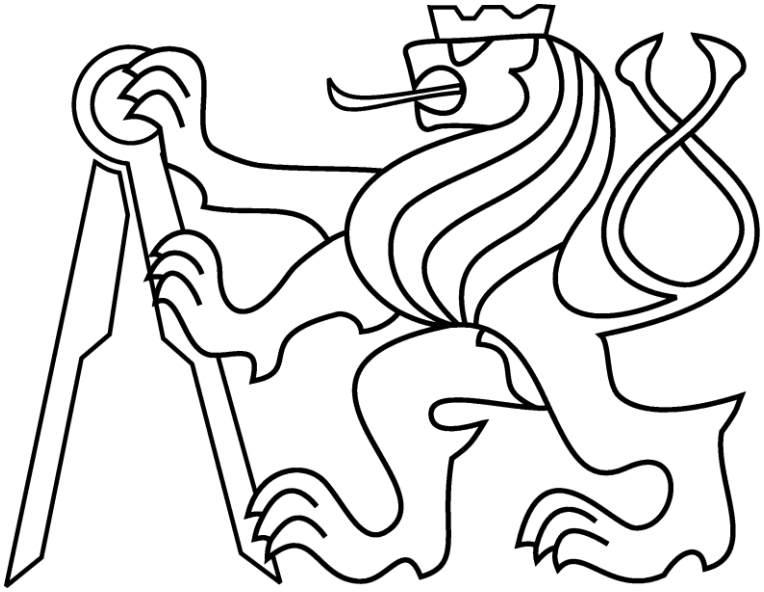
Czech Technical University in Prague

Faculty of Electrical Engineering



Bachelor´s Project

BARRISTER

Engine for duplicity searching in source code

Jiří Fryč

May 10, 2017

Supervisor: ING. ONDŘEJ MACEK, PH.D.

Bachelor thesis assignment

Declaration

I hereby declare that I have completed this submitted thesis independently and that I have listed all the information sources, such as literature and publications, according to Methodological instruction about adherence to Ethical principles during preparations of Bachelor's thesis.

I have no objection to the usage of this work in compliance with the Act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague, May 25, 2017 . . . . . .

signature

Acknowledgements

I would like to thank all the people who helped me with this project. At first, I would like to thank my supervisor Ing. Ondřej Macek, PH.D. for a lot of useful advice and aid with kicking off this project, so it will not end up in a drawer but will be used by our faculty. I would also like to thank Bc. Michal Roch and Ing. Jan Kočí for help with interconnectivity to faculty systems. And last but not least I would like to thank Ing. Jan Kubr and Ing. Jan Drchal, Ph.D. for providing me necessary data from faculty servers for proper analysis and testing.

Abstract

In this bachelor thesis, we analyze possibilities for plagiarism detection inside source files and implement an ideal solution for CTU FEE. Due to this focus on CTU FEE, we try to solve plagiarism detection mostly for the source code of students and their homework´s. An important part of this project is extensibility by another functions or languages. Work is being written in Java due to requirements by CTU CZM and profusely use XML and XSD technologies for input and output files. These files are not in the human readable format because they are used by another service called Prosecutor inside CTU CZM.

Keywords

Java, File testing, Plagiarism, Source code, Source code tokenization, Comparison

Abstrakt

V rámci bakalářská práce analyzuje možnosti pro detekci plagiátů ve zdrojových kódech a vytváříme ideální řešení pro ČVUT FEL.Vzhledem k zaměření řešíme detekci plagiátů převážně pro testování úloh zadaných studentům. Důležitou částí je možnost případného rozšiřování o další funkcionality, či programovací jazyky. Práce je psaná v Java z důvodů požadavků ČVUT CZM a hojně využívá technologie XML a XSD pro vstupní configurace a výstupní soubory. Vstupem a výstupem této práce není člověkem čitelný záznam a configurace jelikož řešení má sloužit jako serverová service pro system Prosecutor.

Klíčová slova

Java, Testování souborů, Plagiarism, Tokenizace zdrojového kódu, Porovnávání

Content

Dictionary

Submission (Entity)	Submission is a collection of files that are tested as single Entity. E.q.: A collection of files, that solves homework, from a single student.
Submission set	A collection of submissions with some connection, either there are located in the same folder, or contains some semantic connection. E.g.: Student´s homework from the year 2016 Student´s homework from the year 2017
Basecode	The source code that is provided to all entities. E.g.: Interface class provided to students for implementation with comments how each method should work.
Entity (Submission)	An entity which files are tested, also known as submission. E.g.: Students
Token	A small part of the code (usually with only a few characters) with assigned meaning. E.g.: i = 10 ; This code contains multiple tokens, one of them is: "=" with meaning "assigning value." or ";" with meaning "end of the statement."
One-way test (One-way relation)	Type of relation between two submission sets. This specific relation only allows testing inside one submission set and with second submission set. However, disallow testing inside second submission set. E.g.: $Submission´s\ sets\ A, B\ with\ relation\ A \rightarrow B$ $\exists a_1 a_2 \in A \quad a_1 \sim a_2$ $\exists a \in A, \exists b \in B \quad a \sim b$ $\forall b_1, b_2 \in B \quad b_1 \nsim b_2$ Example usage: If we want to test student homework from current year against each other and last year, but we do not want to detect plagiarism between students from last year.
Two-way test (Two-way relation)	Type of relation between two submission sets. This specific relation allows all possible testing for both submission sets.

	E.g.: $Submission's\ sets\ A, B\ with\ relation\ A \rightarrow B$ $\exists a_1 a_2 \in A \quad a_1 \sim a_2$ $\exists a \in A\ , \exists b \in B \quad a \sim b$ $\forall b_1, b_2 \in B \quad b_1 \sim b_2$
Generic code	Type of code that is expected to appear in all or almost all submissions. E.g.: Importing common packages. For example java.util.* in Java
Flagged (flagged entity)	Entity marked as plagiarism.

List of figures

List of tables

1 Project Introduction

Plagiarism, the use of another author´s works, is considered as one of the biggest problems in education. Source codes, texts, and others works are plagiarized to complete assignments, homework or even final exams. Fighting plagiarism in some areas became tough in last twenty years due to the expansion of the Internet, electronic communication and repository like websites. Without the use of automatic detection, teachers cannot handle verifying each of the student's work against each other (some courses contains 100+ students) or the Internet. So there is a need for semi or fully automated system.[1]

This bachelor thesis deals with an implementation of Barrister[2], system that would detect plagiarism in a supplied source codes. We chose this theme after a one month of discussion between several departments on CTU FEE. Firstly we wanted to create a cloud for testing students homework's, heavily focused on performance and code quality testing, with easy to use API for teachers to develop their tests. However, many departments were against it for numerous reasons, mostly they were afraid that we would leave backdoors for other students, but as we can find on http://stm-wiki.cz, their current testing software has already many backdoors which are left untreated for years.

So after that, we (me and Michal Roch, who represent CTU CZM) came with the idea of plagiarism detection software. Currently (in the year 2016) there are not as many publicly accessible tools, which are focused on this problematic. The aim of this project is to analyze and develop a solution for detecting plagiarism inside the source code, in other words, found source codes that are high likely duplicates of the one another.

1.1 Document structure

This paper is separated into nine chapters. 1introduces the reader to this project and problematics around plagiarism and its detection inside the source code. 2 contains an analysis of reasons for plagiarism, methods for detecting plagiarism and others problems surrounding

[1] If you, as a reader, don't believe that plagiarism is one of the biggest problems in education and you are from CTU FEE then I recommend visiting http://stm-wiki.cz (website is only in Czech language) before reading further. It could change your perspective on this topic.

On this website, you will find how well organized students (of the study program STM on the CTU FEE) are, when it comes to sharing information about tests, homework´s, etc. Also, this website has up to 10 years history of the courses.
(But on the other side this website is better organized, regarding information about courses, that any website created on CTU FEE)

[2] Name of the project "Barrister" was taken from the jurisdiction of England, where job Barrister is described as: "A barrister, who can be considered as a jurist, is a lawyer who represents a litigant as an advocate before a court of appropriate jurisdiction. A barrister speaks in court and presents the case before a judge or jury." [15]

plagiarism. At the end of this chapter, we also take a look at major problems in detection and common techniques to obfuscate plagiarized source code.

3provides a comparison of current services for plagiarism detection, based on artificial testing sets of source code and also real students source code obtained from several courses taught at CTU FEE. 4outlines how testing lifecycle in Barrister works. 5 is more technical, maybe not so much interesting for people without programming skills, it is describing the implementation of the Barrister. 6informs about the result of the Barrister testing.

7is above the scope of this assignment. This chapter contains a newly proposed algorithm for plagiarism detection. It differs a lot from current algorithms and could be viewed as active when compared to current algorithms. 8describe future of this project. 9 contains conclusion about this thesis and future steps I will be making in this field.

1.2 Plagiarism

According to Merriam-Webster website word “Plagiarize” has meaning [1]:

- AS TRANSITIVE VERB
  - TO STEAL AND PASS OFF (THE IDEAS OR WORDS OF ANOTHER) AS ONE'S OWN
  - USE (ANOTHER'S PRODUCTION) WITHOUT CREDITING THE SOURCE
- AS INTRANSITIVE VERB
  - TO COMMIT LITERARY THEFT
  - PRESENT AS NEW AND ORIGINAL AN IDEA OR PRODUCT DERIVED FROM AN EXISTING SOURCE

What do we consider as plagiarism?

- Turning in someone else´s work as own.
- Copying parts or ideas from someone else work without giving credit.
- Giving incorrect information about the source of a quotation.
- Copying multiple small parts so that it will constitute together a larger part of code.
- Changing parts but copying the structure of someone else works without giving credit.

On the contrary, we do not consider as plagiarism following:

- Consulting assignment with the teacher.
- Using own older work, but this should be ideally stated it inside of the work.

Also, we should say that in this terms exist gray zone, under which we can found following:

- Consulting assignment with others students or people from uninterested parties.
  - We do not see this as a major problem if and only if they are consulting only a small part of the assignment, for instance, if they are stuck or don’t fully understand what they should do.
- Searching for a solution online.
  - This is becoming a problem in last years. Students often can solve assignment by “gluing” together code from multiple posts, for instance on stackoverflow.com, which solve part of given assignment.
- Sharing/taking advice from other students.
  - To some extent, this should not be considered as plagiarism.

Difference between source code and text regarding plagiarism detection

When we firstly started to research this topic, we found out, that many people believe that source code plagiarism can be detected with services that are focused on plain (or formatted) text. We agree that in some cases this method is sufficient, mainly if plagiarist copied the entire source code and only changed few lines of code. However, in many cases, this is an entirely insufficient method of detection.

This insufficiency is based on the main difference between text and source code, and that is how is read (or executed in case of source code). We, as humans, read text from beginning to end, word by word, line by line. We do not jump back and forth between paragraphs, words or lines. Of course with some exception when we skip to some chapter that we are interested in, or we need to re-read some section that we did not understand. Moreover, that is why we cannot simply reorganize or replace paragraphs, lines or even words that easily, without changing the meaning of the larger part.

However, for the most of the source code types, mainly now in a time of OOP[3], are executing differently. Execution in OOP does not care about names (for instance names of the methods or variables), entire bodies of classes can be reshuffled, and it will not change how source code execute. For example, these operations do not alter (almost) anything about how OOP source code is executing or compiling:

- Renaming functions, methods, classes, packages, variables, parameters.
- Adding or removing whitespace characters (whitespace, new line).
- Reshuffling body of classes
    - Switching positions of methods and variables
- Reshuffling body of methods or functions
    - Not every part of body can be reshuffled without changing context

As we can see, we can do a lot more with source code then can be done with text, based on this we will need a much more accurate algorithm that will be able to see through those changes.

> ***Exceptions***
>
> *As partially implied, some types of source code, for instance, procedural programming that dreadfully depends on line ordering, are similar to text and therefore tools for text plagiarism detection are more likely to succeed for them, then for OOP and functional programming.*
> *This was not proven, so take it only as a deduction, based on how detection algorithm works.*

[3] Object oriented programming

Types of plagiarism inside source code

As we can see in Figure 1-1 Plagiarism types, we can divide plagiarism to multiple types. In most cases, we will either see a significant part of code same as original or combination of at least two of others types. This division should be taken only as basic labeling, that will be further extended in next chapters.

