Detecting code re-use potential

Mario Konecki, Tihomir Orehovački, Alen Lovrenčić
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
{mario.konecki, tihomir.orehovacki, alen.lovrencic}@foi.hr

Abstract. *Living in a dynamic world requires rapid development of both web and desktop applications to support such trend in IT industry. Processes are becoming more complicated, and in turn more demanding. There are two problems regarding applications: development and maintenance. In this paper we are discussing optimization of applications code and re-usability. The main idea is to compare certain application parts or modules and determine the amount of overlapping content. If there is a certain percentage of overlapping, it means that targeted part of code can be optimized in such way that it is programmed in one place and then re-used as such in other places. This speeds up development and makes maintenance easier. In this paper we will present the process of code comparison and pre-processing that is needed to recognize invariants of the same code.*

Keywords: development process, program code, comparison, reusability, optimization

1 Introduction

When looking at the software development process today we can say that there are certain efforts that have been made in order to make this process quicker and easier. Some of the concepts that we encounter here are design patterns, components, metacomponents, etc. Design patterns [4] are tested and reliable solutions for reoccurring problems in software engineering. They are concepts and cannot be translated directly into code. Component [2] is a part of larger software system and it has the ability of providing some service to its surroundings which means that it communicates with other components in order to solve some problem. Components are reusable and flexible which means that they are not specifically designed to run in just one system. Metacomponents [17] are descriptions that make it possible to generate concrete components so they are another step forward.

All of these concepts and more of them brought developers a little bit closer to their goal which is developing applications faster and with a larger reliability. Along with all these concepts there are more aspects that must be considered. One of them is duplication of computer code, especially in components but also in other concepts of software development. It is of our interest to find out how much of software code is duplicated when developing and application in various components and other software parts.

In order to do this we have created a concept upon which an algorithm for computer code comparison will be created. It is our intention to analyze results of this comparison and determine which steps are to be taken in order to optimize the development process. This concept and discussion about possible solutions is presented in the following sections of this paper.

2 Background research

When talking about code comparison the first thing we have to look is comparison in general. Comparison is frequently mentioned when talking about plagiarism. In the field of education this problem is mostly related to Higher Education [13].

First ideas about comparison came related to student essays and papers. It was of great interest to find out how many similarities there are. When talking about essays and papers we can say that there are two categories [13]:

- plagiarism – taking content from Web and other sources and declaring it as one's own.
- collusion – collaboration between students when working on some assignment that was meant to be done individually.

There are two frequently used methods for plagiarism detection [13]:

- Turnitin – a browser-based tool that compares uploaded files against a base of Web content and with related student papers.
- Ferret copy detector – a standalone system that is based on a fact that most ordinary words appear quite rarely in texts.

In the Brown corpus of 1 million words, 40% of the word forms occur only once [10]. This distinctive distribution is even more distinctive on bigrams (two consecutive words) and even more on trigrams (three consecutive words). It was realized that trigrams are the smallest elements by which usage it is possible to fingerprint particular text [13]. Any article has in

average 77% of its trigrams unique [13]. So articles can be processed by dividing text into trigrams and comparing occurrence of this trigrams in particular texts.

When comparing computer code the process can be simplified or can be analyzed from more complex point of view. The Ferret detector/comparator can be used. Code is divided into trigrams with some preprocessing. For example sing "==" must be treated as one word. But also more complex algorithms can be used.

Another aspect of interest already mentioned is optimization of computer code and making maintenance easier. There are several possible algorithms that can be used here [1]:

- Text-based techniques perform little or no transformation to the "raw" source code before attempting to detect identical or similar (sequences
- of) lines of code. Typically, white space and comments are ignored.
- Token-based techniques apply a lexical analysis (tokenization) to the source code and, subsequently, use the tokens as a basis for clone detection.
- AST-based techniques use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar subtrees in this AST.
- PDG-based approaches go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of a semantical nature, such as control and data flow of the program.
- Metrics-based techniques are related to hashing algorithms. For each fragment of a program, the values of a number of metrics are calculated, which are subsequently used to find similar fragments.
- Information Retrieval-based methods aim at discovering similar high level concepts by exploiting semantic similarities present in the source code itself (including the comments).

There are many tools available that have different algorithms and different usage, such as JPlag [15].

Some of the most known tools that can be found for comparison of code or papers are [5]:

- Turnitin – comparison of uploaded papers against the base of articles from Web.
- JPlag – finds similarities between uploaded documents.
- EVE2 – standalone software for papers plagiarism detection with possibility of different strengths of comparison/detection.
- CopyCatchGold – detects similarities between papers even when author changes order of words, sentences or uses only a part of the paper.
- WordCheck – checks similarity of paper with other papers written by same or different author based on frequency of occurring words.
- MOSS – determines computer code similarities.

