SOFTWARE SYSTEM FOR SIMILARITY DETECTION IN THE PICOCOMPUTER ASSEMBLY PROGRAMS

Vojislav Tomašević[1],
Marko Mišić[1],
Violeta Tomašević[2]*

[1]School of Electrical Engineering,
Belgrade, Serbia

[2]Singidunum University,
Belgrade, Serbia

Abstract:

This paper tackles the problem of plagiarism in an academic environment with an emphasis on the detection of similarities between the source codes from student assignments in the programming courses. The detected similarity in these codes greatly helps a human expert to bring the final decision on which codes are plagiarised and to which extent. Since the manual comparison of the source codes is a tedious task, the system for automatic detection of similarities in the assembly programs written for the *picoComputer* architecture is envisioned and implemented. It relies on the application which first performs the scanning and tokenization of the source codes. The pair-wise similarity detection is carried out by the *Greedy String Tiling* algorithm upgraded with the hash-based *Karp-Rabin* modification. A convenient GUI is also provided for efficient communication for the users and the choice of necessary parameters. Two different approaches are pursued in the testing and evaluation of the system. The first test set consists of a starting program with several versions with intentional modifications to simulate plagiarism. The second test set represents a real workload which comprises 250 real source codes from the student assignments. In both cases, the system demonstrated good efficiency.

Keywords:

plagiarism, similarity detection, *picoComputer* architecture, tokenization, *RKR-GST algorithm.*

INTRODUCTION

Nowadays, information and knowledge are easily and widely accessible more than ever, either in paper form or in various electronic forms on the Internet. It is one of the reasons for increasing evidence of unethical conducting in producing different kinds of work (books, papers, source codes, etc.), partly or entirely, and presenting some others' work as own [1]. In order to fight against plagiarism, various tools are developed and exploited to detect the similarities of the submitted works, such as JPlag [2] and Moss [3] for source code similarity detection, or TurnitIn iThenticate [4] for text-based plagiarism.

Correspondence:
Violeta Tomašević

e-mail:
vitomasevic@singidunum.ac.rs

DOI: 10.15308/Sinteza-2022-246-253

In an academic environment, the problem of plagiarism is extremely important and sensitive. Such a practice directly and adversely affects the regularity of acquiring academic diplomas and positions. In the university teaching practice, there is an apparent need to check for the plagiarism of the students' final thesis, homework, assignments, projects, etc [5]. This paper deals with the problem of similarity detection of the students' source codes written to fulfil their assignments in the programming courses. There are several malpractices that students conduct in such cases. Some of them take parts of the code or the entire code from the online sources. However, the students not rarely tend to copy the works of their colleagues, as exact solutions of the appointed problems in student assignments cannot be always found online. Some of them perform lexical and structural modifications in the code intended to hide the plagiarism from the examiner. We had in mind all those scenarios during the development of this system.

It is worth mentioning that, in this context, a similarity of the two codes does not necessarily imply plagiarism. Namely, especially in the first-year programming courses, during solving the appointed problems less experienced students frequently apply the templates and ideas recommended by their teachers. Consequently, a certain level of code similarity can be expected and allowed to some extent. Also, if a part of the project is already implemented and the students are supposed to build upon it, a similarity is inevitable. Therefore, extreme caution is necessary and the software tools for similarity detection only raise some kind of indication about possible plagiarism. The final decision of whether some work is plagiarised or not is always brought by a human examiner [6].

For the purpose of exposing the concept of low-level programming in the introductory programming course at the University of Belgrade, School of Electrical Engineering, the students are taught to write the programs in the assembly language for an educational hypothetical architecture – *picoComputer*. It imposed a need for an appropriate software system for similarity detection of the source codes submitted as homework solutions. Such system is based on the *Greedy String Tiling* algorithm for similarity detection augmented with *Karp-Rabin* modification (*RKR-GST*). The system is successfully implemented and evaluated.

Following this introduction, the second section presents a brief overview of the *picoComputer* architecture, to have an impression of the complexity of demands for the system to fulfil.

