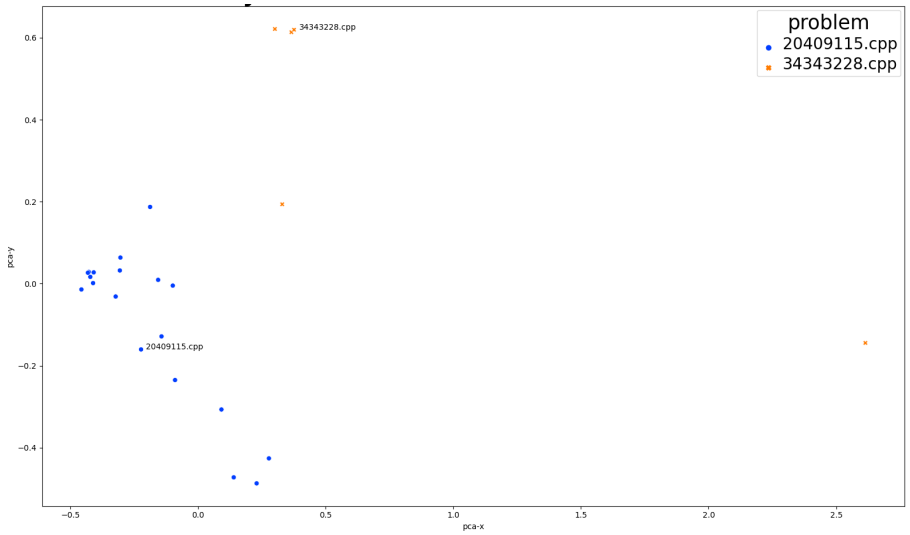
The performance of our algorithm, measured using the silhouette score, is similar to the performance of the baseline, *i.e.,* the LAV algorithm. Our evaluation shows that the use of our node similarity definition has a slight improvement over the baseline in most cases. Moreover, when analyzing specific algorithm pairs identified as similar/disimilar, we respectively note a great resemblance/disparity in the specific implementations. This is beneficial as a notion of diversity between algorithms. However, the validation shows that in some cases using node normalization is detrimental to the performance. We note that, as programs differ but have an important feature in common (*e.g.,* input/output processing), the algorithm will detect these as similar, decreasing the silhouette score.

4. Related Work

Existing approaches for the semantic program analysis often focus on behavior equivalence between programs (*i.e.,* programs' outputs are equivalent for the same inputs). Other approaches strive to detect semantic similarities for programs that differentiate on small deltas. We put our proposed algorithm to analyze similarity in perspective of these existing approaches, divided in two categories: analysis for semantic equivalence, and semantic code clone detection.



(a) ORIGINAL algorithm for problem 558B OK



(b) NORMALIZED-NODE-SYM algorithm for problem 558B OK

Figure 5: PCA-x and PCA-y component evaluation

4.1. Semantic Equivalence of Programs

Control Flow Graph CFGs are among the most used structures to analyze the structure and behavior of programs, as they allow developers to explore all the execution paths of a program. CFGs are normally analyzed using subgraph isomorphisms to detect similar code fragments as related graph sections. Such techniques are problematic for similarity comparison, for two main reasons [4, 5]. First, subgraph isomorphism algorithms are time- and resource-heavy, restricting the complexity of the problems that can be analyzed. Second, this technique is computable only for specific cases, restricting the types of programs to analyze. We use a different method, where nodes' local information determines their similarity, this enables us to assess similarity between any type of programs.

LAV [6] uses CFG neighbor and node similarity to assess student assignments for introductory programming courses, when compared with the solutions provided by the course's instructors. The neighbor matching method used in LAV is the base for the implementation of our approach. This method calculates the similarity between two nodes taking into account their in and out nodes, following Equations 1-4 in Section 2.2. While LAV enhances code similarity using nodes' content, which corresponds to a linear code sequence to calculate graph nodes similarity, we use the LRE index and normalize node similarity between programs to weight the relevance of each neighbor for the nodes