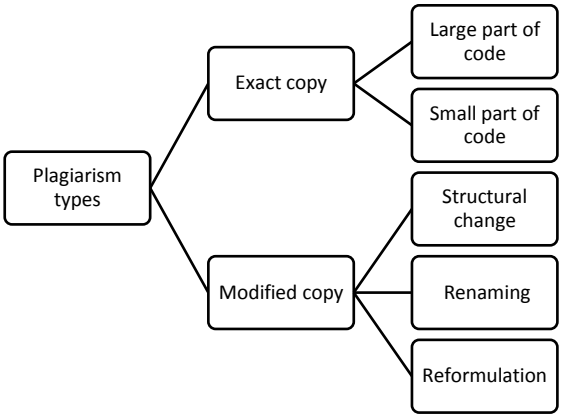


Figure 1-1 Plagiarism types

Magnitude of plagiarism in education

The extent of plagiarism in education is an extremely sensitive topic for both parts, students, and teachers. Students will not admit that they cheat because of fear of punishment, possibly even due to the conscience. Teachers also won´t acknowledge that students cheat during homework our tests so that it would not encourage more students.

As stated in Chapter 3, Section E I had access to students source code that they submitted, and on average there are five to ten students from one hundred that copy another student work. (I do not count approximately another five students that have very similar work to another student, but on closer look, we can see that they heavily edited entire source code.) Based on that, we know that approximately 10% of students use plagiarism on the CTU FEE.

> ***Personal note as a student***
>
> *This is a rather personal note, and I do not want to blame anyone specifically, but each year I see on the final exam someone with the mobile phone under the table. There is a significant lack of surveillance during final exams that would detect and punish them. Alternatively, in some cases, those students are even ignored by supervisors of the exam and by that they get away with it.*
> *As a student, who does not ever cheated during the final exam, this is highly demotivating. Moreover,* ***highly demotivating*** *is an understatement. Some students cheated all the way during college. The only thing that is worst than that is knowing that university is more trying to hide any evidence of cheating students, then punishing them. They care more about public reputation and don't want to take necessary steps like changing results of the final exam or even revoking bachelor/master degrees to punish students that were discovered for cheating later.*

Students source code in contrary to advanced programmers source code

As one of the Czech proverbs states “Zvyk je železná košile.” (In English closes proverb would be “Habit is a second nature”). There is one crucial difference, apart from skill level, between students who started programming and programmer who spend thousands of hours programming.

Advanced programmers have his habits and preferences. Students, on the other hand, does not and their code style and code quality change rapidly. Lack of habits makes almost impossible to detect source code plagiarism based purely on his coding history. So in Barrister we will not focus any afford in this way of detection. (This could still be beneficial for example in companies)

Student plagiarism in opposition to industry plagiarism

As we can see in Annex A most of student´s code is one relatively short file. When they try to copy another student's code they mostly take an entire file and only slightly change it (obfuscated it). When we deal with industry or large project plagiarism, then we can in most cases see an entirely different pattern. In most cases, plagiarism happens on the small part of code.

Some companies take industry plagiarism to the extreme, like for instance Oracle with their lawsuit against Google from the year 2012, where they suited Google for **9 billion dollars** claiming that Google stole **nine lines of code.** Code bellow is the one they say was stolen:

```
private static void rangeCheck(int arrayLen, int fromIndex, int toIndex {
     if (fromIndex > toIndex)
          throw new IllegalArgumentException("fromIndex(" + fromIndex +
               ") > toIndex(" + toIndex+")");
     if (fromIndex < 0)
          throw new ArrayIndexOutOfBoundsException(fromIndex);
     if (toIndex > arrayLen)
          throw new ArrayIndexOutOfBoundsException(toIndex);
}
```

By this example, we want to show how some accusations of plagiarism can be absurd. Therefore the Barrister will only focus on plagiarism where we can say with high certainty that given files are a copy of some other work. We do not want this work to start witch hunts between students and teachers.

Unintentional plagiarism

Over the last few years, we saw huge grown of websites like Stack Overflow that serves for solving problems with programming and many other parts of IT. Plenty of students uses it for discussing problems with source code, searching for solutions, learning, and general discussion. Community on this website mostly refuse to solve their entire homework’s and quickly remove posts that are asking for this so students cannot get quick solution for their homework, but there is also another problem. Moreover, that is if we look at the source code as an instrument for solving tasks, that we can split to numerous sub-tasks. For instance, if some task should solve the mathematical equation and write it to file, then we could have two smaller sub-tasks calculating mathematical equation and writing an integer to file. This sub-tasks could be easily found solved on Stack Overflow. If many students do this, then they source code will be highly likely similar. This type of plagiarism we considered as unintentional, based on the same source where students were searching solutions for some of the fundamental sub-tasks.

1.3 General requirements

General requirements are our expectations from this project and key points for implementation.

Local testing

All testing need to be done locally, so students personal information and source code will not leave school to the third party services. Local testing is also necessary from a legal point of view.

Open source

This project should be open source so anyone can reuse it another system or extend it by new ways of a testing. The ideal license would be AGPLv3 [2] because its handle open-source usage for services used in the background of another publicly available software.

Security by obscurity

Some people on CTU FEE objected against open sourcing this system, as it will be utilized for detecting plagiarism on our faculty, mostly because they like the idea of security by obscurity. In other words secrecy of the design or/and implementation as one of the main method of providing security. By this, they hide theoretical or even actual security risks and vulnerabilities and believe that it is sufficient for preventing a successful attack.[4]

Adjustable

The Barrister should be configurable per each run, preferably via XML file and with default sets of values, that should be ideal for most cases of testing.

Easily parseable results

Results should be easily and speedily readable by another service. Human readability is not the key point of this service.

Runnable from console

The Barrister should be runnable from console or accessible as server service. So it can be usable by another service without manual setup/run by users.

Multithreading

The Barrister should use multithreading for performance gain. Moreover, in future for easier transmission to CUDA calculating.

[4] Personally I don't believe that for the most of systems used in an education it is a good idea. Yes there is greater risk of exposing vulneraries, but also students can help to develop and maintain these systems or learn from them.

I would rather experience successful attack against mine system and have an open discussion with student who successfully penetrated security, without any punishment for him (if he didn´t try to hide the facts that he attacked the system), then hide everything, be closed to discuss problems with system and hope that nobody is abusing some of the bugs.

Low requirements on system

The Barrister should have low requirements and manage to perform with limited resources. Low requirements are needed because Barrister will run as a service in the background, possibly multiple instances of the Barrister will be running at the same time.

Multiplatform

The Barrister should be runnable on main platforms like Windows, Linux, and Mac without any difference in test results. Multiplatform support should be ensured by using strctfp parameter for all floating points operations and united access to file system.

2 Analysis

The analysis is a foundation of every successful project. It helps us determinate possible risks, understand what is excepted from the project and how we should proc in fulfilling this excepted functionality.

At the beginning of this chapter, we will find the small survey on reasons behind plagiarism. Section B is about methods for detecting plagiarism inside the source code. Section C gives an overview of the current services that are used for detecting plagiarism. Section D is this analysis of student codes from several courses on the CTU FEE. Moreover, last two sections are about major problems in detecting and various techniques that students use for obfuscating plagiarism.

2.1 Survey on reasons for plagiarism

Before the actual start of the analysis, a small survey was performed to determinate reasons why students plagiarize source code. Expected benefits of this survey are a better understanding of students who plagiarize and why.

The survey was performed in the form of a personal interview between students of CVUT FEL; all participating students wanted to remain anonymous. 15 students that conveyed survey confirmed that they cheat. In table and chart below we can find results.

Reason	Student count
Stuck on some problem inside code	6
Stuck on problem with testing server	4
Not knowing how to solve assignment	2
Not enough time	2
Laziness	1

Table 2-1 Survey results

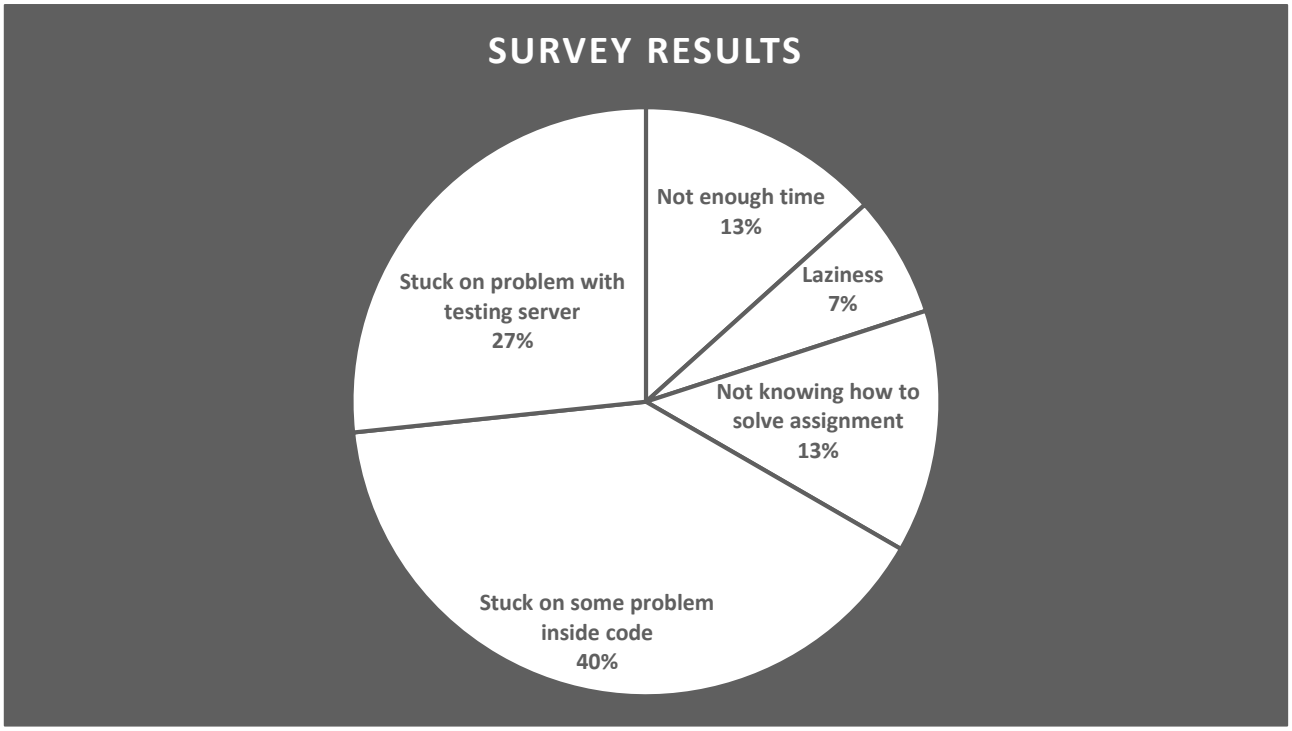


Figure 2-1 Survey results

Stuck on problem with testing server

From my personal experience is this specific to a few courses on CTU FEE that are running on old or insufficient testing services, require specially adapted students code to function properly. It would help to extend documentation for that homework.

Laziness

Laziness should be self-expletory. The student was too lazy or too disinterested in the course to even try to complete homework and immediately went to find the easiest way to complete the assignment.

2.2 Methods for detecting plagiarism in source code

All current methods of detection are based on methods that were firstly developed to for different purposed. Mostly for comparing texts or others forms of data. The University of Sheffield written several publications that were also used for improving search algorithm in Google. [3]

Passive metrics

Passive metrics are used with source code file without executing source code. We simply parse the file and test it via these metrics.

- String similarity
- Token similarity
- Variable counting
- Operator counting

Active metrics

Active metrics need to see source code during entire execution. Furthermore, the execution has to be 100% same for all tested files. This metrics can be tricky especially for speed measurement because it is hard to ensure that any other application will not use system resources during the test.

- Speed measurement
- Memory footprint
- Call graph
- Number of loops and conditionals

String similarity

For string similarity, we can use several algorithms. Most commonly we use Hamming distance or Levenshtein distance. That are basic and see how many bits were changed. For comparing small parts, for instance, comments from source code is also used Sørensen–Dice coefficient. This coefficient has several advantages over other ones. Is often used in Lexicography as a tool for detecting semantic, syntagmatic and paradigmatic relations. [4]

Token similarity [5]

The token similarity is similar to string similarity with few changes. We do not look at the source code as a bunch of lines and characters, but we replace it for tokens by assigning each part of language syntax own token, for example:

```
Syntax   | Token
If       | 0
Else     | 1
else if  | 2
try      | 3
+=       | 4
==       | 5
```

After that, we do string similarity but with generated token arrays. Those tokens will give us much more precise results, because during tokenization we lose some information from source code, like whitespaces, names of variables and few others. This loss of some information is a good thing because reorganization and renaming of variables are most common practice during plagiarism.

Variable counting

Variable counting metrics is based on the idea that many students only rename and reorganize source code but leave some variables inside classes and methods. same. Based on that we can assume that we will find the same number of the variable of given type in both codes, original and plagiarism. [6]

Operator counting

Similar to variable counting, only with operators instead of variables.