The third section describes the structure of the similarity detection application and the GUI along with some implementation details. After that, the system is evaluated by performing some representative tests, and the results of the evaluations are analyzed in the fourth section. Finally, the paper concludes with a summary of the work done and a proposal of the future work.

2. THE *picoComputer* ARCHITECTURE

The educational computer architecture *picoComputer* (*pC*) is developed by prof. J. Dujmović in 1989 [7]. The goal was to conveniently introduce the students at the School of Electrical Engineering to the basic principles of the computer architecture and programming in an assembly language. Being a minor part of an introductory course, the architecture was quite restrictive, as implied by the prefix *pico*.

pC is a 16-bit machine with the three-address instruction format, as an instruction consists of the operation code (4-bit) and up to 3 operand fields (4-bit each – one bit for the addressing mode and 3-bit for the address). The instructions are either one-word (16b) or two-word (32b). The main memory of 64K 16-bit words is logically divided into the fixed area (0-7) and free area (rest of the address space). All addresses are 16-bit long. The fixed area accommodates only directly accessible data, while program and indirectly accessible data reside in the free area. There is a system stack that occupies the top of address space and grows downwards. The layout of the main memory address space is shown in Figure 1.

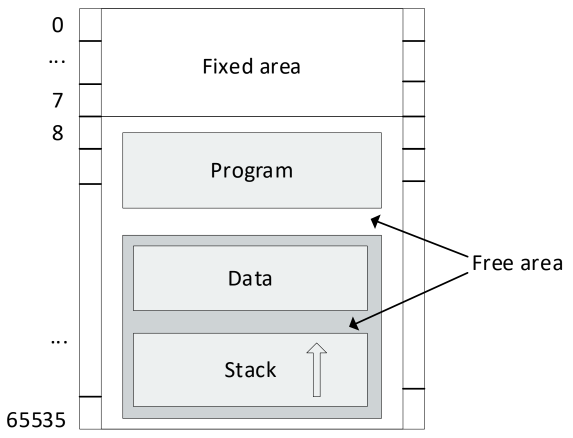


Figure 1 – The main memory address space.

In the *pC* architecture, the programmer can use three memory addressing modes:

1) *Direct memory addressing* – an operand resides in the fixed area, as only locations 0 to 7 can be addressed with 3-bit address;
2) *Indirect memory addressing* – the address of an operand resides in the fixed area, while the operand itself can reside anywhere in the main memory because of the 16-bit address;
3) *Immediate addressing* – an operand is a 16-bit constant found in the second instruction word.

The instruction set of the *pC* assembly language encompasses four typical groups of instructions:

1) *Data transfer instruction* (MOV) – enables the one-word memory-to-memory transfer or loading the constant to memory, or copying a contiguous memory block of data from source to destination, as well;
2) *Arithmetic instructions* (ADD, SUB, MUL, DIV) – enable the basic four arithmetic operations (addition, subtraction, multiplication, division);
3) *Control instructions* – there are three subgroups: conditional branch instructions (BEQ – condition is equality of the two operands, and BGT - condition assumes that the first operand is greater than the second), subroutine handling instructions (JSR for a subroutine call and RTS form a return from subroutine), and instruction for ending the program execution (STOP);
4) *Input/Output instructions* (IN, OUT) – enable the communication of a user and the program – input data entry from the keyboard and display of output data on the monitor.

There are also two assembler directives for defining the symbolic constants and determining the layout of the program in the memory address space.

3. THE SIMILARITY DETECTION SYSTEM

The system consists of two components [8]:

- the command-line similarity detection application implemented in *C++*,
- graphical user interface (GUI) implemented in *Java*.

The flow diagram of activities within the implemented system for similarity detection of the source programs written in the *pC* assembly language is presented in Figure 2.