Table 2
Silhouette score for Codeforces programs

Problem	Submission	ORIGINAL	ORIGINAL-NODE-SYM	NORMALIZED-NODE-SYM
All	**All**	0.149	**0.157**	0.057
All	**OK**	0.234	**0.240**	0.204
All	**WRONG ANSWER**	0.207	**0.210**	0.101
All	**RUNTIME ERROR**	0.050	**0.067**	0.002
558B	**OK**	0.276	0.298	**0.483**
558B	**WRONG ANSWER**	0.403	**0.446**	0.407
558B	**RUNTIME ERROR**	0.350	**0.373**	0.342
922E	**OK**	0.320	0.304	**0.419**
922E	**WRONG ANSWER**	0.525	0.522	**0.683**
922E	**RUNTIME ERROR**	0.555	0.418	**0.622**
1142C	**OK**	0.331	0.541	**0.694**
1142C	**WRONG ANSWER**	0.561	0.555	**0.620**
1142C	**RUNTIME ERROR**	0.382	0.371	**0.426**
1579A	**OK**	0.480	**0.529**	0.442
1579A	**WRONG ANSWER**	0.508	**0.530**	0.482
1579A	**RUNTIME ERROR**	0.322	0.337	**0.351**
1553G	**OK**	0.284	0.307	**0.640**
1553G	**WRONG ANSWER**	0.579	**0.644**	0.594
1553G	**RUNTIME ERROR**	0.506	0.519	**0.620**

under comparison, with better results in many cases, and a defined metric to differentiate algorithms.

Symbolic execution Symbolic execution methods can be used to focus the similarity analysis between programs on their behavior, rather than their structure [11]. This method performs analysis on programs' structure, represented as a CFG. The result of the symbolic step is a set of variable stores and path conditions. While using symbolic execution methods can detect similarity between programs with similar behavior, in generally they are unable to prove equivalence [12]. Moreover, if the programs obtain the same behavior using different techniques, the symbolic method will not reach close similarity. Our approach addresses this shortcoming.

Abstract interpretation In response to the problems presented in symbolic execution methods, abstract interpretations methods aim to increase the precision of existing proposals that suffer of under-approximation. Partush and Yahav [12] present speculative correlating semantics to allow the bounded representation of program difference. The technique uses SCORE, that given an abstracted pair of programs, finds the state of minimal difference between them. Similar to our approach, the use of program abstractions in SCORE allows the comparison of programs with more significant differences.

Differential symbolic execution (DSE) Finally, DSE methods encompass the analysis of programs based on a preliminary *differential program analysis* [13], exploring only the parts of the programs that have changed. DSE performs a method level comparison of programs, taking the symbolic summaries of each method to check functional or path equivalence [14]. DSE presents an improvement over symbolic execution methods using the path equivalence, which takes into account both the final behavior of programs, and the way programs are structured for a more faithful comparison between programs. ARDiff [2] improves the assessment of program equivalence to improve results using refinement abstractions to optimize the symbolic summaries needed to prove equivalence. As with SCORE, this approach is effective in evaluating small differences between two programs.

Trostanetski et al. [15] extend DSE with a modular execution improving the search of procedures that have already been explored (an alternative method to refinement abstraction). The contribution of this approach is the possibility to both prove and disprove equivalences between programs. This characteristic is similar to our proposed metric, we can state that programs with a lower PSA metric are different.

4.2. Semantic Code Clone Detection

A vast body of work exists on the topic of code clone detection [16, 17]. However, of the 54 tools in the most recent survey [16], only 9 tools report the detection of Type-4 or semantic clones. As the datasets to evaluate these tools are not available, we discuss the algorithm and results of the main existing tools in perspective of our approach.

CCSharp [5] specializes in detecting Type-4 clones using Program Dependence Graphs (PDGs). CCSharp is based on subgraph isomorphism of synthesized PDGs to improve the algorithm's performance. Program similarity in CCSharp is based on string and numerical similarity. String similarity evaluates the distance between input and output parameters, then evaluates the similarity of function names. Numerical similarity takes the characteristic vectors extracted from the PDG (*i.e.,* the meaning of nodes) and calculates their euclidean distance. Based on graph isomorphism, CCSharp may not be able to cope with programs with strong syntactic differences, as the ones we are targeting. Similar to our approach, Nasirloo and Azimzadeh [18] use normalization to refine the clone analysis over PDGs.

Sheneamer et al. [19] present a machine learning based method to detect Type-4 clones on obfuscated code. This approach is based on the features extracted from a Bytecode Dependency Graph (BDG), and features extracted from an AST or PDG. This method is similar to other CFG methods discussed before, in that CFGs are the base structure to extract information about the programs. In this case we also require example programs to use as the training set of the machine learning algorithm, to further related observed features as semantic clones during testing. The amount of data required to use machine learning techniques makes such approaches inappropriate for the comparison of diverse programs.

5. Conclusion and Future Work

This paper presents a new approach to measure similitude between different programs. Unlike existing approaches that focus on the equivalence of programs based on their behavior, our approach analyzes similitude from the perspective of programs' structure and complexity. A particular contribution of our approach is that it is agnostic to the specific program under analysis. That is, we do not require the programs to be similar beforehand (*i.e.,* are delta variations of each other). In fact, we strive to analyze programs that are different in kind, which is of use to evaluate how diverse a code base is, or how diverse are different implementations of a system functionality.