Call graph and Speed measurement

A call graph is a representation of calling relations inside the application. A node represents methods and edges calling between them. The only problem is that using this in plagiarism is very problematic; we would have to plug in testing itself so we can observe these calls. Also, we would have ensured that no random data are processed, so each application behaves same. [7]

Speed measurement coexists with call graph but only cares for how much time each call need to completion.

```
        From            | To                  | Calls         | Time
 Player.Main()  | Car.MoveForward() | 20045 calls  | 0.12s
 Player.Main()  | Car.SteerLeft()   | 693 calls    | 0.54s
 Player.Main()  | Car.Brake()       | 358 calls    | 0.06s
```

Memory footprint

We used outside measurement of program memory usage during execution in a contrary to speed measurement when we measured the required time for each call to complete. We take samples each few milliseconds and create footprint graph.

2.3 Current software solutions for plagiarism detection

Current solutions for Java language are JPlag, Moss, Sherlock, Plaggie, SIM. Currently, only JPlag and Moss are being used in the broader range. Sherlock, Plaggie, and SIM were not maintained in years, and most of the people are avoiding them without giving them a chance. Full information about these software's can be found in 3. We did not include any other because there aren't any or are only forks of JPlag with small to none adjustments.

2.4 Analysis of students codes

We got access to students homework from three different courses (A7B01OMO, A7B36PSI, and A7B01DSA). Totally almost 2000 unique files. Firstly we tested them in JPlag so we could have a better idea which students have matches and are therefore interesting for manual investigation. After that we also wanted to test these files in Moss, but because Moss is external service I have to obfuscate any personal information that could be present in these data. So we created Python script for removing any personal information (names and emails) from names of the files and their content. This python script can be found on attached CD.

During the manual examination, we discovered that some students do not even bother to delete original author name from the file name or comments inside the code. As Ing. Jan Kubr told us some international students with little knowledge of Czech language were also caught because they used in comments perfect Czech language. Other think we found out was that results of JPlag and Moss are almost same with only a few exceptions.

From what we saw only interesting results are over 75% match score, under it, we discover only one or two homework that is possibly plagiarism, but students went a long way in rewriting them so only a few lines remained same. Between 75% and 85% match score, we found only one false positive, other files were plagiarism. Therefore for fully automated testing, we would recommend setting that would mark as plagiarism files with over 85% match score, with manual checkup of marked files I would recommend setting that would mark files with over 70% match score. Also please note that for fully automated testing we need to take in consideration students that repeat the course and therefore is high likely they will use their homework from last time they have taken this course.

2.5 Biggest problems in detecting plagiarism

Code length

The main issue in detecting is the length of homework. Some homework are short and because of that is high like that they will contain very similar code. Ideally, homework should have at least 200 lines of code.

For example, let's say that we give students homework which they can solve through roughly 200 lines of code. Here we can probably (based on homework complexity) say that students who created original homework will have between each other around thirty to seventy percent match. Homework that would require 400 lines of code would create matches between twenty to sixty percent match.

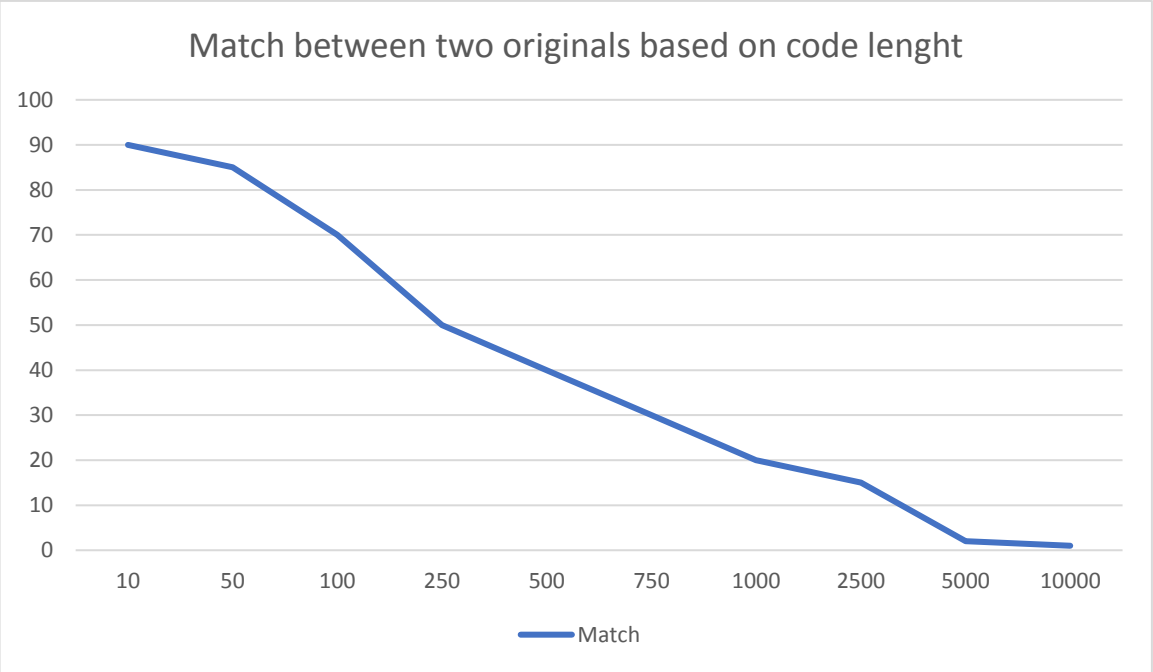


Figure 2-2 Rough estimations of relation between code length and match of two originals

Code complexity

Code complexity is another issue, if students are just starting out with programming they only knew a small piece of language syntax and will produce similar results, then programmers that know different OOP styles and actively use them.

Common parts of code

Common areas of the code can be divided into two groups. The first group includes things like importing standard packages and others language common parts. The second more interesting group contains given the structure of code by assignment. For instance, if we have homework that requires implementing:

```
interface List<T>
{
	public T get(int index);
	public void set(int index T item);
}
```

Then we know that all students will have a class that implements List<T> and also has methods get and set with same parameters and return types. This means that we should have some system that will take this part of the code and remove it.

For instance, Moss remove all code that is present in over 50% of students files from detection. From our point, this is highly dangerous if the student did found out. They would simply all handed in same code knowing that Moss will find nothing. On the contrary JPlag in this situation require access to files with common code and use them to filtering them out of students files.

3 Current software solutions

This chapter is focused on current software for detecting plagiarism. As stated in the last chapter we did not include any other software because there aren't any or is the only fork of JPlag. Firstly we take a look at information about all software´s and then we will compare them against each other. Then on data sets that were specially created for this comparison and then test itself.

3.1 JPlag

License		GPL v3 (Open source)
Supported languages		Java, C#, C/C++, Scheme
Supported version	- Java	1.1, 1.2, 1.5, 1.5dm, 1.7
	- C#	1.2 (One of the first version of C#, Inadequate)
	- C/C++	Insufficient information
Others supported compares		Text compare, char stream compare
Website		https://jplag.ipd.kit.edu/
Source code		https://github.com/jplag/jplag
Version used for compare		7e0efe941a5b7ca966aa5f53ebade33e75203b7c (commit)

Table 3-1 JPlag information

Currently (3.2.2017) JPlag is, together with Moss, one of the best software for detecting plagiarism in Java, unfortunately, have many problems, for instance, all calculations are only one threaded and have an issue with maintenance.

3.2 Moss

License	Unknown
Supported languages	C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, HCL2
Supported versions	Insufficient information
Others supported compares	None
Website	https://theory.stanford.edu/~aiken/moss/
Source code	Close source
Version used for compare	Version from May 18, 2014

Table 3-2 Moss information

Moss is offering by far the largest selection of supported languages for detecting plagiarism. Unfortunately is close sourced and all files intended for testing must be uploaded to Stanford servers. This could be problematic with Data Protection Acts and others laws because we hand over students intellectual property to the third site. Furthermore, is not stated which versions of languages are supported and how precisely their engine work (security by obscurity).

3.3 Sherlock [8]

License	GPL v2
Supported languages	Java, C++, others languages in “basic/text mode.”
Supported versions	Insufficient information
Others supported compares	Tokenization and text control, but with highly complex configuration
Website	https://www2.warwick.ac.uk/fac/sci/dcs/research/ias/software/sherlock/
Source code	On website above
Version used for compare	Last version from the year 2003

Table 3-3 Sherlock information

3.4 Plaggie [9]

License	GPL v1
Supported languages	Java
Supported versions	Insufficient information
Others supported compares	None
Website	https://www.cs.hut.fi/Software/Plaggie/
Source code	On website above
Version used for compare	Last version from the year 2006

Table 3-4 Plaggie information

Plaggie describes itself as highly similar to JPlag with the user interface. However, in last 11 years did not receive any upgrade or maintenance. Therefore for our usage is not favorite. We are interested only in XML output; the user interface is not needed.

3.5 SIM [10]

License	Open-source (unspecified)
Supported languages	C, C++, Java, Pascal, Modula-2, Miranda, and Lisp
Supported versions	Insufficient information
Others supported compares	None
Website	https://dickgrune.com/Programs/similarity_tester/
Source code	On website above
Version used for compare	3.0

Table 3-5 SIM information

SIM is possibly one of the oldest implementation for detecting plagiarism, unfortunately in last twenty years did not receive any update.

3.6 Data used for comparison

For comparison, we created special artificial sets that are focused on many aspects of plagiarism detection. These sets can be found on attached CD.

Artificial set 1

Artificial set 1 is for comparison of detecting capabilities between changed versions of original and original. This set was made from the solution for homework from course A7B36PSI on topic UDP communication. Selected was mainly because:

- Sufficient number of rows
- Commented code
- Subclasses
- Solution use package java.nio.*, which can be replaced with java.io.* without larger need to change the structure of the code. This will allow testing detection of changed technology without changing the structure.
- Changed code can be tested for validity and that they will manage to finish a task.

File	Description
p00_original.java	Original file
p00_original_copy.java	Copy of the original file
p01_without_comments.java	Without comments (further only _wc)
p02_empty_lines_wc.java	Without empty lines
p03_converted_to_one_line.java	Without empty lines and converted to one line
p04_var_renamed_wc.java	Renamed variables
p05_var_and_methods_renamed_wc.java	Renamed variables and methods
p06_method_order_change_wc.java	Changed order of the methods inside classes
p07_method_lines_order_change_wc.java	Changed order of the body of the method while maintaining functionality
p08_order_change_wc.java	Changed order of the methods and their bodies while maintaining functionality
p09_dummy_code_insertion_wc.java	Dummy code inserted to the bodies of the methods
p10_dummy_code_insertion_2_wc.java	Dummy code inserted. New classes and methods (never called).
p11_dummy_code_insertion_3_wc.java	Dummy code inserted to the bodies of the methods and creation of new classes and methods (always called from non-dummy code)
p12_changed_comments.java	Changing content of the comments.
99_rewrite_to_io.java	Rewritten code to the java.io package

Table 3-6 Artificial set for testing 1

Artificial set 2

Artificial set 2 is for testing how well software solutions ignore fundamental parts of code and code from a homework assignment.

File	Description
i00_imp_0.java	These four files contain different empty class but same package imports.
i00_imp_1.java	
i00_imp_2.java	
i00_imp_3.java	
ixx_task.java	Assignment
i01_ imp _0.java, **i01_ imp _1.java**	Structure from assignment, different implementation.
i02_ imp _0.java, **i02_ imp _1.java**	Structure and comments from assignment, different implementation.
i03_ imp _0.java, **i03_ imp _1.java**	Structure and comments from assignment, but both files contains same changes to comments.
i04_imp_0.java, **i04_imp_1.java**	Source code from auto-generation of GUI, same tool (IDEA) used, but both files are original.

Table 3-7 Artificial set for testing 2

Artificial set 3

Artificial set 3 is for comparison of capabilities to test short assignments. Each of these files is considered as "original, " but the source code was still written by me, in other words, a one person that has known all of the other files content. So this should not be viewed from a distance.

File	Description
s00_first_student.java	First original file
s01_second_student.java	Second original file
s02_third_student.java	Third original file

Table 3-8 Artificial set for testing 3

3.7 Comparison

Computer used for testing

Computer part	
CPU	Intel® Core™ i7-4790K
GPU	MSI GeForce GTX 1070 SEA HAWK
SSD	Samsung 850 EVO 500GB
MB	MSI Z270 TOMAHAWK
RAM	32GB
Internet connection	Download 300Mb/s Upload 150Mb/s
Operating system	Windows 7 64bit
JVM	Oracle JRockit JVM

Table 3-9 Computer used for testing

Testing was done on my private computer, that I mainly use for programming, games and in the past for cutting videos. In days when was this thesis written are these specs almost high-end. Internet connection is stated because Moss run only on third party servers. During testing, all

nonessential process was terminated. We also use tuned version of Oracle JRockit JVM, but this should not create a difference in results of the testing. This version used strict garbage collector and modified how and how much memory get each thread.