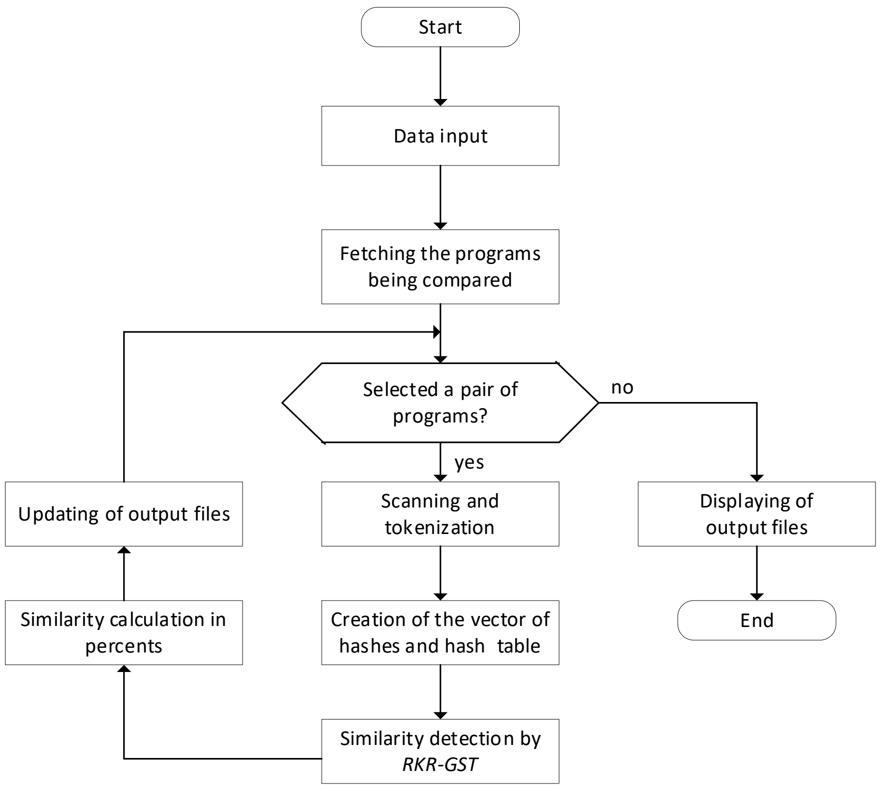

Start
Data input
Fetching the programs being compared
Selected a pair of programs?
no
yes
Updating of output files
Scanning and tokenization
Displaying of output files
Similarity calculation in percents
Creation of the vector of hashes and hash table
End
Similarity detection by RKR-GST

Figure 2 – The similarity detection system flow chart.

3.1. THE SIMILARITY DETECTION ALGORITHM

The algorithms for similarity detection usually apply a range of different approaches (strings, tokens, parsing trees, dependency graphs, various metrics, etc.) [9]. In an academic environment, a lower level of detection is regarded as effective since the functional similarity is quite expected in the student source codes. Also, those methods are much more effective in the case of large code repositories, which is common in an academic environment Therefore, in comparing the student assignments the similarity detection techniques are usually based on tokenization and string matching. A representative and widely used algorithm which relies on these two concepts is the *Greedy String Tiling (GST)* algorithm [10].

This algorithm performs searching for matchings in the two strings. It was firstly applied for DNA sequences matching, but also proved viable for the source code plagiarism detection, as well. It considers only matching of the sequences of the length no less than the MML (minimum matching length). The one-to-one matching is guaranteed and changing the positions of the matched subsequence would not affect the detection. The longer matchings are favored over the shorter ones since they are considered as better indicators of similarity.

The *GST* consists of the two phases which are executed iteratively until no one new matching longer than MML can be found. It incurs rather significant time complexity between O(n^2) and O(n^3). In order to decrease the time complexity of the *GST* algorithm, the *Karp-Rabin* modification is used which reduces the time-consuming character-based matching by the use of hashing. The rolling hash function calculates the hashes of substrings, and substrings with the same hash value are matched. This algorithm is known as *Running Karp Rabin Greedy String Tiling (RKR-GST)* [11].

