Detecting Source Code Similarity Using Compression

IVAN PRIBELA, GORDANA RAKIĆ and ZORAN BUDIMAC, University of Novi Sad

Different forms of plagiarism make fair assessment of student assignments more difficult. Source code plagiarisms pose a significant challenge especially for automated assessment systems aimed for students' programming solutions. Different automated assessment systems employ different text or source code similarity detection tools, and all of these tools have their advantages and disadvantages. In this paper we revitalize the idea of similarity detection based on string complexity and compression. We slightly adapt an existing, third party, approach, implement it and evaluate its potential on synthetically generated cases and on a small set of real student solutions. On synthetic cases we showed that average deviation (in absolute values) from the expected similarity is less than 1% (0.94%). On the real life examples of student programming solutions we compare our results with those of two established tools. The average difference is around 18.1% and 11.6%, while the average difference between those two tools is 10.8%. However, the results of all three tools follow the same trend. Finally, a deviation in some extent is expected as observed tools apply different approaches that are sensitive to other factors of similarities. Gained results additionally demonstrate open challenges in the field.

1. INTRODUCTION

Automatic assessment systems encounter significant problems when assessing students' solutions to programming assignments. Main difficulties arise from lack of academic discipline among the students. While the students are expected to submit their original solutions, sometimes this is not the case. The assignments that they are expected to solve are usually very similar, and there is a restricted set of varieties among the correct solutions. Therefore, plagiarism detection is always an open question in the assessment of students' work.

Multiple tools and approaches are available for plagiarism detection. They are based on source code similarity detection algorithms or code clone detection tools. Available approaches rely on lexical, syntactic, structural, or semantic information about the programming solution. Thus, algorithms analyze character streams, token streams, controls structures or dependencies in the observed code. Usually, they are applied to the source code or some of its intermediate representations in the form of an intermediate language, a tree or a graph. All of these similarity analysis approaches have their advantages and disadvantages. Their sensitivity to different categories of similarities and precision varies and therefore their applicability and usefulness depend on objectives of the analysis [Novak et al. 2019].

Although high similarity between two projects does not mean that one is plagiarized from the other, it is often a good indicator. The students that are submitting work of other students usually make

This work was partially supported by the Ministry of Education, Science, and Technological development, Republic of Serbia, through project no. OI 174023: "Intelligent techniques and their integration into wide-spectrum decision support".
Author's address: Ivan Pribela, Gordana Rakić, Zoran Budimac, University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: ivan.pribela@dmi.uns.ac.rs gordana.rakic@dmi.uns.ac.rs gordana.rakic@dmi.uns.ac.rs zoran.budimac@dmi.uns.ac.rs

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

CEUR-WS.org/Vol-2508/paper-pri.pdf

only small superficial adjustments. They often do not change the structure of the code but only rename variables and functions or reorder statements that they know will not change the program correctness.

In this paper we describe an approach to similarity analysis based on string complexity and compression. We have chosen this method because the initial study presented in [Chen et al. 2004] showed promising results and a fresh perspective on settled practices, but the system is not available anymore for use or extension.

Other approaches to measuring similarity and detecting plagiarism were concentrated around feature comparison and program structure matching. Early attempts are usually based on computing different software metrics and comparing the results. More recent tools usually use a form of greedy string tiling to compare two streams of tokens instead a summary indicator. Both types of approaches have their strengths and weaknesses with attribute counting systems performing better in the case of very small changes, and structure comparison systems exhibiting better performance when only a part of the code is copied.

In section 2, we describe the background and motivation for our work, that lays in the previous third-party solution [Chen et al. 2004] that introduced this approach and showed perspective but is not available anymore. Description of our approach and our contribution to the previous results can be found in the section 3. Results of our contribution, demonstrated on both artificial cases and real-life student solutions, are described in the section 4, followed by conclusions and the future work in section 5.

2. BACKGROUND AND RELATED WORK

Kolmogorov complexity $K(s)$ of a string s [Zvonkin and Levin 1970] is the length of the shortest computer program that with no given input produces the string s as the output. It can be considered as a measure of the amount of information contained in the string s.

The conditional Kolmogorov complexity $K(s|t)$ of two strings s and t [Zvonkin and Levin 1970] is the length of the shortest computer program that produces the string s as the output with the string t given as an input. It can be viewed as a measure of the amount of information contained in the string s that is not present in the string t.

The chain Kolmogorov complexity $K(s,t)$ of two strings s and t [Zvonkin and Levin 1970] is the length of the shortest computer program that produces the string s followed by the string t with the information on how to separate them.