Together with our approach, we posit a new corpus for evaluating similitude between different programs. Our corpus contains 566 different C++ programs extracted from submitted solutions to competitive programming problems from the Codeforces online judge. In the evaluation, we validate that our approach is effective in detecting similarity between programs at par with existing approaches. Additionally, our approach is effective in identifying diverse families of programs for a same problem with different algorithmic solutions, outperforming the precision of existing solutions by up to 77%.

We identify two avenues of future work. First, we will continue to improve the precision. One possible improvement would be to provide the CFG nodes with more information from the LLVM instructions. Second, we want to apply our approach as a new metric for technical debt for development companies, and as a recommender for existing functionality to implement in new products.

References

[1] V. Nallur, E. O'Toole, N. Cardozo, S. Clarke, Algorithm Diversity: A Mechanism for Distributive Justice in a Socio-Technical MAS, in: Proceedings of the International Conference on Autonomous Agents & Multiagent Systems, AAMAS'16, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2016, pp. 420–428.

[2] S. Badihi, F. Akinotcho, Y. Li, J. Rubin, ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code, in: Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'20, 2020, pp. 13–24. doi:10.1145/3368089.3409757.

[3] F. E. Allen, Control flow analysis, ACM Sigplan Notices 5 (1970) 1–19.

[4] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings Eighth Working Conference on Reverse Engineering, 2001, pp. 301–309. doi:10.1109/WCRE.2001.957835.

[5] M. Wang, P. Wang, Y. Xu, CCSharp: An efficient three-phase code clone detector using modified PDGs, in: Asia-Pacific Software Engineering Conference, APSEC'17, IEEE, 2017, pp. 100–109.

[6] M. Vujošević-Janičić, M. Nikolić, D. Tošić, V. Kuncak, Software verification and graph similarity for automated evaluation of students' assignments, Information and Software Technology 55 (2013) 1004–1016.

[7] M. Nikolic, Measuring similarity of graphs and their nodes by neighbor matching, CoRR abs/1009.5290 (2010). URL: http://arxiv.org/abs/1009.5290. arXiv:1009.5290.

[8] Q. Zhang, M. Li, Y. Deng, S. Mahadevan, Measure the similarity of nodes in the complex networks, CoRR abs/1502.00780 (2015). URL: http://arxiv.org/abs/1502.00780. arXiv:1502.00780.

[9] I. T. Jolliffe, Principal Component Analysis, Springer-Verlag, 1986.

[10] P. J. Rousseeuw, Silhouettes: A graphical aid to the interpretation and validation of cluster analysis, Journal of Computational and Applied Mathematics 20 (1987) 53–65. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257. doi:https://doi.org/10.1016/0377-0427(87)90125-7.

[11] S. M. Arifi, R. B. Abbou, A. Zahi, A New Similarity-based Method for Assessing Programming Assignments using Symbolic Execution, International Journal of Applied Engineering Research 13 (2018) 1963–1981.

[12] N. Partush, E. Yahav, Abstract semantic differencing via speculative correlation, in: Proceedings of the International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'14, 2014, pp. 811–828. doi:10.1145/2660193.2660245.

[13] J. Winstead, D. Evans, Towards differential program analysis, in: ICSE Workshop on Dynamic Analysis, 2003.

[14] S. Person, M. B. Dwyer, S. Elbaum, C. S. Păsăreanu, Differential symbolic execution, in: Proceedings of the International Symposium on Foundations of Software Engineering, FSE'08, ACM, 2008, pp. 226–237. doi:10.1145/1453101.1453131.

[15] A. Trostanetski, O. Grumberg, D. Kroening, Modular demand-driven analysis of semantic difference for program versions, in: International Static Analysis Symposium, Springer, 2017, pp. 405–427.

[16] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, B. Maqbool, A Systematic Review on Code Clone Detection, IEEE Access 7 (2019) 86121–86144. doi:10.1109/ACCESS.2019.2918202.

[17] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, IEEE Transactions on Software Engineering 33 (2007) 577–591. doi:10.1109/TSE.2007.70725.

[18] H. Nasirloo, F. Azimzadeh, Semantic code clone detection using abstract memory states and program dependency graphs, in: International Conference on Web Research, ICWR'18, IEEE, 2018, pp. 19–27.

[19] A. Sheneamer, S. Roy, J. Kalita, A detection framework for semantic code clones and obfuscated code, Expert Systems with Applications 97 (2018) 405–420.