3 Computer code comparison

In order to compare two pieces of computer code a proper concept for this action has been developed. This concept considers C-like languages code comparison but the concept is applicable to all programming languages.

The steps that are to be taken in order to perform the comparison are:

1. divide program code into parts where one part is one function
2. remove all declarations of variables or functions
3. replace all variable names with a constant name X
4. replace all function names with a constant name Y
5. remove all input commands (lines)
6. remove all output commands (lines)
7. remove all blank lines
8. remove all blank spaces
9. if there are lines with only "(" and ")" then read all those lines, lines between them and form one line of format (content)
10. if there are "{" or "}" at the beginning or end of lines then move these brackets to a new line before or after the content between them
11. compare all lines of all computer code by parts that are result of the first step
12. also compare the size of these parts in order to try predicting the content of a party by its size

Pseudo code of this process is given below:

read input files(s)
if there is more than one function per file
    split all parts into smaller parts that consists of only one function
for every small part (function) do the following
        remove all declarations of variables or functions
        replace all variable names with a constant name X
        replace all function name with a constant name Y
        remove all input commands (lines)
        remove all output commands (lines)
        remove all blank lines
        remove all blank spaces

if there are lines with only "(" and ")" then
read all those lines and form one line of format (content)
if there are "{" or "}" at the beginning or end of lines then
move these brackets to a new line before or after the content between them
compare all lines of all computer code by parts (all with all comparison)
compare the size of these parts in order to try predicting the content of a party by its size

A flowchart diagram is also given below:

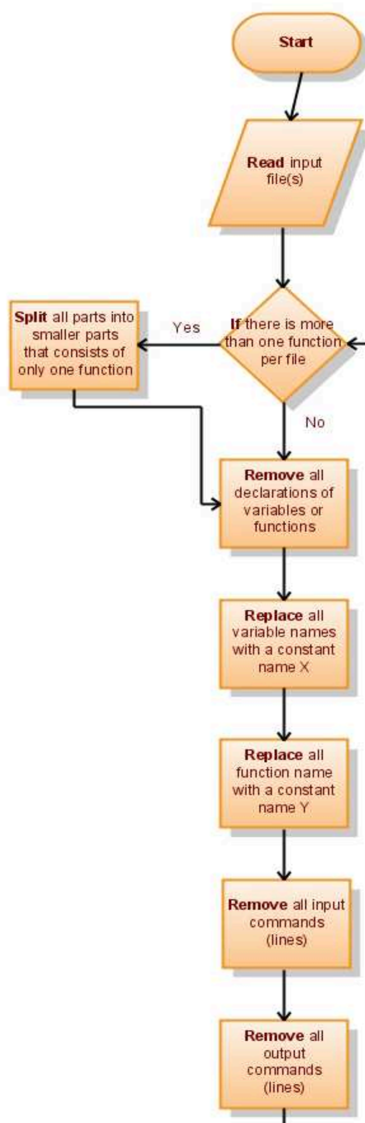

Figure 1. Flowchart part one

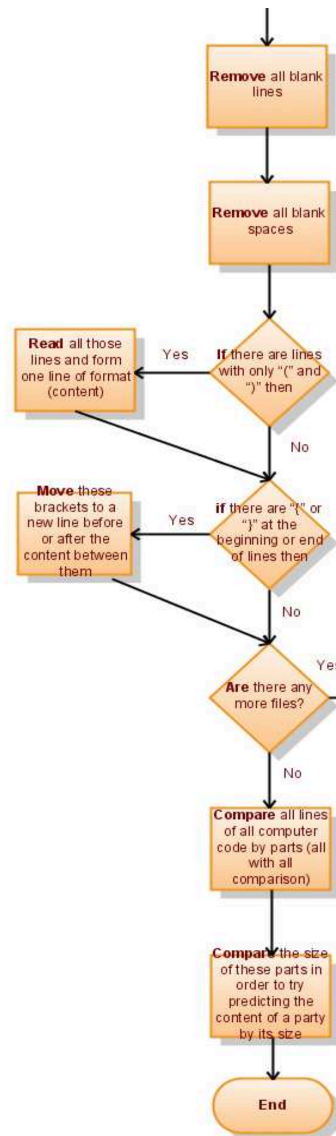

Figure 2. Flowchart part two

This algorithm takes n programs as an input and then does the comparison but it also tries to learn in step 12 where it tries to recognize over time the sizes of programs and connecting them to some specific code pattern.

The input for this algorithm can be various programs or modules of the same program. In this way one can se how much code is duplicated in an application and then can isolate this parts and put them in just one place. In this case one would for example create instances of the same class rather than several classes that are too similar and can be joined into one.