Test runtime

All test run with strictfp on for every floating point operation. This should create stable results amongst all operating system, JVM, and computer configuration.

Strictfp defines how should floating point operation be executed and how should be floats and doubles stored in memory. Without it, it would be on JVM and operating system to decide how they will handle it. Moreover, it would create slightly different results on different configurations.

```
Strictfp was activated via parameter '-XX+:-StrictFP'
```

3.8 Results

Artificial set 1

In the table below we will find only results that didn't produce 100% match in all software´s.

Subject A	Subject B	JPlag	Moss	SIM	Sherlock	Plaggie
p00_original.java	p09_dummy_code_insertion_wc	94%	95%	100%	/	X
p00_original.java	p11_dummy_code_insertion_3_wc	88%	85%	98%	/	X
p00_original.java	p07_method_lines_order_change_wc	92%	90%	92%	/	X
p00_original.java	p06_method_order_change_wc	86%	88%	91%	/	X
p00_original.java	p08_order_change_wc	83%	85%	88%	/	X

Plaggie

Testing failed without producing any results. This happened because of the unsupported syntax of Java 1.7 and above. More precisely testing failed because of this line:

```
Int a= 2_000;
```

This line of code was contained only in one file, yet testing failed completely, influencing all files.

Sherlock

Sherlock testing is little special, he had similar token test results as JPlag, Moss, and SIM but he also calculated "normalize results" that were around 5% for all tests, after that Sherlock calculated an average value between normalize and token results, creating results around 55%.

Artificial set 2

First subject	Second subject	Match	Description
i00_imp_2	i00_imp_1	100%	Same files.
i00_imp_2	i00_imp_0	<50%	Only imports are the same.
i00_imp_0	i00_imp_1	<50%	Only imports are the same.
i00_imp_0	i00_imp_3	~80%	Same imports, but line ordering inside methods is different.
i01_imp_0	i01_imp_1	<60%	Files are differently implemented and only structure from the assignment is same.
i02_imp_0	i02_imp_1	<60%	Files are differently implemented and only structure **and comments** from assignment is same.
i03_imp_0	i03_imp_1	<60%	Files are differently implemented and only structure **and comments** from the assignment are same. Moreover, comments contain same changes.
i04_imp_0	i04_imp_1	<60%	Two different files only generated by same tool.

Table 3-10 Artificial set 2.i00 Expected results

First	Second	JPlag	Moss	Sherlock	SIM	Plaggie
i00_imp_2	i00_imp_1	100%	99%	88%	No match	100%
i00_imp_2	i00_imp_0	49.2%	79%	73%	No match	43.1%
i00_imp_0	i00_imp_1	49.2%	79%	73%	No match	43.1%
i00_imp_0	i00_imp_3	26.1%	No match	No match	No match	38.1%
i01_imp_0	i01_imp_1	67.6%	58%	78%	77%	23.9%
i02_imp_0	i02_imp_1	65.6%	57%	85%	76%	13.9%
i03_imp_0	i03_imp_1	65.4%	57%	80%	76%	13.9%
i04_imp_0	i04_imp_1	90.7%	92%	54%	98%	82.6%

Table 3-11 Artificial set 2 Results

Artificial set 3

First subject	Second subject	Match	Description
s00_first_student	s02_third_student	<70%	Files are differently implemented but with same technology.
s00_first_student	s01_second_student	<50%	Files are completely differently implemented.

Table 3-12 Artificial set 3 Expected results

First	Second	JPlag	Moss	Sherlock	SIM	Plaggie
s02_third_student	s00_first_student	67.6%	38%	63%	35%	40%
s01_second_student	s00_first_student	0%	22%	0%	65%	0%

Table 3-13 Artificial set 3 Results

Comparing critical points

	JPlag	Moss	Plaggie	Sherlock	SIM
Launchable from console	✓	✓	✓	✓	✓
Active development	✓ (partially)	?	✖	✖	✖
License	GPL v3	?	?	GPL v2	GPL v1
Support Java 1.8	✖	✖*	✖	✖	✖
Documentation	✓	✖	✖	✖	✖
Open source	✓	✖	✓	✓	✓

Table 3-14 Comparison of critical points

* Detected in testing

3.9 Main issues

There are only two problems that were found during comparison of those software. One is highly connected to the fact that three of those software´s weren´t updated in more than ten years.

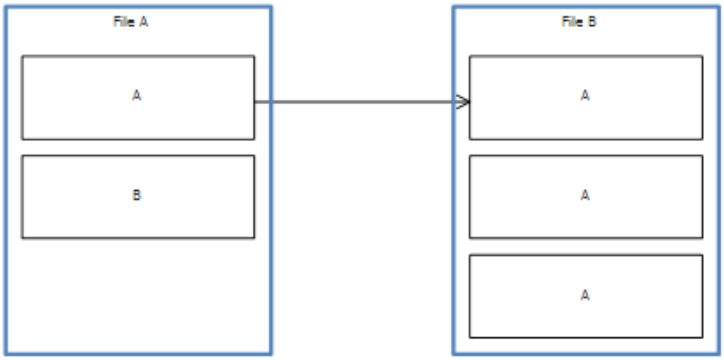


Figure 3-1 Code duplicity

Code duplicity

One of the most serious flaws in all mentioned software is a propensity to ignore code duplicities. In another world, if a student takes someone else code and inserts it three times to his empty file then detection software would found only 33% match. This is possibly exploitable by students.

Version difference

As we can see mainly in SIM results, thanks to 20 years without maintenance Java changed too much, and now SIM gives highly untrustworthy results. Same partially apply for Plaggie and Sherlock.

There are two different approaches how to solve this at least partially. First, is ignoring unknown parts of source code, this method is used inside Plaggie, SIM and Sherlock (possibly even Moss, but without access to source code behind Moss we cannot be sure). As we can see on results, this option is not working that well. The second approach is trying to replace unknown parts of syntax inside source code by anonymous tokens. This approach is used by JPlag and partially solves problems with unknown syntax. Partially because of this only work if changes in syntax between the version of language inside source code and version of language supported by software are small because the software does not know the relation between different anonymous tokens or anonymous tokens and normal tokens.

Uncompilable code

The uncompilable code was not tested because it is not important for this work. Plagiarism testing on CTU FEE is run only on fully compilable source codes after they were tested for functionality inside Moodle or BRUTE.

3.10 Conclusion

In this section, we would like to start with reasons for rejecting each of the tested systems and after that describe reasons for choosing a winner. Mainly because all of the tested systems performed above expectations in the most of the areas and were rejected only for few reasons.

Reasons for rejecting Moss

Rejecting Moss was not easy, all things considered, it would be probably the winner for private testing thanks to highly precise results, but few things made this system inapplicable for CTU FEE, Prosecutor:

- Close-source
- Remote testing on third party site
    - We would have no idea what is happening with students personal information.
- Dependency on third party servers
    - We would not have any control over service availability
    - They could shut down Moss indefinitely
- Inadequate documentation

Reasons for rejecting Plaggie, SIM, and Sherlock

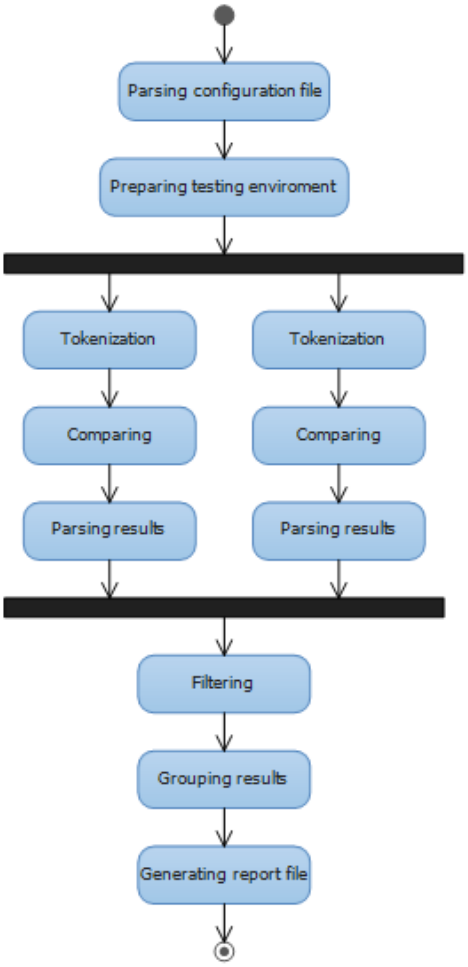
Main reasons for rejecting Plaggie, SIM, and Sherlock, were similar, all of them are pretty old and unmaintained. They first versions were written over fourteen years ago. That would make creating improvements difficult, especially because they use their parser with languages support for versions from that time.

Reasons for choosing JPlag

JPlag and Moss are only one that is actively developed to this day. Unfortunately, Moss is close-source and that leaves us with JPlag, which also have relatively good results.

4 Test lifecycle

Test lifecycle describes how testing proceed from begging to end. So we can see each step and understand what that step do and why is needed in lifecycle. This chapter is closely connected with next chapter about implementation. (Because implementation implement this lifecycle to Barrister)



UML 4-1: Lifecycle of testing in Barrister

4.1 Parsing configuration file

Structure and content of configuration file and his scheme are described in XSD files that can be found on attached CD.

As the first step of the test is parsing provided configuration file, this is accomplished by JAXB library that read the configuration file and parses it to Java classes/entities. These entities are easily reusable inside the entire application.

If the configuration file is not valid against XML schema, that test ends without generating report file; this is one of only two reasons that can cause Barrister not to produce anything and therefore forcing the user to look at console/log output. This is because we need a configuration file to determine where we should generate report file. The second reason for not generating report file is that during preparation step we found out that output location is nonwritable.

4.2 Preparing testing environment

This step can be divided to:

- Testing if we can create and write result file.
    - The optimization that we do not end up after half an hour test with a message saying that we do not have rights to write to this location.
- Setting up strictfp
- Validating file locations stated in the configuration file.
- Preparing multithreading

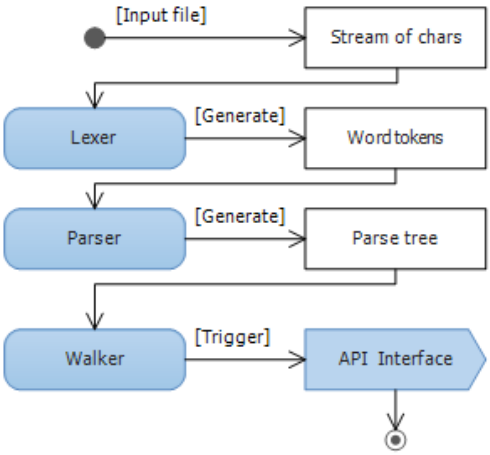


Figure 4-1 Tokenization

4.3 Tokenization

Tokenization is separated into three steps, or four if we count reading the file. As shown in figure 4-1, a stream of characters goes into Lexer that generates word tokens which are sent to the Parser.

The parser then transforms it to parse tree. Some systems use parse tree directly but in Barrister we use an additional step, and that is Walker, which lets us filter out unwanted code.

For instance trailing ";" or:

```
        import java.util.*; //This is contained in almost every student's code in Java. For
plagiarism check, it does not say anything.
```

> *The best way to imagine the difference between Lexer and Parser is that Lexer work on "word level" and Parser work on "grammatical level."*

Example of tokenization

The entire process of tokenization can be somewhat difficult to grasp, especially for users who do not know much about interpreted languages such as Python, or process of compilation for languages like Java, C#, C++. So it will be best to describe the process used to tokenization in Barrister for example. This example is in pseudo-code and only contain few lines for simplicity, but should give us overview over the entire process.

Input file

Let's expect that we have this file, which we want to tokenize:

```
        this.X = 1000;
        X = 1_000;
        X=1000;
        this.X   =    1_000;
        1000 >> this.X;
```

Lexer

Lexer splits file to word tokens. This splitting have many rules, some of them are based on programming language, for instance in most cases comments inside source code aren´t converted to word tokens but instead are stored separately, but in PHP some parts of comments are kept because they act as method/class/variable annotations describing how they should be interpreted. Also, all whitespace characters are removed unless they are inside string. So all lines from our example except for the last one will be converted to this "chopped" array:

```
        X|=|1000|;|
```

However, as stated the last line is different, and Lexer will convert it to this:

```
        1000|>>|X|;|
```

Moreover, that is one of the main reasons why we also need Parser. Others reasons are for instance shuffling operators in boolean logic:

```
        a && b || c
        b && a || c  // This lane has same logic meaning as the first line
        a && c || b  // This lane has different logic meaning than first and the second line
        c && a || b  // This lane also has completely different logic meaning then all above
```

This is commonly used by students to hide plagiarism, as found out by examination of students codes (Appendix A). Possibly because they understand the Boolean logic from others (mathematics) courses, but doesn't understand others parts of programming.

