Identification of Similar Source Codes based on Longest Common Substrings

René Arnulfo García Hernández
Autonomous University of the State of Mexico
Santiago Tianguistenco, San Pedro Tlaltizapan
State of Mexico, Mexico
renearnulfo@hotmail.com

Yulia Ledeneva
Autonomous University of the State of Mexico
Santiago Tianguistenco, San Pedro Tlaltizapan
State of Mexico, Mexico
yledeneva@yahoo.com

ABSTRACT
In this paper, we describe the system developed by Autonomous University of the State of Mexico (in Spanish, UAEM) for the detection of source code re-use (SOCO) task of FIRE 2014. The aim of the SOCO task is to detect the most similar code pairs between a large source code collection in java and c languages. Our method is divided in for phases: preprocessing, similarity measure, ranking and getting the decision. One way to measure the similarity between a pair of codes is to use the length of the Longest Common Substring (LCS). However, beside the LCS there is another important set of longest common substrings (shorter than LCS) that are not taking into account. Our hypothesis is that if we use all the longest common substrings (LCSs) is possible to improve the detection of similarity between two codes. The second hypothesis is that a re-use case not only depends on the value of a measure but also depends on the similarity of other codes. For this, we get other parameters using the LCSs measure with respect to other codes. For taking a re-use case decision, we obtain some rules using the training corpus of SOCO.

Categories and Subject Descriptors
H.3 [**Information Storage and Retrieval**]: H.3.1 Content Analysis and Indexing; H.3.3 Information Search and Retrieval; H.3.4 Systems and Software

General Terms
Algorithms, Measurement, Performance, Experimentation, Languages

Keywords
Source Code Reuse, Longest Common Substrings, Similar Codes, Java Code Reuse, C Code Reuse.

1. INTRODUCTION

Even though is common to find a lot of web pages showing source codes in different languages, the source code is the result of an intellectual effort, for such reason, it is protected by copyright laws. Normally, the source code in the WEB is presented in short fragments with tutorial proposes. However, re-using source code of works brings economic problems for the author and legal problems for whom make the act. Some automatic tools have been developed to assist with the problematic of the re-use detection of source code. These tools can be classified in intrinsic or extrinsic tools. The intrinsic tools search re-uses cases in a given collection of source codes. In this case, every code in the collection is considering suspects. In the extrinsic tools, the problem consists of given a suspect to find the re-use case in other collections, like the Web.

JPLAG and MOSS are examples of free tools. JPLAG [1] was developed by Guido Malpohl in 1996 which supports Java, C#, C++, Scheme and natural language text. JPLAG uses a variation of the Karp-Rabin comparison algorithm developed by Wise [2]. First, JPLAG converts the source code into strings of token employing a parser. The parser brings more semantic information and depends on the source language. MOSS [3] (Measure Of Software Similarity) was developed by Alex Aiken in 1994. MOSS works with different languages: C, C++, Java, Pascal, Ada, Lisp and Scheme. MOSS is based on getting fingerprints which identifying a source code in particular way. According to the Web page of MOOS the ideas of its algorithm can be found in [7]. The idea is that the more common fingerprints exist between a pair of source codes, the more similar they are. Fingerprints are a small subset of all the n-grams (substrings of *n* characters) that exists in a source code. The fingerprint is form with a unique value that represents an n-gram (normally, a hash functions is applied).

Sherlock and PMD are open-source free tools. Sherlock was developed by the department of Computational Science of the University of Warwick. Sherlock works with source codes and natural language texts. Also, PDM uses the string-matching comparison algorithm of Karp-Rabin. PDM supports Java, JSP, C, C++, Fortran and PHP.

CodeMatch is a commercial tool that supports the languages: BASIC, C, C++, C#, Delphi, Flash, Java, etc. CodeMatch is based on the combinations of five algorithms: Source Line Matching, Comment Line Matching, Word Matching, Partial Word Matching, and Semantic Sequence Matching. For processing a source code, first CodeMatch separates comments, identifiers (name of variables, names of constants, names of functions, etc.) and functional code. Word Matching algorithm obtains for each code a substring of words (eliminating reserved words) that allow counting the number of common words in this sequence. Unlike Word Matching, for Partial Word Matching is not necessary that the complete words matching, it could be partial. Source Line Matching compares the source lines (excluding comments) of the source code pair. On the contrary, Comment Line Matching compares the lines of comments excluding the lines with functional code. Semantic Sequence Matching compares the lines of codes using the first word (excluding comments) of the pair of source codes. Finally, a single score is given for the similarity of the source code pair.

The state of the art in the s research deals with the detection of source-code re-use across programming languages [6].

2. Proposed system

Our system (UAEM) used for the detection of source code reuse is divided into four phases.

2.1 Preprocessing phase

In the first phase, only the lexical items (like {,},(,),+,*,;.etc.) of each source code are separated with a whitespace and more than one whitespace is removed. The result of this phase is a string of tokens of the source code. This phase depends on the input language, but for C and Java is almost the same. Even thought, we test some options like remove comments or identifiers, the evaluation in the training corpus decreases.