The chain rule for Kolmogorov complexity [Li et al. 2008] states that $K(s,t) \leq K(s) + K(t|s) + O(log(K(s,t)))$ or $K(s,t) \approx K(s) + K(t|s)$. In other words, the length of the shortest computer program that produces the string s followed by the string t is approximately the same as the length of the shortest computer program that produces the string s plus the length of the shortest computer program that produces the string t having the string s as input.

The algorithmic mutual information for Kolmogorov complexity $I_K(s,t)$ is defined as $I_K(s,t) = K(s) - K(s|t)$ and represents the amount of information about the string s that is contained in the string t. This quantity is symmetric up to a logarithmic factor as shown in [Gács 1974] which gives us $I_K(s,t) \approx I_K(t,s)$.

Having all this in mind we get $K(s,t) \approx K(s) + K(t|s) \approx K(t) + K(s|t) \approx K(t,s)$.

If we take the absolute amount of mutual information in the strings s and t and divide it by total amount of information in both strings as in formula 1 we get a normalized measure of similarity between the strings s and t.

$$sim(s,t) = \frac{I_K(s,t)}{K(s,t)} \tag{1}$$

If the strings s and t are identical, this will be equal to one, if the two string are completely unrelated, the value will be zero. However sim function is not a properly defined metric function. On the other hand, the function $d_{sim}(s,t) = 1 - sim(s,t)$ satisfies all the necessary conditions for a metric up to a logarithmic error as shown in [Li et al. 2001].

Unfortunately, Kolmogorov complexity is not computable. It is not possible to compute even the lower bound [Chaitin 1974]. However, a reasonable upper bound can be expressed as the length of the compressed representation of the string as produced by a reasonable compression algorithm. This is the approach taken in a tool called SID [Chen et al. 2004] that employs algorithmic complexity to check for similarities in source code. As approximation for this value, a special compression algorithm called TokenCompress is used. However, this tool is no longer available for usage, which is one of the motivating factors behind our work.

3. DESCRIPTION OF THE APPROACH

Our tool is also based on measuring Kolmogorov complexity, or more accurately approximating it using compression similarly to the work done in [Chen et al. 2004].

A compressed version of the string can be viewed as a program written in a language executed by the corresponding decompressor. Interpreted by the decompressor, the output of such program is the original string. Also such program tends to be the shortest as the main goal of a compression algorithm is to produce the shortest possible sequence that decompresses into the original input.

As the original compression algorithm described in [Chen et al. 2004] is not available. We have chosen GNU gzip library to approximate $K(s)$. This library uses Lempel-Ziv coding (LZ77) [Ziv and Lempel 1977] and Huffman coding [Huffman 1952], and is considered the industry standard for general compression with a good balance of time and compression ratio. We define $|s|$ as the length in bytes of the compressed form of the string s when compressed using the mentioned library. As stated earlier, the value $|s|$ is a reasonable approximation for $K(s)$. Similarly, $K(s,t)$ is approximated by $|st|$ and $K(s|t)$ is approximated using the formula $K(s|t) \approx K(t,s) - K(t)$ based on the chain rule for Kolmogorov complexity.

Having this in mind, the final similarity is calculated using the formula 2.

$$sim(s,t) = \frac{|s| + |t| - |ts|}{|st|} \tag{2}$$

In order to calculate the percentage of the common information we have to further interpret this similarity metric, as was done in [Chen et al. 2004]. If we assume that the two strings s and t are approximately the same length as is common among student solutions, then $sim(s,t) = \frac{p}{2-p}$ where p is the percentage of the information common in both strings [Chen et al. 2002]. This gives us the formula 3 to calculate p.

$$p = \frac{2sim(s,t)}{1 + sim(s,t)} \tag{3}$$

However, when students try to hide similarities, one of the methods they employ is adding redundant or unused code thus increasing the total length of the copy. In such cases, when compared with each other, the copy and the original can have considerable length difference.

To combat this we have proposed two new measures in addition to the one already described in [Chen et al. 2004]. These measures represent the percentage of string s that is present in t and vice versa, and are given with formulas 4 and 5.

$$sim_s(s,t) = \frac{|s| + |t| - |ts|}{|s|} \tag{4}$$

$$sim_t(s,t) = \frac{|s| + |t| - |ts|}{|t|} \tag{5}$$

These two measures are not always symmetrical. For example, if we have a string s representing the source code form one student solution, and string t generated from s by adding two times more unused code, then $sim(s,t)$ will not give a strong indication about possible plagiarism as the percentage will be around 50%. However, $sim_s(s,t)$ will give the percentage around 100% immediately stating that one string is present in the other in its entirety. Similar situation can be observed when a part of one solution is copied from the other without adding or modifying it much.