3.2. THE SIMILARITY DETECTION APPLICATION

The application compares the similarity for each pair of files with the source codes written in the *pC* assembly language from a given data set. It is implemented in the *C++* programming language under *Linux Ubuntu* operating system.

The similarity detection is carried out in three phases:

1. Scanning and tokenization of a source program;
2. Hashing;
3. Running of the *RKR-GST* algorithm (with some potential optimizations).

The token set is based on the *pC* assembly language instruction set. It consists of 15 different tokens. Twelve tokens are based on instructions, one token for each instruction. Three additional tokens are introduced to model the labels and *pC* assembly language directives. The token set is chosen to balance both robustness to source code modifications and the precision of the system.

Phase 1: For the scanning process, the main class is *Scanner* with its methods scan (which scans an entire input file) and *scanLine* (which is called from the *scan* method for each text line of the input file to scan it).

The *scanLine* method firstly eliminates every blank, non-*ASCII* character (when writing in *Notepad*, especially on *Windows OS,* so-called *BOM* characters are inserted), as well as the comments from the entry text line. Then, the presence of the label at the beginning of the filtered line is checked. If the label exists, a new token is inserted into the token list which represents the output of the scanning process. Finally, the method tries to match the rest of the text line with some of the regular templates (*pattern, regex*). In case of a successful match, an appropriate token is added to the token list. One token from the token list is added for each regular expression.

Token and *TokenList* are the classes that represent the token and token list items, respectively. The T*oken* class contains appropriate member fields for token identification, for its location in the original file, and for keeping whether the token is marked or not. The *TokenList* class represents the list of tokens generated in the tokenization of one file, and such a list is used as one of two input strings in the *RKR-GST* algorithm. This class also contains the member fields related to hashing. The token list is implemented as a vector (*vector* type in *C++ STL* library) of pointers to objects of the *Element* class which, along with the pointer to the corresponding object of the *Token* class, contains also its ordinal number in the token list.