2.2 Similarity measure phase

In the second phase, for each source code given as a string, the similarity measure with respect to the other source codes is obtained. The sum of the different lengths of the longest common substrings between the two source codes (normalized to the length of the longest code) is used as the similarity measure. For this phase, we used the algorithm in [4].

2.3 Ranking phase

In the third phase, a set of parameters that allow later the identification of cases of re-use is obtained using comparisons done in the previous phase. The parameters obtained are: the value of the DISTANCE (1 - similarity), the RANKING of the distance (rank order of the most similar), the GAP that exists with the next closest code (it is only calculated for the first 10 closest codes) and, using the maximum gap between the 10 most closest codes, the codes that are (B)efore or (A)fter the maximum gap (RELATIVE DIFFERENCE) are labeled. The result of the third phase is a matrix where each row represents a comparison of a source code with other codes (columns) and each cell represents a pair of source codes in both directions.

2.4 Reuse decision phase

For taking the decision, a source code pair X↔Y will be a reuse case, if there is evidence of reuse in both directions, it means, X→Y and Y→X. A reuse case exists when the DISTANCE is less than 0.45 or the GAP is greater than 0.14, but also it is important that one of the additional conditions is achieved. The first condition is that the RANKING must be, at least, in the second position and, the second condition, that the label of the RELATIVE DIFFERENCE must be B. The first run for C and Java languages were processed with above conditions. However, in some cases the evidence in one direction was very high and in the other direction was almost reliable, but according to the training corpus in Java and C, in most of the cases this pair was a code reuse case. In the second run, if there were not high evidence of reuse in one direction, then the pair can be considered as reuse case whether at least one of the both codes has the RANKING of 1 and the RELATIVE DIFFERENCE of B and the GAP greater than 0.1.

3. Training experiments

The training corpus consists of 259 source codes in Java and 79 sources codes in C. Relevant judgments in Java has 84 pairs and in C has 26 pairs. In table 1 is showed the results of our system with the training corpus for C and Java. In the first run with Java the system gets a better recall than precision, and in the second run the precision is better. However, with C language the system obtains the same precision, the difference was in the recall. Most of the rules were tuning with Java corpus since more re-use cases exists in the relevant judgments. In this sense, we think that the results in C are worse since there are only 26 pairs for training.

Table 1. Results with training corpus according to our evaluation.

Corpus-Run	Precision	Recall	F-measure
Java-Run1	0.78	0.83	0.80
Java-Run2	0.85	0.80	0.83
C-Run1	0.80	0.58	0.67
C-Run2	0.80	0.63	0.71

4. Testing experiments

We were surprised when the test corpus was delivered, it was bigger than we expected. The Java corpus has 12,080 files divided in 6 scenarios. In the case of C corpus, it has 19,895 files divided in 6 scenarios, too. Table 2 shows the distribution of the corpus according to SOCO scenarios.

Table 2. Distribution of test corpus according to the scenario.

Scenario	Java	C
A1	3,241	5,408
A2	3,093	5,195
B1	3,268	4,939
B2	2,266	3,873
C1	124	335
C2	88	145

At the beginning, the system was not optimized for running with bigger collections. The time estimated for processing the whole corpus was of 3 months, unacceptable for SOCO time competition. After a reprograming the system was possible to process the collections in one day using a computer with CPU Xenon with 6 cores and 32 GB in RAM.

Since we did not know how the evaluation will be done, it could be done: by scenario, by language or by runs; we decide to do the combination of 2 runs to use 3 options that we are able to submit. These is the explanation of why run 2 and run 3 are the same in Java, and why run 1 and run 2 are the same in C. The results of our system (UAEM) with the test corpus for C are showed in Table 3. The f-measure results for UAEM-run1 and UAEM-run2 (actually, it correspond to the system tuning to C-Run1 in training phase) were better than UAEM-run3 (it corresponds to C-Run2 in training phase). However, in the training phase the system tuning to C-Run1 was worse in the recall and the precision was better than recall. We think this variation is possible since the training corpus is very small compared with the test corpus.

Table 3. Results of the systems for C according to the first SOCO evaluation.

Rank	Team-Run	Precision	Recall	F-measure
1	UAEM-run1	0.306	0.500	0.380

2	UAEM-run2	0.306	0.500	0.380
3	UAEM-run3	0.260	0.500	0.342
#	Baseline-1	0.400	0.069	0.117
#	Baseline-2	0.040	0.280	0.060
4	UAM-C-run1	0.007	0.494	0.013
5	UAM-C-run3	0.007	0.493	0.013
6	UAM-C-run2	0.005	0.444	0.010

The results of our system (UAEM) with the test corpus for Java are showed in Table 4. The evaluation for UAEM-run2 and UAEM-run3 (actually, it corresponds to the system tuning to JAVA-Run2 in the training phase) were better than UAEM-run1 (it corresponds to JAVA-Run1). As in previous evaluation, the system has a different behavior with respect to the training phase. Nevertheless, the results in Java were better than in C.