Parser

Parser convert code from Lexer to something that we call parser tree, this tree as showed on figure bellow contains ordered tokens from Lexer in a meaningful way.

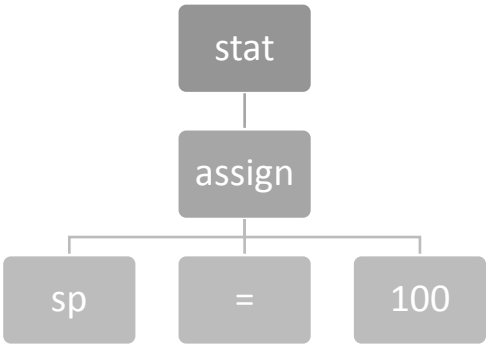


Figure 4-2 Parser tree

Walker

Walker simply “walks” true parser tree and calls API of the external program. Barrister is connected on this level, but he is listening to the only part of calls. For example, we do not listen to comments blocks because we handle them separately. An example of calls for parser tree above:

```
enterStat(StatContext);
enterAssign(AssignContext);
enterExpr(ExprContext);
exitExpr(ExprContext);
exitAssign(AssignContext);
exitStat(StatContext);
```

Future recommendation

This step could be extended by the generation of pseudo-code, that could help in the examination of flagged entities. Pseudo-code would make some similarities more visible and made code readable even for less skilled programmers.

4.4 Comparing

Comparison algorithm is difficult to grasp at first. We need to define global variables, that determinate for which range of matches we are looking for and how precise these matches has to be. Then we move to processing stage, and after that, we also need post-processing to

Variables

Variable	Typical values	Description
min-match	40-60 (%)	Minimal match between two entities, expressed in percentage
max-match	100 (%)	The maximal match between two entities, expressed in percentage. Useful in some scenarios on CTU FEE courses.

sensibility	15-30	How much tokens we need to create a matching group of tokens.
error-margin	3	Maximum of unmatching tokens in the matching group.

Table 4-1 Comparing variables

Processing

Processing that you can find two paragraphs bellow utilize variable from table 4-1 and should give us idea how works. This outline of processing is taken directly from Barrister implementation with only few reductions that serves only to performance optimalization.

As we can see on figure 4-3 bellow this algorithm is optimized by changed order of token testing. After we compare first token we move to position of first token plus sensibility and error-margin and make our way back, when we reach first token we continue from position of sensibility with error-margin forward. This is because we know that some tokens are accompanied by others. Thus tokens after first are more likely same. If we start from position of required sensibility we can faster dismiss non matching parts of code.

1. Put all tokens from file A to stack S.
2. Select token T_{a1} on top of the stack S.
    a. If stack S is empty, then end comparison.
3. Find matching token T_{bx} to T_{a1} from file B.
    a. If such token does not exist, pop out T_{a1} from stack S and repeat step 2.
4. Create matching group M and add matching tokens T_b and T_{a1} to it.
5. Select token T_{an} where n=1+sensibility from file A and T_{bs} where s=x+sensibility from file B.
6. Compare tokens T_{an} and T_{bs}
    a. If they do not match, then increase the number of errors.
7. If some errors are larger than error-margin:
    a. Remove all errors from the end of the matching group.
    b. If some matching pairs of the tokens are larger than sensibility:
        i. Record matching group M
        ii. Remove tokens in matching group M from stack S.
    c. Else:
        i. Remove token T_{a1} from stack S.
    d. Go to step 2.
8. Add a pair of tokens T_{an} and T_{bs} to matching group M.
9. Select tokens:
    a. If n=1 or n>1+sensibility: n=n+1 and s=s+1
    b. Else: n=n-1 and s=s-1
10. If such a tokens do not exist then:
    a. If some matching pairs of the tokens are larger than sensibility:
        i. Record matching group M.
        ii. Remove tokens in matching group M from stack S.
    b. Else:
        i. Remove token T_{a1} from stack S.
    c. Go to step 2.
11. Go to step 6.

Figure 4-3 Comparison order

Post-processing

Post-processing is used to get rid of all references to data and then to create output match object that contains all matches from processing stage to this object we also attach console log that was assigned to this processing.

4.5 Parsing results

This step is asynchronous focusing on calculating the percentage of matches and joining pairs of files (comparisons A->B with B->A). Synchronous is because comparisons of A->B and B->A can happen in different threads at different times. Mainly because of performance tuning in comparing section.

4.6 Grouping results

Similar results are grouped together to form a group of files that possibly have a same origin. These steps in Barrister is very simple by comparing tokens inside matching ranges against each other. We do not try to create a perfect grouping or determine original work in groups. So groups should be only considered as suggestions.

4.7 Filtering

Filtering filter matches above or under wanted limit. There is only one exception for not filtering match outside of this wanted limit, and that is if a match is under wanted limit, but is also part of the group.

4.8 Generating report file

This is a simple step because objects we use for storing information about matches are prepared for storing as XML entities. Firstly we create root DOM element; then we copy root element of the configuration file from the first step as a child element, so anyone can later examine test

condition or even rerun test with same conditions. After that we push inside matches, ordered from highest matches to lowest. Next, we fill necessary metadata like the time when was a test run. Lastly, we validate file via XSD schema.

5 Implementation

This chapter discusses the implementation of the Barrister and is heavily cross-referenced with the previous Chapter, which describes the lifecycle of an entire testing process.

Section 5.B provides the technical information about the Barrister, including the system and software design decisions. Section 5.C outlines the structure of the Barrister, showing the various directories and packages. It reviews and explains the organization and design of the packages. This section also covers the libraries and tools used in this application. Section 5.D provides an overview of the XSD Schemas for input and output files of Barrister. Section 5.E outlines usages of XSLT for XML used in this application. Section 5.F alludes to the multilingual potential of the system and also multilingual handling of files. Section 5.G provides a summary of this chapter.

5.1 Used technologies

This work contains only a few technologies so it could be as much as possible lightweight.

Java

Java was a programming language with biggest community, functionality, and platform support on the planet. In current years Java is under its current owner Oracle in decline. They have serious problems keeping up with others languages, and frankly, all open source projects Oracle bought falls apart.

Antlr

Antlr is leading library for parsing, interpreting or compiling source code in most of the current languages, its supported under C#, Java, C++ and many others. Anyone can extend it by additional functionality and has a stable community for continued support and maintenance.

JAXB

JAXB is one of few libraries that are currently used for mapping Java classes to XML files and vice versa. It also supports the generation of these classes by importing XSD schema file. This functionality was used in Barrister for generating classes for input configuration and output result file. There are few problems through, mainly bad performance rate against native DOM or XPath.

XML

XML is a software- and hardware-independent tool for storing and transporting data.

- XML stands for eXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to store and transport data
- XML was designed to be self-descriptive
- XML is a W3C Recommendation [11]

Benefits of XML

1. Data reuse
    - The same data can be used and presented in much different software. For example, with XSLT we can natively present data as HTML page.
2. Non-proprietary software
    - XML does not belong to a particular company or group of individuals.
3. Unicode

- Multi-lingual
- Interoperability

4. XSD Schema
    - Definition of XML structure and format
5. XSLT
    - Transforming XML document to other XML documents, or other formats.

Downsides of XML

1. XSLT
    - XSLT is becoming quite obsolete and in a few years could be completely unsupported in most of website browsers and systems.

5.2 Structure of the Barrister

This section is heavily focused on technical aspects of Barrister, and without basic programming skills, can be difficult to read and grasp. Also if you are only interested in functional aspects of Barrister, then you can safely skip this section.

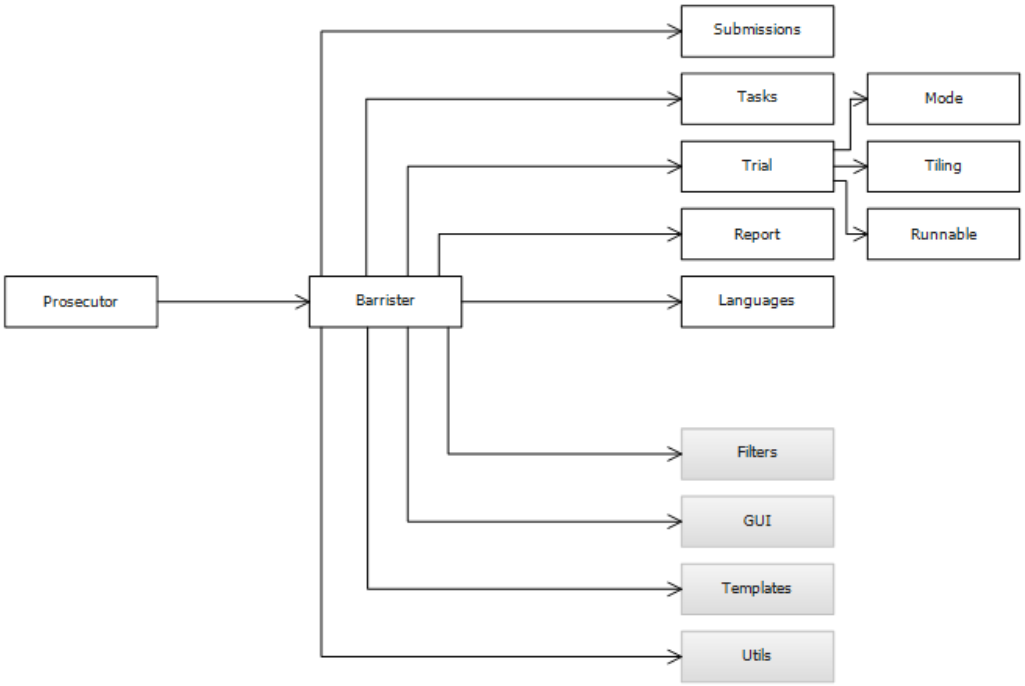


Figure 5-1 Structure of the Barrister

Barrister

package: prosecutor.barrister

The main package of entire software, contains configuration classes and main class that is used for starting Barrister.

Submissions

package: prosecutor.barrister.submissions

This package contains classes for handling entities and their source code files (submissions).

Tasks

package: prosecutor.barrister.tasks

Tasks package contains different tasks that can be executed from the console for instance:

```
        barrister version                 // Calls VersionTask
barrister compare inputConf.xml   // Calls CompareTask
```

Trials

package: prosecutor.barrister.trials

Trials package contains main part of the Barrister, we furthermore divide package to sub packages. Mode for different types of comparison modes. Tiling for comparison algorithms. Runnable for thread wrappers.

Report

package: prosecutor.barrister.report

Report package contains classes handling output file and grouping + sanitization of results.

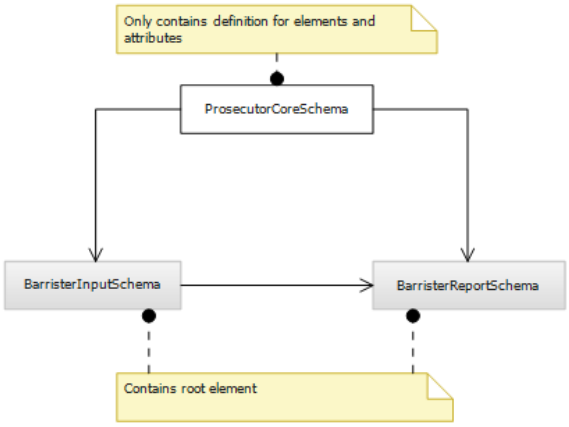
Languages

package: prosecutor.barrister.languages

Package languages provide support for Multilanguage support.

5.3 XSD Schema

The first phase of implementation contained creation of three interconnected XSD Schema that will be used for validation and relational mapping of objects between program and input/output XML files. These were selected because of easement during next stages. In many projects, designers must update XSD schema files and serialization entities inside code simultaneously, otherwise XML files itself will not be valid. Here we use XSD schema for generation of serialization entities.



UML 5-1 Relations between XSD files

XSD Schemas inside project

- ProsecutorCoreSchema.xsd
  - Core schema is containing a definition of objects used in both other schemas.
- BarristerInputSchema.xsd
  - Contains definition for input file.
- BarristerReportSchema.xsd
  - Contains definition for report (output) file.