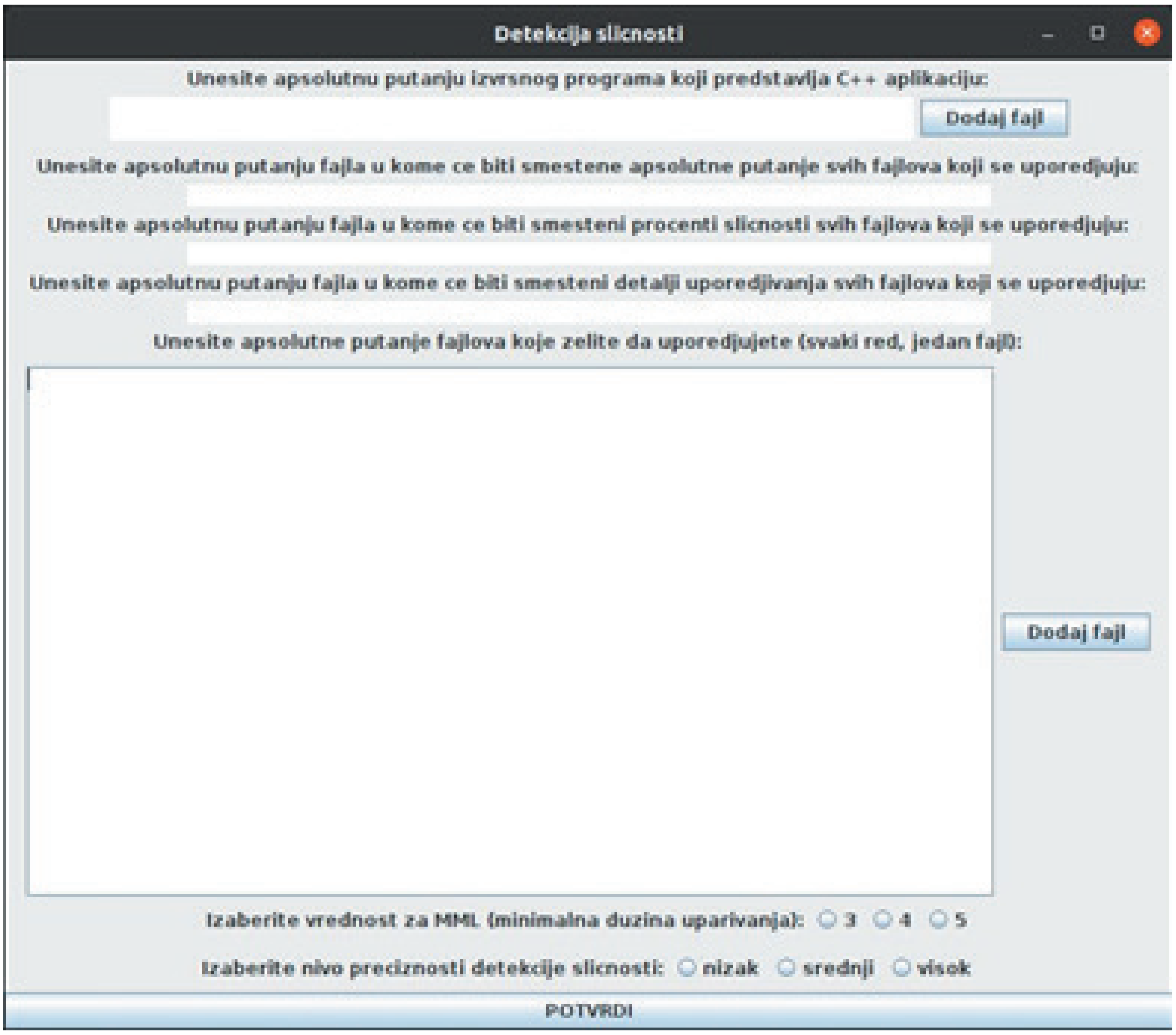
Phase 2: After the scanning and creation of the token lists for both programs are completed, the static *createHashesForTokenList* method of the *KarpRabinGST* class is called. It is a wrapper around the *fillHashes* method of the *TokenList* class. This method is responsible for generating the hashes of all consecutive substrings of the MML length for the tokenized program. The rolling hash function is used for this purpose. The chosen value of the base is 17 since it is the first prime number higher than the number of tokens (15).

During the generation of the hash values, they are also included in a vector of hashes and in the *map* hash table. These two structures are the member fields of the *TokenList* class and they have a role in the execution of the *RKR GST* algorithm. The *map* hash table is organized as an *unordered_map* structure from the *C++ STL* library. In this structure, each hash is connected with a vector of initial positions of all substrings corresponding to that hash value.

Phase 3: After the vector of hashes and the hash table for tokenized representations of both files under comparison are created, the static *runGST* method of the *KarpRabinGST* class is called. The input arguments of this method are the token lists for both files, the accuracy level of the similarity detection (the optimization level of the algorithm), and the value of MML. The output of the method is the vector of pointers to the objects of the *Match* class which contains the information about one matched substring (indices of the first substring token in the token lists for both files and the number of tokes in the matched substring). This method contains the implementation of the *RKR-GST* algorithm itself.

3.3. THE GUI SUBSYSTEM

For the purpose of the interaction with users, a Java application in Eclipse development environment is implemented using Java SE11. It is packed in the .jar file in order to enable the execution independently on the platform used. The appearance of a part of GUI is shown in Figure 3.


Detekcija slicnosti

Unesite apsolutnu putanju izvrsnog programa koji predstavlja C++ aplikaciju:

Unesite apsolutnu putanju fajla u kome ce biti smestene apsolutne putanje svih fajlova koji se uporedjuju:

Unesite apsolutnu putanju fajla u kome ce biti smesteni procenti slicnosti svih fajlova koji se uporedjuju:

Unesite apsolutnu putanju fajla u kome ce biti smesteni detalji uporedjivanja svih fajlova koji se uporedjuju:

Unesite apsolutne putanje fajlova koje zelite da uporedjujete (svaki red, jedan fajl):

Izaberite vrednost za MML (minimalna duzina uparivanja): ○ 3 ○ 4 ○ 5

Izaberite nivo preciznosti detekcije slicnosti: ○ nizak ○ srednji ○ visok

Figure 3 – An example of GUI.