4. RESULTS

To explore the viability of our proposed approach, we tested the performance of our tool both in isolation and in comparison with two other similarity detection tools.

4.1 Synthetic cases

In the first step we tested our tool using synthetic benchmarks. We have performed 22 separate tests by supplying two randomly generated strings as input to our tool and recording the calculated similarities. Each test was repeated 1000 times and the average similarity was recorded. In each test we generated three sequences of random data of predefined lengths for that test: unique content for the first input string, unique content for the second input string and content common for both strings. The two inputs supplied to the tool were generated by combining the content from the appropriate random sequences.

The results of the tests are given in Table I. The value $unq(s)$ represents the total amount of data unique to the string s, while $unq(t)$ represent the same for the string t. The value $cmn(s,t)$ represents the total amount of data common in both strings. The next two groups of values represent the expected and calculated percentages of similarity, while the last three columns contain percentage of difference between expected and calculated values. The value p represent the percentage of data common to both strings. The value p_s represent the percentage of data in the string s that is also present in the string t. The value p_t represent the percentage of data in the string t that is also present in the string s.

As can be seen from the data, the similarity calculated by our tool is very close to the real values. More precisely, values of all three functions differ from the expected value within an interval between 0.04% and 2.69%. It can be observed that the calculated values differ from the expected ones on average 0.94% for both strings, 1.1% for the amount of data s has in common with t, and 0.85% for the amount of data t has in common with s, while standard deviation of differences is 0.59%, 0.69% and 0.57% respectively. The retrieved values are consistently slightly lower because of the imperfections of the compression algorithm used to approximate the Kolmogorov complexity. With a compression algorithm closer to the perfect compression these values would probably be closer to the expected.

4.2 Real world examples

In the second step we have compared the results from our tool with SIM [Grune 2006] and JPlag [Prechelt et al. 2002]. SIM tests lexical similarity in natural language texts and in programs written in various programming languages. Its main purpose is to detect duplicated code in large software projects and to detect plagiarism between different software projects. JPlag is a system that finds similarities among multiple sets of source code files. It is aimed at detecting software plagiarism and as such was used by expert witnesses in several intellectual property cases.

Table I. Similarity percentages for the synthetic benchmarks

Amount of data			Expected similarity			Calculated similarity			Difference in similarities		
$unq(s)$	$unq(t)$	$cmn(s,t)$	p	p_s	p_t	p	p_s	p_t	p	p_s	p_t
0	0	3000	1.00000	1.00000	1.00000	0.97311	0.97311	0.97311	0.0269	0.0269	0.0269
3000	3000	0	0.00000	0.00000	0.00000	0.00100	0.00100	0.00100	0.0010	0.0010	0.0010
3000	3000	3000	0.50000	0.50000	0.50000	0.48801	0.48801	0.48801	0.0120	0.0120	0.0120
3000	3000	6000	0.66667	0.66667	0.66667	0.65475	0.65475	0.65475	0.0119	0.0119	0.0119
6000	6000	3000	0.33333	0.33333	0.33333	0.32538	0.32538	0.32538	0.0080	0.0080	0.0080
3000	3000	9000	0.75000	0.75000	0.75000	0.73850	0.73850	0.73850	0.0115	0.0115	0.0115
9000	9000	3000	0.25000	0.25000	0.25000	0.24302	0.24302	0.24302	0.0070	0.0070	0.0070
0	3000	3000	0.66667	1.00000	0.50000	0.64981	0.97440	0.48744	0.0169	0.0256	0.0126
0	3000	6000	0.80000	1.00000	0.66667	0.78571	0.98204	0.65481	0.0143	0.0180	0.0119
0	3000	9000	0.85714	1.00000	0.75000	0.84400	0.98462	0.73852	0.0131	0.0154	0.0115
3000	6000	0	0.00000	0.00000	0.00000	0.00067	0.00100	0.00050	0.0007	0.0010	0.0005
3000	6000	3000	0.40000	0.50000	0.33333	0.39051	0.48808	0.32544	0.0095	0.0119	0.0079
3000	6000	6000	0.57143	0.66667	0.50000	0.56127	0.65479	0.49113	0.0102	0.0119	0.0089
3000	6000	9000	0.66667	0.75000	0.60000	0.65587	0.73783	0.59029	0.0108	0.0122	0.0097
3000	9000	0	0.00000	0.00000	0.00000	0.00050	0.00100	0.00033	0.0005	0.0010	0.0003
3000	9000	3000	0.33333	0.50000	0.25000	0.32549	0.48816	0.24414	0.0078	0.0118	0.0059
3000	9000	6000	0.50000	0.66667	0.40000	0.48995	0.65321	0.39198	0.0101	0.0135	0.0080
3000	9000	9000	0.60000	0.75000	0.50000	0.59057	0.73830	0.49210	0.0094	0.0117	0.0079
6000	9000	0	0.00000	0.00000	0.00000	0.00040	0.00050	0.00033	0.0004	0.0005	0.0003
6000	9000	3000	0.28571	0.33333	0.25000	0.27714	0.32331	0.24250	0.0086	0.0100	0.0075
6000	9000	6000	0.44444	0.50000	0.40000	0.43589	0.49037	0.39231	0.0086	0.0096	0.0077
6000	9000	9000	0.54545	0.60000	0.50000	0.53696	0.59073	0.49216	0.0085	0.0093	0.0078