XSD schema for configuration file

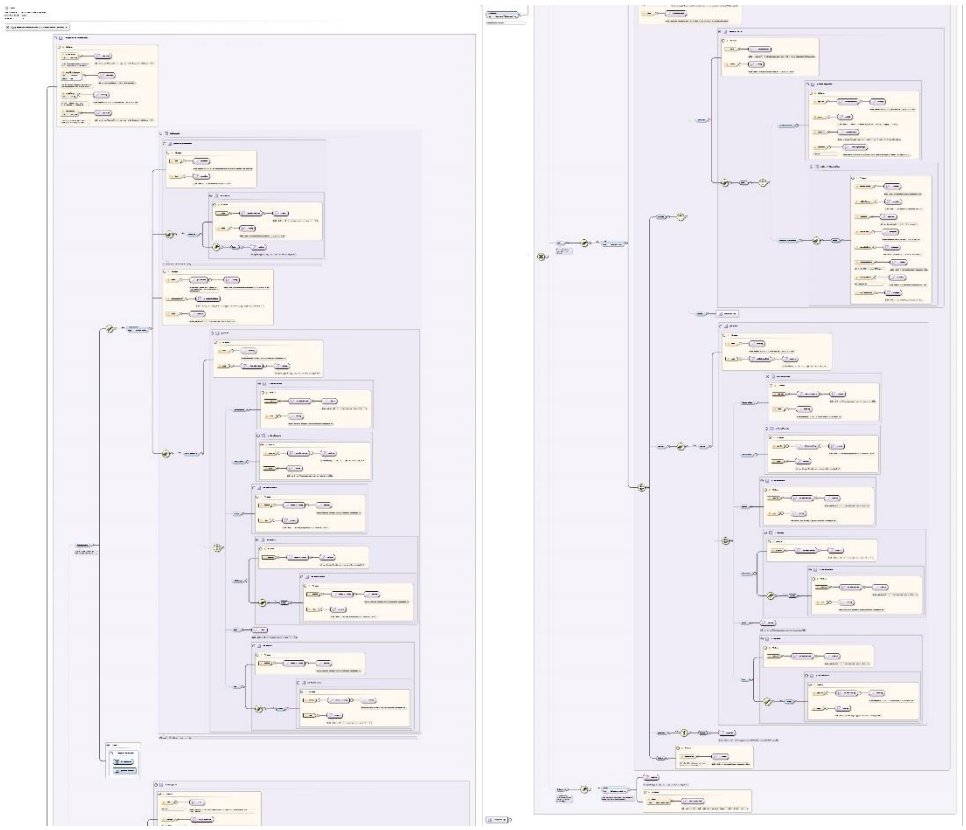


Figure 5-2 Preview tree view of XSD schema for configuration file (splitted in half)

As we can see in the figure above, XSD for the configuration file is quite extensive, this figure can be found in full resolution on CD. So we will focus only on high levels of this schema. Rest is fully described in XSD file itself, that can be found in attached CD. (For displaying comments inside XSD is recommended to use specialized software designed for displaying XSD files, for example, Oxygen XML Editor)

Root attributes

Attribute	Name	Type	Description
outputLocation	Output location	xs:anyURI	Location of result XML. (Indicating file location, not folder location)
outputEntityLocations	Output entities	xs:boolean	If barrister should also output entities and their files to a folder with result XML.
projectName	Project Name	xs:string	Optional, the name of the test. Purely for the visual enrichment of result file. (Has no functional meaning)
rootDirectory	Root directory	xs:anyURI	Root directory/folder for testing, if configuration file uses relative path, then this folder will be used as root. If this value is empty or null, then folder from which Barrister was started is used.

Elements under root

Element	Name	Description
EntitiesLocations	Entity Locations	Location of files that will be compared/tested. Also, contains information which files in this location should be tested and which one should be excluded or handled differently.
Trials	Trials	Describes sets of test which will be run over files. Also, contains information how they should be configurated.
Options	Options	Provides map/dictionary of key and value for setting additional parameters. For instance even parameters for JVM itself.

XSD schema for report file

XSD schema for report file is even more extensive than for configuration file, so you can only found it on attached CD.

5.4 XSLT

You can find XSLT for results from the Barrister on the attached CD. This XSLT transform humanly badly readable XML to HTML website. Unfortunately, this part of the project was not completed because of problems with XSLT inside new versions of browsers and lack of support for XSLTv2 standard.

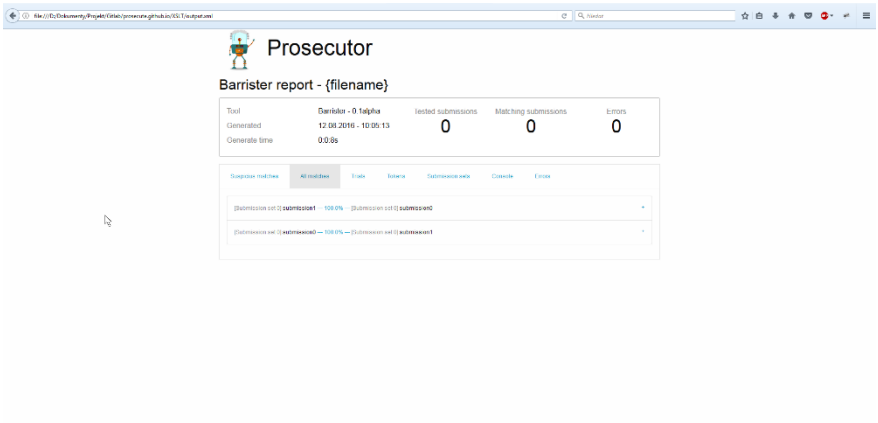


Figure 5-3 XSLT output example

5.5 Multi-lingual support

Currently Barrister support only English, but can be easily extended by providing translated resources to a lang folder inside source code and after recompilation, they will be included to Barrister.

5.6 Summary

Implementation was done without any bigger problems. Thanks to extensity of Antlr and JAXB we manage to easily manage to handle entire input and output process, even for source code files. The only problem during implementation stage was with algorithms used for comparison from JPlag. It is high likely that JPlag itself use algorithms from some other software that were written in C or C++ because of the code structure. We repaired many problems and improved performance.

Our algorithm also takes in count more tokens from source code that algorithm inside JPlag this is based on knowledge of newer language version and newer version of Antlr libraries.

6 Testing

This chapter describes the testing and evaluation of the Barrister performance and functionality.

6.1 Unit testing

Unit Testing is a level of software testing where individual units/ components of the software are tested. The purpose is to validate that each unit of the software performs as designed. [12]

The project contains several unit testing that test main parts of application and integration tests that test workflow of the entire application. Further testing was not performed because acceptance testing were sufficient way of proving complete functionality of the application.

6.2 Acceptance testing

The biggest part of Barrister testing was done on real fresh data from course A7B36PSI on the end of winter semester 2016/2017, together with Ing. Jan Kubr we run several tests over course data and then manually compared each of detected files. By request of Ing. Jan Kubr all student names were blacked out. We also manually removed a few students files that were repeating the course and used same homework solution from last time they have taken this course.

Course A7B36PSI contains two homework; both require the student to write approximately 500 lines of code and have a large number of possible solutions.

Homework about TCP

Firstly we will look at TCP homework, as we will see in tables and figures Barrister found out fewer students, but this was predicted because Barrister is capable of one-way comparison and thus he filtered out matches between old courses.

Jplag	1 student	2 student	3 student	4 student	5 student	6 student
Group 1	[redacted]	[redacted] (99.5%)	[redacted] (96.7%)	[redacted] (76.4%)	[redacted] (62.8%)	
Group 2	[redacted]	[redacted] (96.3%)	[redacted] (80.8%)	[redacted] (61.3%)		
Group 3	[redacted]	[redacted] (95.6%)	[redacted] (94.1%)	[redacted] (86.2%)	[redacted] (79.6%)	[redacted] (71.1%)
Group 4	[redacted]	[redacted] (92.3%)	[redacted] (84.6%)	[redacted] (78.1%)	[redacted] (71.1%)	
Group 5	[redacted]	[redacted] (83.9%)	[redacted] (77.3%)	[redacted] (63.2%)		

Table 6-1 JPlag groups in TCP homework

Barrister	Student count	Students usernames
Group 1	3	[redacted], [redacted],
Group 2	3	[redacted], [redacted],

Table 6-2 Barrister groups in TCP homework

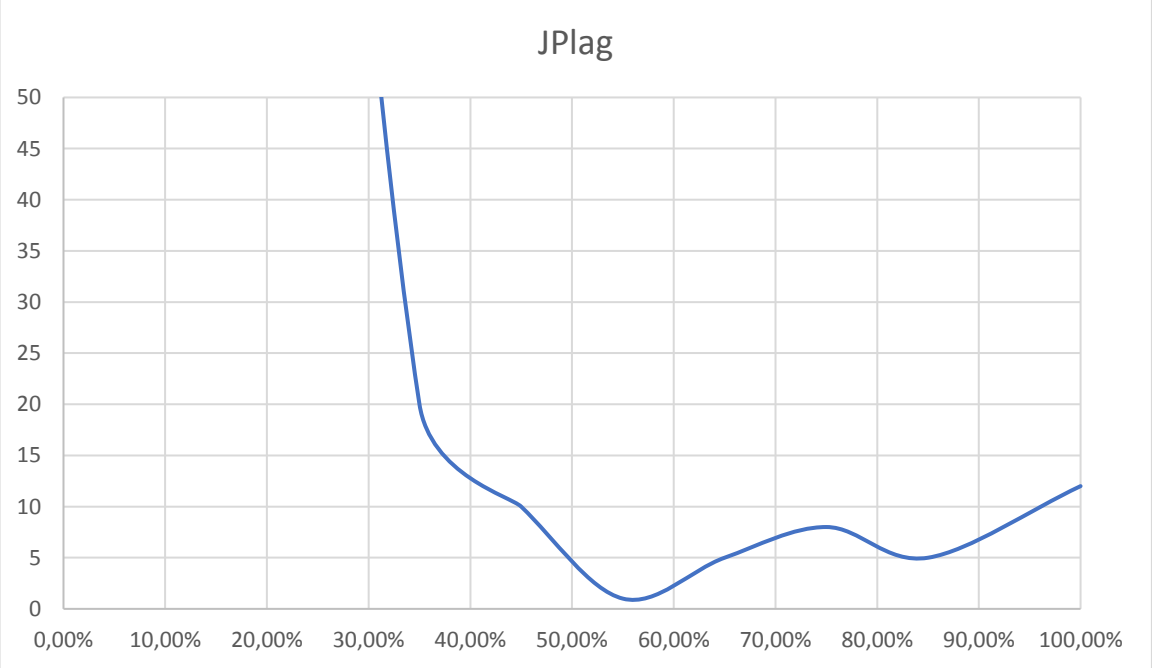


Figure 6-1 JPlag matches in TCP homework

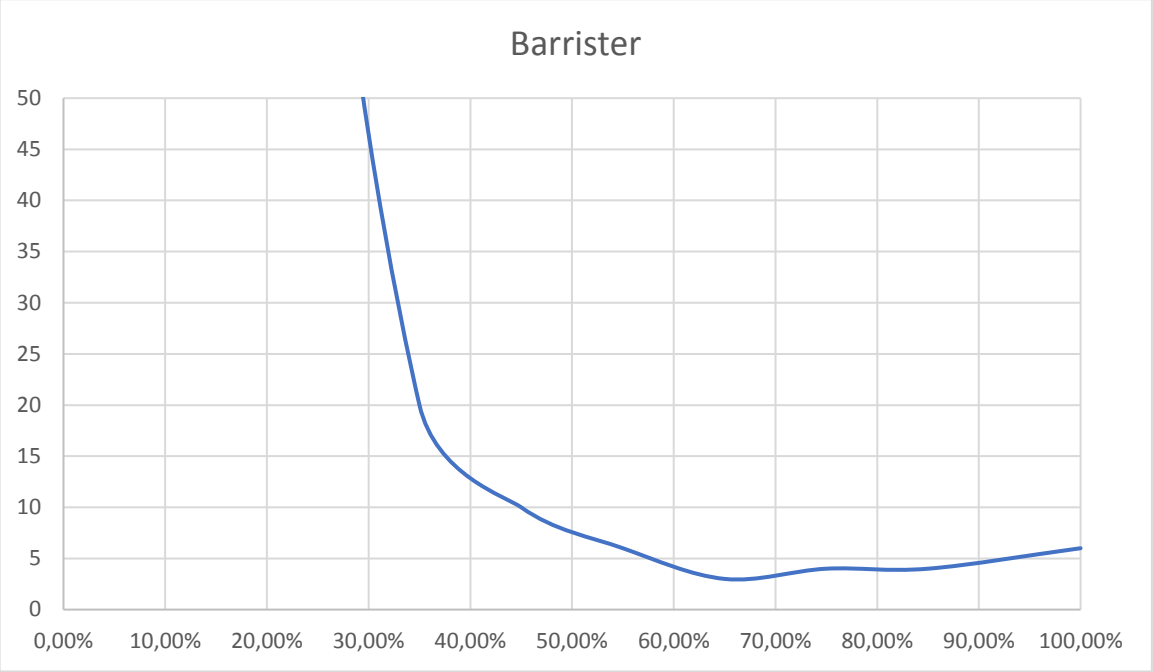


Figure 6-2 Barrister matches in TCP homework

Homework about UDP

Secondly, we will look at UDP homework, results are similar to TCP homework.