It can be seen that a user should fill several fields and choose some options before starting the execution of the similarity detection algorithm. The first field requires the entry of an absolute path to the execution file of the *C++* application. The *JFileChooser* GUI component enables the choice of the execution file. Click on the *Dodaj fajl* button opens a dialog box where the desired file can be found and chosen or the path to the file can be manually entered.

The next three text fields are automatically filled with the directory part of the execution file path if it is selected using *Dodaj fajl* button. These three fields are used for entering absolute paths of: a) all files which should be checked for possible similarity, b) the file which represents the main system output, and c) the file which presents detailed information about the results of the similarity detection.

Below four text fields, there is a text area (*JTextArea*) where a user enters the absolute paths of all files (one file per line of entry) meant for similarity detection. This entry can be either manual or by using open dialog initiated with a click on the appropriate button.

Besides that, the user has to select the value of MML, one of the basic parameters of the *RKR-GST* algorithm. Only three values (3, 4, 5) are offered since a lower value would lead to some random matchings, while a higher value would be too restrictive having in mind that student *pC* assembly programs are relatively short. Finally, a level of precision in similarity detection must be chosen.

After all parameters and files names are specified, the execution of the *RKR-GST* algorithm is started by clicking the button on the bottom of the frame.

The main output result is stored in a file that contains the percentage of similarity for all pairs of files ('each with each') submitted for the similarity detection. An additional output result is a file with information about the token set used for tokenization in the *RKR-GST* algorithm, the starting positions of all matched subsequences in both files, and their lengths. The result of matching is provided from each pair of files.

4. TESTING OF THE SYSTEM

The system for similarity detection was tested using two different data sets:

1) Manually created set of the testing examples (simulation of plagiarism);
2) Real data set consisting of 250 student assignments.

In the first case, starting with the reference assembly program *prog_v0.pca*, three new programs, semantically equivalent to the reference one, are created. The intentional lexical and structural modifications are introduced in these three versions in order to hide plagiarism. A different time is spent for these modifications intended to deceive the system, so one hour was needed for *prog_v1.pca*, two hours for *prog_v2.pca*), and four hours for *prog_v4.pca*. Table 1 presents the results of the simulation for the given MML values and a high level of detection precision.

Comparison		Similarity (%)		
First program	**Second program**	**MML = 3**	**MML = 4**	**MML = 5**
prog_v0.pca	*prog_v1.pca*	87.7551	81.6327	73.4694
prog_v0.pca	*prog_v2.pca*	69.0909	52.7273	38.1818
prog_v0.pca	*prog_v4.pca*	53.913	33.0435	26.087
prog_v1.pca	*prog_v2.pca*	78.1818	72.7273	58.1818
prog_v1.pca	*prog_v4.pca*	59.1304	48.6957	27.8261
prog_v2.pca	*prog_v4.pca*	83.4646	83.4646	77.1654

Table 1 - The results of the plagiarism simulation.

Group	**Similarity (%)**									
	0-10	**10-20**	**20-30**	**30-40**	**40-50**	**50-60**	**60-70**	**70-80**	**80-90**	**90-100**
Group 1	0	4	37	123	100	31	4	1	0	0
Group 2	0	6	47	109	99	33	6	0	0	0
Group 3	0	8	46	114	95	30	6	1	0	0
Group 4	0	7	55	110	81	35	12	0	0	0
Group 5	0	1	22	108	111	48	7	3	0	0
Group 6	0	3	46	122	85	36	4	4	0	0
Group 7	0	2	25	96	115	49	11	2	0	0
Group 8	0	6	53	103	95	36	7	0	0	0
Group 9	0	3	48	125	87	34	1	2	0	0
Group 10	0	1	21	108	114	47	7	1	1	0
Total	0	41	400	1118	982	379	65	14	1	0

Table 2 - The results of the testing on a real data set

It can be seen from the table that the similarity scores are higher in the consecutive code versions. Differences between versions 0 and 1 are basically in the lexical changes and the RKR-GST algorithm can mainly detect them. The high similarity of versions 2 and 4 can be explained by somewhat fewer modifications done in the last two hours. Because of the simplicity of the pC assembly language, the number of possible modifications is limited. For MML = 3 the similarity percentage between the original and final version exceeds 50% which evidences that although the significant success in hiding the plagiarism can be achieved in four hours, it is still insufficient to fully remove the doubt about the possible plagiarism.