Table 4. Results of the systems for java according to first SOCO evaluation.

Rank	Team-Run	Precision	Recall	F-measure
1	UAEM-run2	0.641	0.969	0.771
2	UAEM-run3	0.641	0.969	0.771
3	UAEM-run1	0.759	0.472	0.582
4	UAM-C-run1	0.633	0.435	0.515
5	DCU-run3	0.775	0.360	0.492
6	DCU-run2	0.777	0.350	0.482
7	DCU-run1	0.658	0.364	0.468
8	UAM-C-run3	0.926	0.311	0.465
#	Baseline-2	0.464	0.288	0.356
#	Baseline-1	0.617	0.080	0.141
9	UAM-C-run2	0.029	0.343	0.054

The configuration of Baseline-1 corresponds to JPLAG program with the default parameters, and the configuration of Baseline-2 consists of a character 3-gram model weighted using term frequency and cosine measure to compute the similarity. This baseline considers as re-used cases all source code pairs that surpass a similarity threshold of 0.9. An overview of the SOCO Track can be found in [5]

4.1 Second SOCO evaluation

The second evaluation by SOCO eliminates our third run since was a combination of the runs 1 and 2. Results for C are showed in table 5 and results for java in table 6.

Table 5. Results of the systems for C according to the second SOCO evaluation.

Rank	Team-Run	Precision	Recall	F-measure
1	UAEM-run1	0.282	0.100	0.440
2	UAEM-run2	0.240	0.100	0.387
#	Baseline-2	0.258	0.345	0.295
#	Baseline-1	0.350	0.130	0.190
4	UAM-C-run1	0.006	1.000	0.013
5	UAM-C-run3	0.006	0.997	0.013
6	UAM-C-run2	0.005	0.950	0.010

According to the second evaluation by SOCO, our system retains the top position and the f-measure was increased, but the UAM-C team obtained the same f-measure.

Table 6. Results of the systems for java according to second SOCO evaluation.

Rank	Team-Run	Precision	Recall	F-measure
1	UAM-C-run3	0.691	0.968	0.807
2	DCU-run2	0.530	0.995	0.692
3	DCU-run3	0.515	1.000	0.680
4	DCU-run1	0.432	0.995	0.602
#	baseline 2	0.457	0.712	0.556
5	UAEM-run1	0.385	1.000	0.556
6	UAM-C-run1	0.349	1.000	0.517
#	baseline 1	0.542	0.293	0.380
7	UAEM-run2	0.158	1.000	0.273
8	UAM-C-run2	0.019	0.928	0.037

According to the second evaluation in java, our system obtains the fifth position with run1 and the seventh position with run 2.

5. Conclusions and future work

In this paper, a new system for detecting re-use of source code is described. The proposed system works in four phases. The preprocessing phase is very interesting since it does not require sophisticated processes or dictionaries, making the execution of this phase very fast. It is worth noting that all phases of our system work with words, making the process faster than when working with characters. The second phase introduces a new measure based on the different lengths of longest common substrings between the pairs of source codes which outperform LCS. Third phase presents a new way for considering other parameters derived from the LCSs measure. These parameters allow proposing some rules for catch some groups of re-use cases. According to first SOCO evaluation, our system outperforms other systems in both cases. Even thought, the evaluation of Java reach the best f-measure score, the results in C are also relevant, since was the unique system that surpass both baselines in the first evaluation.

In both second evaluations is interesting to observe that all of the systems obtains excellent recalls between 0.928 and 1.000. Therefore, as a future work we must concentrate our efforts in precision. Also, as future work, we think the rules for C can be improved considering some more re-use cases.

6. REFERENCES

[1] L. Prechelt, G. Malpohl and M. Phlippsen, 2000. JPlag: Finding plagiarism among a set of programs. Technical Report, Universität Karlsruhe, Germany.

[2] A. Aiken. 1998. MOSS (Measure Of Software Similarity) plagiarism detection system. http://www.cs.berkeley.edu/~moss/ (as of April 2000) and personal communication, University of Berkeley, CA.

[3] R.M. Karp and M.O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31(2), 249-260.

[4] R.A. García-Hernández, J. Martínez-Trinidad and J. Carrasco-Ochoa, 2006. A new algorithm for fast discovery of maximal sequential patterns in a document collection. Computational Linguistics and Intelligent Text Processing, LNCS 3878, Springer, 514-523, Mexico.

[5] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello, 2014. PAN@FIRE 2014: Overview of SOCO Track on the Detection of SOurce COde Re-use. In *Proceedings of the Sixth Forum for Information Retrieval Evaluation* (FIRE 2014), Bangalore, India.

[6] E. Flores, A. Barrón-Cedeño, P. Rosso and L. Moreno, 2012. Detecting source code re-use across programming languages. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics:* Demostration Session, NAACL, 1-4.

[7] S. Schleimer, D. Wilkerson, A. Aiken, 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International conference on Management of data*, 76-85, CA.