Jplag	1 student	2 student	3 student	4 student	5 student
Group 1	[redacted]	[redacted] (77%)	[redacted] (62.2%)	[redacted] (57.2%)	[redacted] (46.4%)
Group 2	[redacted]	[redacted] (59.6%)	[redacted] (55.9%)	[redacted] (55.5%)	
Group 3	[redacted]	[redacted] (54%)	[redacted] (46%)		

Table 6-3 JPlag groups in UDP homework

Barrister	Student count	Students usernames
Group 1	5	[redacted], [redacted], [redacted], [redacted]

Table 6-4 Barrister groups in UDP homework

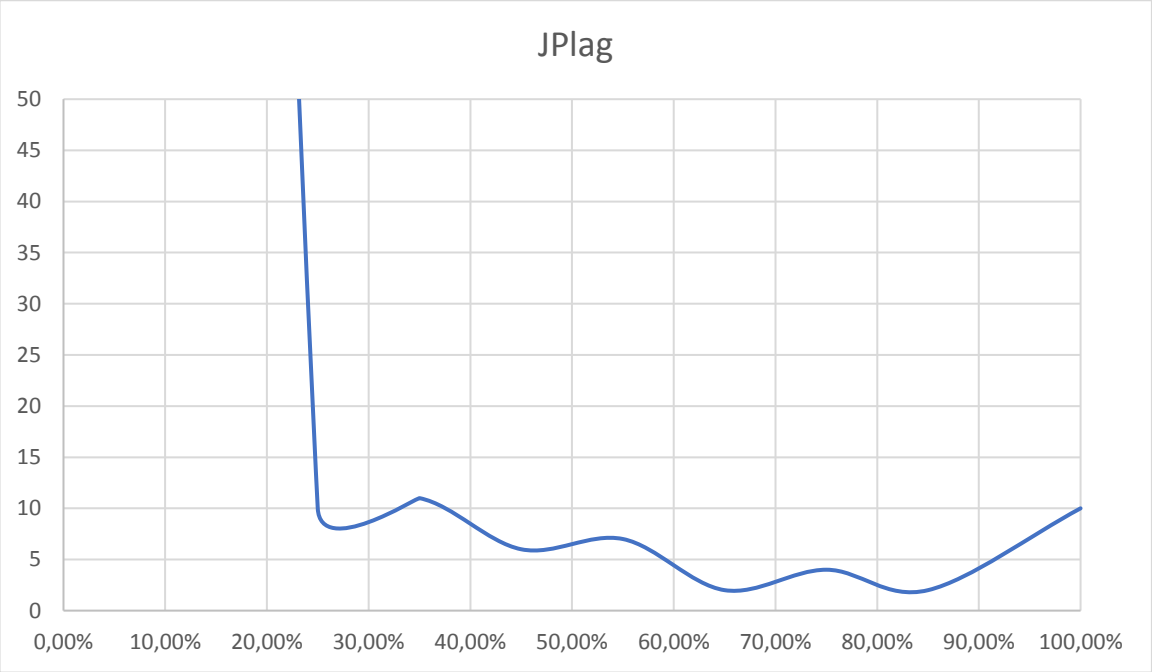


Figure 6-3 JPlag matches in UDP homework

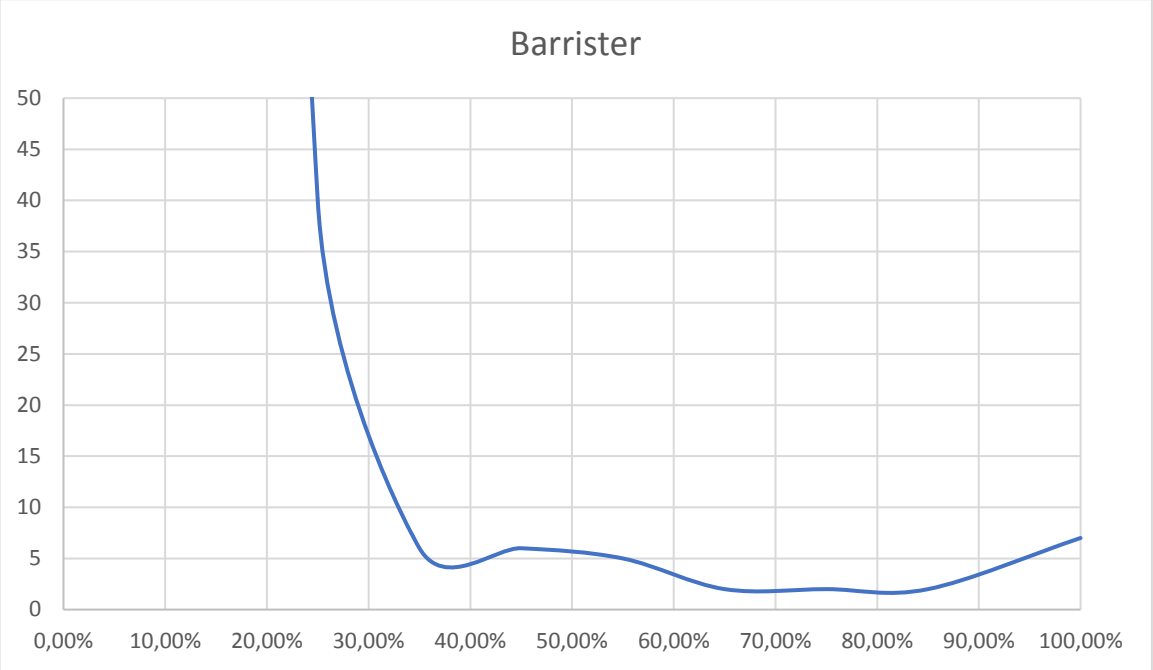


Figure 6-4 Barrister matches in UDP homework

Conclusion

As we can see on Figures there were some changes between JPlag and Barrister results, Barrister filtered out matches between old courses and left only matches in current course and between the current course and old courses. Also, new algorithm filtered out few false positive results with a match between 65 and 90 percent.

Thanks to this users receive more precise results from the Barrister. They don´t have to go thru additional matches from old/different courses and also don´t have to go through many of false positive.

Also, this testing proved one of the expectation and that around 10% students in courses cheat. Moreover, we should be more aware of this problem. The current solution is still very basic, based only on passive algorithms, it is highly possible that if we had an engine that also checks for plagiarism on the internet, we would found even more matches. Alternatively, if we would have an efficient algorithm that would try to transform one code to another while recording approximation time need to do these changes by a human.

It is hard to speculate how many students plagiarize from the internet, but if students know that currently, we can check only their solutions between each other and not against internet sources, it could be another 10%. Together with proved cheaters, it adds up to 20%, and that is untenable. It would be best to make homework more distinctive or let students solve different problems, but this would require more a distinctive approach to each student and therefore more tutors or other personal.

7 Morphing Approximation Algorithm

This chapter is about the proposition of completely different algorithm for comparing two source codes. This algorithm was not and is not one of the goals in this thesis, but during work on the thesis I found out that an algorithm based on morphing one source code to another could be beneficial for detecting software plagiarism. Based on that I included this chapter to the thesis, and I intend to focus on implementing this algorithm in the future. The working title for this algorithm is MAA (Morphing Approximation Algorithm). This algorithm is focused on student´s work and student´s homework and has strict limitations for other use cases.

7.1 Ideas behind MAA

The main idea behind this algorithm is that a student who is using other students source code as its own, have either **lack of time** or **lack of skill** needed to create this source code on their own and they do not want to spend too much time alternating this source code. Therefore steps they made should be easily back-traceable. Because either they made only a few steps or they made steps that weren't too much invasive to the source code (renaming variables, alternating Boolean logic, etc.).

Also in this algorithm, we do not consider a student who spent lots of time, especially if they spent more time that would take writing it alone, as a plagiarist. The same applies to students that used highly complex changes to source code.

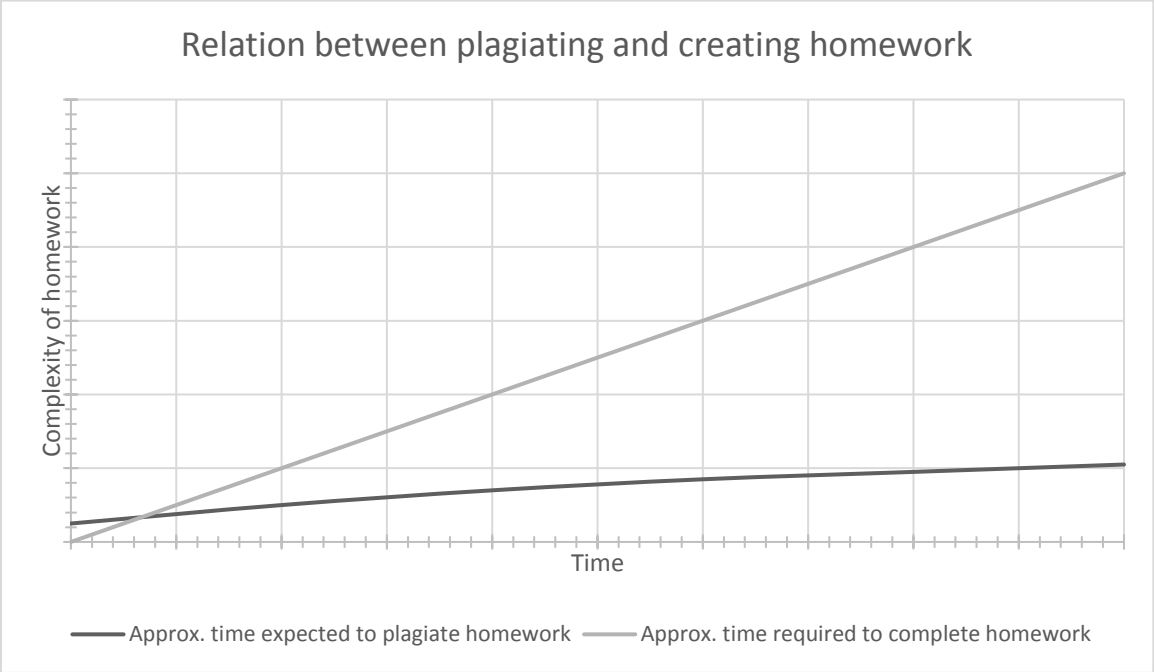


Figure 7-1 Time relation between plagiarism and creating original based on complexity based on anylysis

7.2 Comparison with current algorithms

From first look, you could say that these algorithms are practically the same, but they are not. We can look at current algorithms as passive ones; they only iterate over source code without

any change to its context. This method can be viewed as active one with several loops. Each loop is trying to determinate which steps are needed to do to make one file to another, thus backtracking steps needed to do to create plagiarism.

Advantages over current algorithms

+ Back-tracing steps student make while alternating original work.
+ Estimated time student spent alternating original work.
+ The estimated complexity of steps student makes to alter original work.
+ Can be beneficial in others fields like teaching AI programming or self-repairing source code.

Disadvantages over current algorithms

- More complex algorithm
    - High demands on calculating power
    - Challenging to create
- This algorithm expects that both compared source codes solve same or almost the same task. It will become useless if we want to detect plagiarism only in part of the solution.

7.3 Activity diagram

Bellow you can see activity diagram that would be used by MAA. This is only a first draw of such algorithm so it is high likely that this diagram will change in future.