In the case of the real data set, it was not known in advance which programs were plagiarised and which are not. The data set of 250 student assignments is grouped into 10 groups (25 assignments per group). The comparisons on similarity are performed for each pair in a group (300 comparisons per group). Obtained percentages of similarity are classified into 10 ranges (10% each). Table 2 presents the obtained results for MML = 3 and a high level of detection precision.

It is evident that the percentage of similarity in several groups is relatively high, which raises the doubts about plagiarism. These are candidates for more careful manual examination by the instructor.

5. CONCLUSION

Contemporary information systems and wide accessibility to various kinds of knowledge have made the possibility of plagiarism easier than ever. Often some students in fulfilling their assignments cannot resist using the results of others' work, especially in the programming courses. In order to detect the similarities of the source codes written in the *picoComputer* assembly language, an appropriate similarity detection system is developed at the University of Belgrade, School of Electrical Engineering. The application written in *C++* for finding the similarities is based on the well-known *Greedy String Tiling* algorithm. In order to decrease its time complexity, the algorithm is extended with *Karp-Rabin* modification based on the rolling hash function. The GUI written in *Java* is also supplied for choosing the input data set and flexible choice of parameters. The system is successfully tested using two data sets: artificially simulated plagiarism of different levels in an example program and real data set consisting of 250 student assignments from an ongoing university course.

There are several avenues for prospective future work. By modification of the front-end part (scanning in the first phase), this system can be adapted for similarity detection in some other language. Also, the system is quite flexible and can be ported to a different platform (e.g., Windows). Finally, with an aim to increase its precision, some other token set can be envisioned which takes into account not only the instruction mnemonics but also the addressing modes of the operands. The choice of a token set could be one of the parameters of the system.

6. REFERENCES

[1] "What is Plagiarism?," 18 May 2017. [Online]. Available: https://www.plagiarism.org/article/what-is-plagiarism. [Accessed 20 March 2022].

[2] L. Prechelt, G. Malpohl and M. Philippsen, "Finding plagiarsms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016-1038, 2002.

[3] S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *ACM SIGMOD international conference on Management of data*, 2003.

[4] "TurnitIn iThenticate," [Online]. Available: https://www.turnitin.com/. [Accessed 23 March 2022].

[5] M. J. Mišić, J. Ž. Protić and M. V. Tomašević, "Improving source code plagiarism detection: Lessons learned," in *25th Telecommunication Forum (TELFOR), IEEE*, 2017.

[6] F. Culwin and T. Lancaster, "Visualising intra-corpal plagiarism," in *Fifth International Conference on Information Visualisation, IEEE*, 2001.

[7] J. J. Dujmović, Programski jezici i metode programiranja, Beograd: Akademska misao, 1991.

[8] V. M. Tomašević, "Softverski sistem za detekciju sličnosti u asemblerskom kodu picoComputera," Elektrotehnički fakultet, Beograd, 2021.

[9] M. Novak, M. Joy and D. Kermek, "Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review," *ACM Transactions on Computing Education (TOCE)*, vol. 19, no. 3, p. 27, 2019.

[10] M. J. Wise, "Detection of Similarities in Student Programs: YAP'ing may be Preferable to Plague'ing," in *Twenty-Third SIGCSE Technical Symposium*, Kansas City, USA, 1992.

[11] M. Wise, "Running Karp-Rabin matching and greedy string tiling," Basser Department of Computer Science, University of Sydney, 1993.