By doing this application code is reduced and optimized, maintenance of the code is made easier and future development quicker.

4 String similarity/difference metric

The 11th step of out code comparison process compares lines of code. These lines can be observed as pure strings. There are several algorithms in information theory and computer science that deal with calculating so called edit distance (number of operations required to transform one string into another). Some of more known algorithm are:

- Hamming distance [6] which is applicable for comparing strings of the same length and presents the number of position for which the strings are different.
- Levenshtein distance [11][14][9] measures the amount of difference between two string. It represents a minimum of operations that are needed to transform one string into another. Allowed operations are insertion, deletion or substitution of a single character.
- Damerau-Levenshtein distance [3][12] is a generalization of Levenshtein distance and it is virtually the same algorithm but it also allows the transposition of two characters as an operation.
- Jaro-Winkler distance [8] is a measure of similarity between two strings.

Some of the other algorithms that can be found are:

- Wagner-Fischer edit distance [18]
- Ukkonen [16]
- Hirshberg [7]
- etc.

5 Conclusion and future work

Computer code comparison and optimization is a definite need in the overall development process. In this article we give an idea of coming just one step closer to faster and more reliable software development and easier maintenance.

It is our intention in our future work to develop a prototype of this comparison algorithm and conduct a detail case study where we would find out about robustness and reliability of this algorithm. A more detailed research will be conducted upon detailed testing of the prototype. When we establish well tested and proven model for this aspect of software development we will research in more detail the possibilities and areas of interest where this concept could find its value. A more aggressive benchmark has to be taken in order to develop a suitable and usable algorithm that will process the analysis in a reasonable amount of time.

We will also give an index of usability according to the programming areas. For example we think that web and distributed applications would greatly benefit of this model.

References

[1] Bruntink, M., Deursen, A., Engelen, R., Tourwe, T.: **On the Use of Clone Detection for Identifying Crosscutting Concern Code**, IEEE Transactions on Software Engineering, Vol. 31, No. 10, 2005, pp. 804-818.

[2] Crnkovic, I., Larsson, M.: **Building Reliable Component-Based Software Systems**, Artech House, Boston, 2002.

[3] Damerau, F.J.: A technique for computer detection and correction of spelling errors, Communications of the ACM, 1964.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.; **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995.

[5] Gaither, R.: **Plagiarism Detection Services**, Shapiro Undergraduate Library, University of Michigan, 2002.

[6] Hamming, R. W.: **Error Detecting and Error Correcting Codes**, Bell System Technical Journal, Vol. 26, No. 2, 1950, pp. 147-160.

[7] Hirschberg, D. S.: **A linear space algorithm for computing maximal common subsequences**, Communications of the ACM, Vol. 18, No. 6, 1975, pp. 341-343.

[8] Jaro, M. A.: **Advances in record linking methodology as applied to the 1985 census of Tampa Florida**, Journal of the American Statistical Society, Vol. 84, 1989, pp. 414–420.

[9] Konstantinidis, S.: **Computing the Levenshtein distance of a regular language**, Information Theory Workshop, 29 August – 1 September 2005, IEEE.

[10] Kupiec, J.: **Robust part-of-speech tagging using a hidden Markov model**, Computer Speech and Language, Vol. 6, 1992, pp. 225-242.

[11] Levenshtein, V. I.: **Binary codes capable of correcting deletions, insertions, and reversals**, Soviet Physics Doklady, Vol. 10, No. 8, 1966, pp. 707–710.

[12] Lowrance, R., Wagner, R: **An Extension of the String-to-String Correction Problem**, Journal of the ACM, Vol. 22, No. 2, 1975, pp. 177-183.

[13] Lyon, C., Barrett, R., Malcolm, J.: **Plagiarism Is Easy, But Also Easy To Detect**, Plagiary: Cross-Disciplinary Studies in Plagiarism, Fabrication, and Falsification, Vol. 1, No. 5, 2006, pp. 1-10

[14] Navarro, G.: **A guided tour to approximate string matching**, ACM Computing Surveys, Vol. 33, No. 1, 2001, pp. 31–88.

[15] Prechelt, L., Malpohl, G., Philippsen, M.: **Finding Plagiarisms among a Set of Programs with JPlag**, Journal of Universal Computer Science, Vol. 8, No. 11, 2002, pp. 1-44.

[16] Ukkonen, E.: **Algorithms for approximate string matching**, Information and Control, Vol. 64, 1985, pp. 100-118.

[17] Villacıs, J. E.: **The Component Architecture Toolkit**, Indiana University, Department of Computer Science, 1999.

[18] Wagner, R. A., Fischer, M. J.; **The String-to-String Correction Problem,** Journal of the ACM, Vol. 21, No. 1, 1974, pp. 168-173.