As sample data, we have selected a few student solutions to practical exercises. All solutions belong to a single group of second year students working independently on the same assignment. The students were not sharing code among them. Along with student solutions one produced by us was added. We have randomly chosen one of the student solutions and changed all variable names without changing the rest of the code. This was done to illustrate a common way students try to mask plagiarism.

The results of this test are illustrated by charts in figures 1 and 2. From the starting 14 solutions, all possible pairs were generated. Figure 1 shows 182 points in total representing each pair twice, once for each component in the pair. For each of these components, the figure shows the percentage of the code that is also present in the other component of the pair, as reported by all three tools. Figure 2 contains the similarity values reported for the pair as a whole. As SIM does not report this measure, it is not included in the graph on this figure.

The results from our tool are comparable with the results from SIM and JPlag. As can be seen from the charts, percentages are roughly consistent within a margin, both for a single solution and for the whole pair.

There are a few instances in which each of the tools give a slightly different result from the others. The discrepancies between our tool and each of the other two are roughly the same as differences between them and all three tools are showing the same trend. Anyhow, in no situation one of the tools gives a strong indication of similarity while the other two are indicating the opposite.

However, our tool usually gives a slightly higher percentage of similarity. This is rooted in two fundamental reasons. Firstly, both SIM and JPlag have a concept of a minimum run length. This parameter dictates how many consecutive matches are needed to consider a part of the program code a potential copy. Our tool does not have such restriction stated explicitly. However, implicitly as a feature of standard compression algorithms, a match that is too small is disregarded on the basis that its compressed representation is longer that the uncompressed.

Fig. 1. Percentage of similarity within one solution in a pair

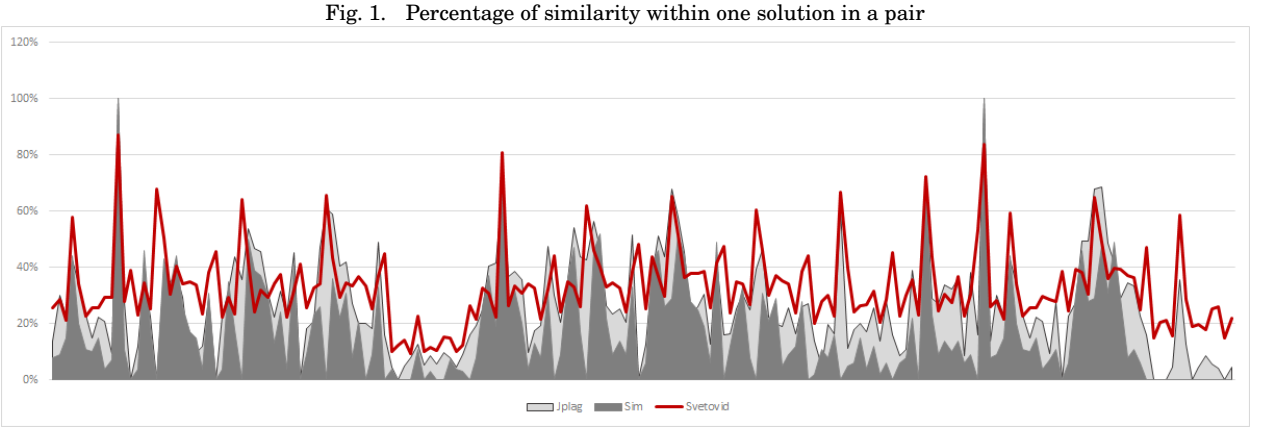
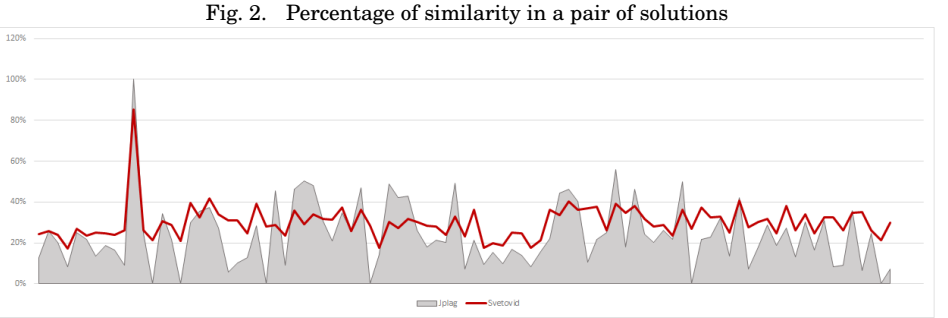


Fig. 2. Percentage of similarity in a pair of solutions



Secondly, our tool is basing the comparison on the level of characters, while SIM and JPlag are comparing token streams. This results in our tool detecting finer similarities that do not span whole tokens, like for example, between *Integer*, *LongInteger* and *ShortInteger* or by capturing a part of a token in front or behind the group detected by other tools.

This does not mean that either of the approaches is better that the other. It depends on the particular case and warrants further investigation.

5. CONCLUSIONS AND FUTURE WORK

In the area of automatic assessment of students' programming solutions, one of the open challenges is plagiarism detection. While there are numerous tools for source code similarity analysis and for code clone detection, there is still no tool fully suitable to be used for these purposes. Some of the factors that affect the usefulness of these tools are the applied approach and the used similarity algorithm. In this paper we explore possibilities of employing string complexity and compression in source code similarity measurement. We evaluate gained results on synthetic examples in comparison with the expected values based on the background theory, as well as on real-life student solutions in comparison with alternative tools.

In comparison with the expected values, we gain slightly lower similarity values. The cause of this is the imperfection of the chosen compression algorithm, and the fact that it implements lossless compression which considers the equality of string sequences and not actually their similarity. Hence, in

the future work we can experiment with alternative algorithms that are lossy and sacrifice a little accuracy to, for example, better capture attempts at renaming variables.

Results obtained from our tool, when compared to the similarity values acquired from competitive tools, show the same general trend, although our calculated similarity values tend to be slightly higher. The differences are caused by different observation unit employed (character- versus token-based observation). Experimentation in this direction may bring higher precision, but also lower detection capabilities.

Also, one of the important features in similarity analysis is the consideration of the base code, as student assignments usually include some predefined elements for the future solution. This is another significant direction for future work.

REFERENCES

Gregory J Chaitin. 1974. Information-theoretic limitations of formal systems. *Journal of the ACM (JACM)* 21, 3 (1974), 403–424.

Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. 2004. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory* 50, 7 (2004), 1545–1551.

Xin Chen, Ming Li, Brian Mckinnon, and Amit Seker. 2002. A theory of uncheatable program plagiarism detection and its practical implementation. *SID website at http://dna.cs.ucsb.edu/SID* (2002).

Peter Gács. 1974. On the symmetry of algorithmic information. In *Doklady Akademii Nauk*, Vol. 218. Russian Academy of Sciences, 1265–1267.

Dick Grune. 2006. The software and text similarity tester SIM. (2006).

David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

Ming Li, Jonathan H Badger, Xin Chen, Sam Kwong, Paul Kearney, and Haoyong Zhang. 2001. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics* 17, 2 (2001), 149–154.

Ming Li, Paul Vitányi, and others. 2008. *An introduction to Kolmogorov complexity and its applications*. Vol. 3. Springer.

Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Transactions on Computing Education (TOCE)* 19, 3 (2019), 27.

Lutz Prechelt, Guido Malpohl, Michael Philippsen, and others. 2002. Finding plagiarisms among a set of programs with JPlag. *J. UCS* 8, 11 (2002), 1016.

Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

Alexander K Zvonkin and Leonid A Levin. 1970. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys* 25, 6 (1970), 83.