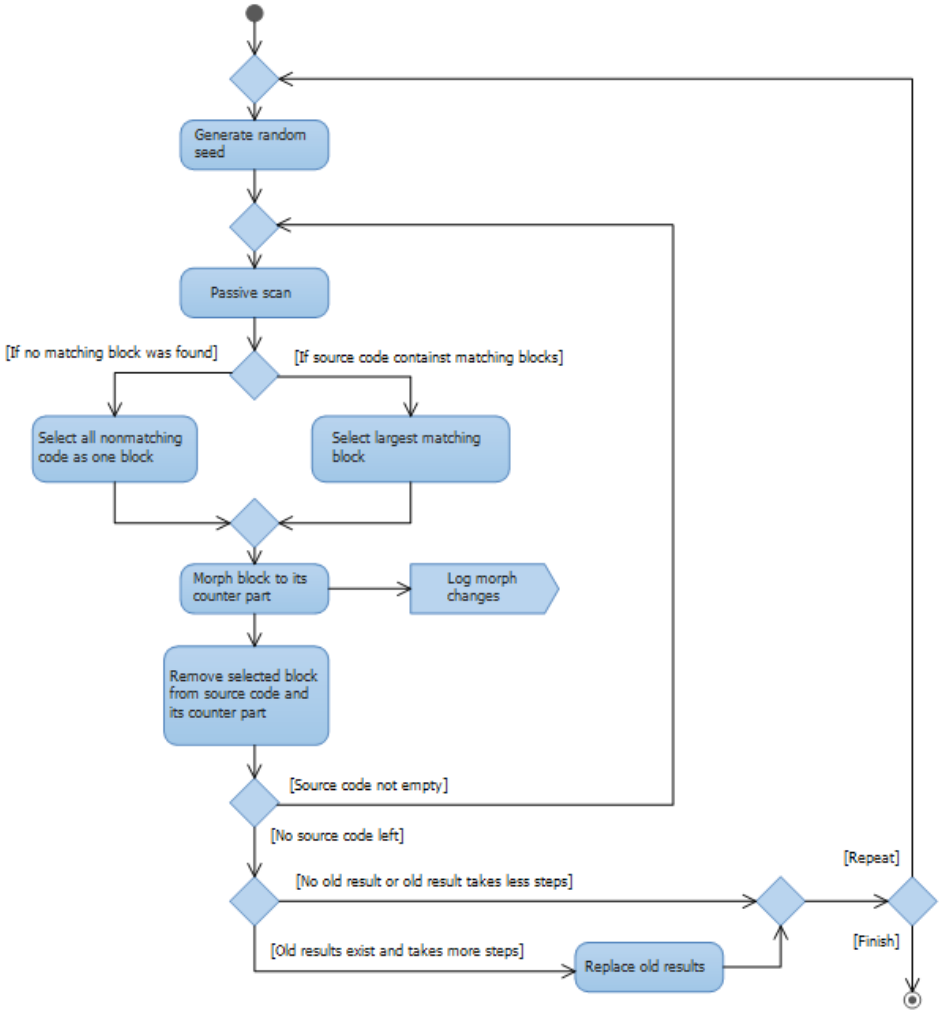


Figure 7-2 Morphing approximation algorithm lifecycle

Generating random seed

We need to generate a random seed, so each pass is unique. Random seed will be used in determining a starting point for the passive scan, etc.

Passive scan

The passive scan is same as scanning for a matching block inside JPlag, Barrister or Moss. The only difference is what happens with these matching block after.

Morph block to its counterpart

Morphing will make all necessary changes needed to make one block from another. All changes are logged.

7.4 Results

As stated before this algorithm would create more precise results and also give us more information about both works. This information can help us determinate how much likely student plagiarize and if he taken enough time during plagiarism to fully understand homework assignment and therefore learn something from this.

Number of steps needed to morph one source code to another

This will also have a time estimate, thanks to this we can determinate if the student is taken the time to create this plagiarism or if he simply copies entire homework and made only slight adjustments.

Complexity of steps needed to morph one source code to another

Complexity can help us determinate if given student was able to do this changes, or if it is more likely that those changes would be to advance for him.

Approximate time needed to create plagiarism

This would be based on several factors:

- Level of the programmer that we can determinate by complexity of steps
- Number of steps required to morph source code
    - Each of the steps would also have different complexity expressed by a multiplier of that given step.

Significance of these results

- If the approximate time to morph source code is larger than approximate time to create source code by himself than we can safely say that this source code is not plagiarism of another.
- If the complexity of steps is low, we can assume that plagiarism is probable.

7.5 Negative impact

This algorithm could be misused to generate plagiarism to a great scales, and only this algorithm could later detect it. This is based on morphing part of the algorithm. It can approximate two codes but also known which steps can be done to not to change the context and functionality of the code.

However, this negative impact could be taken as advantage in others fields, we could simply use it to learn AI or neural network how to create own source code, or we could implement it to programs to create self-repairing code, each time this code would crash, we would rewrite it, adding exception catchers, preventing death-locks, etc.

8 Future of Project

In the long run, I would like to implement a much more complex solution that would fight a lot more efficiently against plagiarism of any kind; this would require the cooperation of multiple universities, organizations and a significant source of funding and professionals that perfectly understand this problematic and furthermore how each programming language interprets.

Our vision of end-game solution would be:

- Cloud and on-premise software.
- Fully written in .NET Core.
- Connected with OWIN.
- Providing RESTful API.
- Offering client libraries.
- The interactive browser of the test result with an explanation for each match and recommendation for how to deal with the result.
- Option to cross-connect universities, organizations, and institutes.
- Comparison against sources from the internet.
- Option for using CUDA cores for faster testing.
- Comparison of source codes, text files, images and most of the others formats.
- The neural network that would distinguish generic code and try to determinate skill sets of the author and time needed to write supplied source.
- Morphing Approximation Algorithm described in Chapter 8.

To this day there is implemented passive detection for source code inside Barrister. Then inside Prosecutor project, there is RESTful API, client library in PHP and interactive browser.

8.1 Rewriting to .NET

This is one of the first steps I will make after submitting this work. The only reason why Barrister was written in Java was that CTU CZM objected strongly to anything else. Even that it would make many things more simple, faster, etc.

One of biggest advantages of .NET is that any programming language that is under CLI (common language infrastructure) is compatible, allowing people all around the world using C#, F#, C++, Ruby, Python, PHP, Pascal, etc. for extending or rewriting Barrister.

> *Java is not under CLI at this moment, Microsoft makes affords by creating J# that was CLI compatible. However, discontinued J# thanks to Oracle threats of lawsuits.*

Another advantage would be performance. In this moment .Net can be translated to native code, making it almost as fast as applications written in C++. Also, it would allow usage of graphical cards, for clarification usage of CUDA cores, for calculations. With current technology, JAVA can utilize only threads on CPU (in most cases 8 threads). However, with CUDA we could utilize tens of thousands threads for a large part of Barrister comparison process.

There are also others advantages as larger platform support that JAVA, OWIN, support for Cloud computing, Timestamping results, etc..

8.2 GUI

One of the things that are currently in development but unfinished is GUI for this application. That would allow users better usability then console. Bellow, we can see previews of GUI.

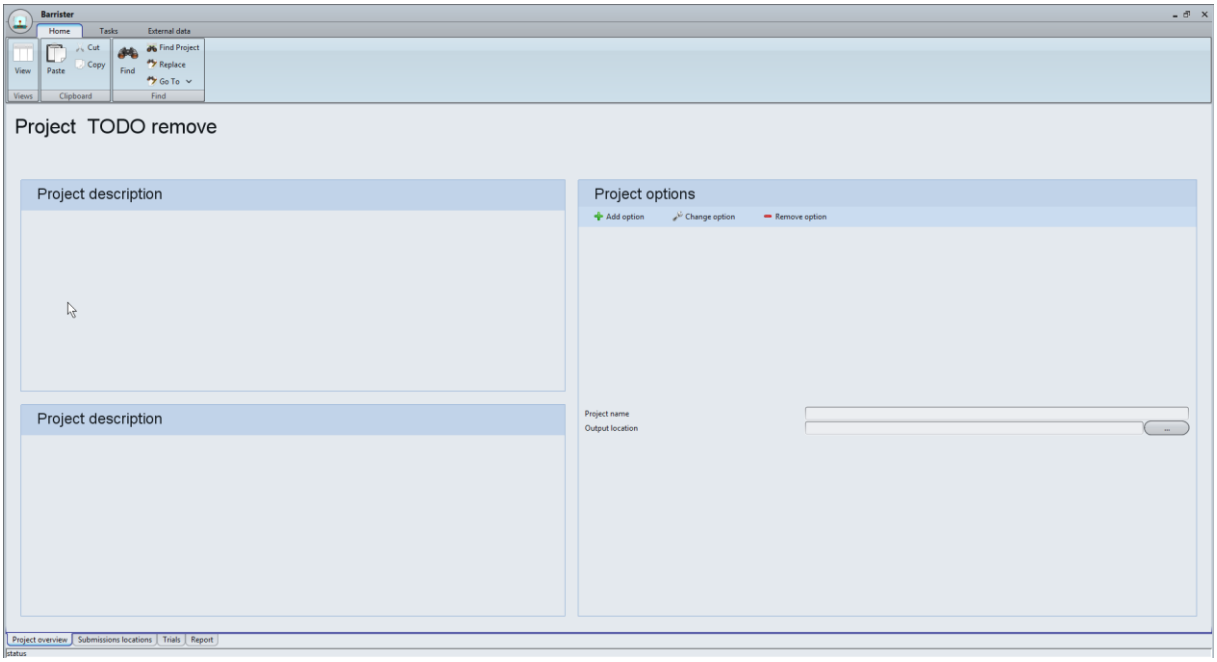


Figure 8-1 GUI

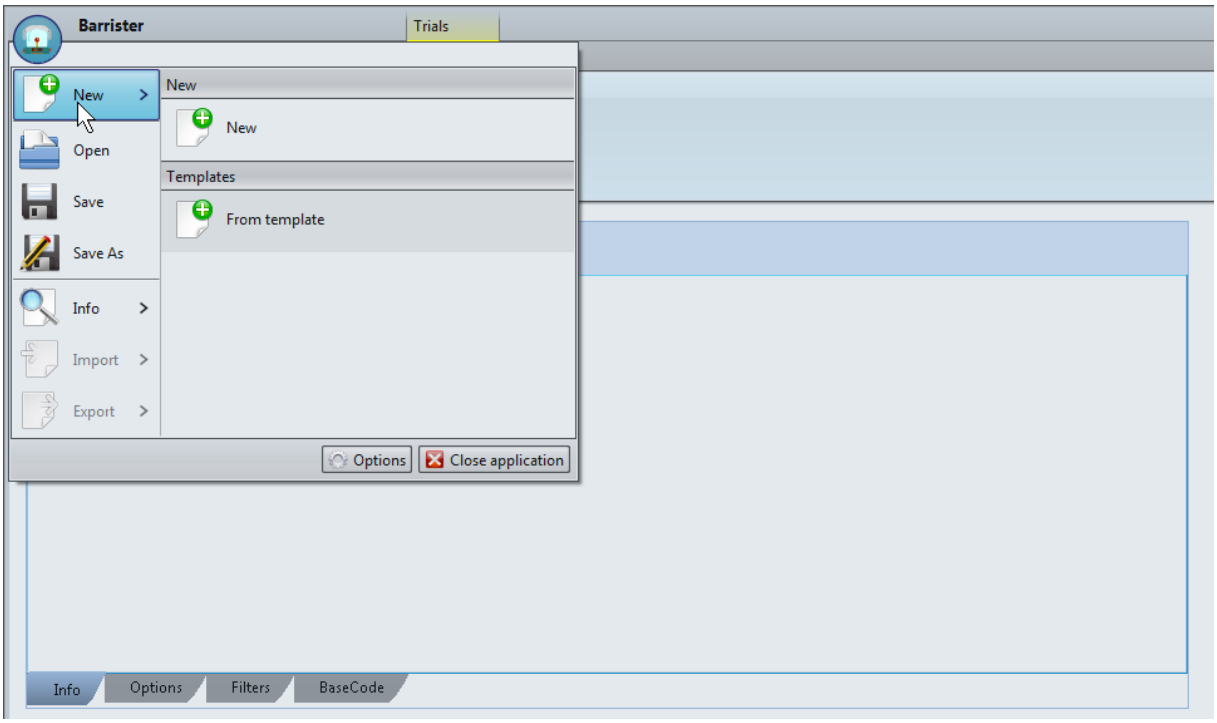


Figure 8-2 GUI 2

8.3 Generating cryptographic timestamp via CA

This would cryptographically prove that results were generated at given time and that no one change those results. Without credentials authority that would confirm generation date and content of results (more precisely confirm hash generated from the content of results), we cannot prove that

someone did not change results at a later point. Currently, not many software use cryptographic timestamping but that also means that they cannot be used as self-standing proves from a law perspective.

9 Conclusion

We manage to develop new plagiarism detection software based on JPlag that solves many problems JPlag has. In the analysis we revealed many of the problems connected with plagiarism that we later settled in implementation. Our multi-threading comparison algorithm utilizes current multi-core processor and therefore provides greater performance.

In a comparison of current solutions for detecting plagiarism we found out that most of them are under-founded and under-developed, most of them did not receive any update in last years.

In testing, we not only tested our application but also helped with detection of cheaters in one of the CTO FEE courses. So we have direct results proving that Barrister can and is helping courses in the detection of plagiarism.

We think that with future work we will be able to create an ultimate solution to this problem. Unfortunately more than anything else this problematic would require founding and creation of task force preferably by U.N., EU or cooperation of several universities that would take it to a larger scale. If we peek outside our scope to plagiarism of Bachelor thesis and others, we can see that students can take their work from different university and no one knows about it.

This bachelor thesis successfully fulfilled its assignment and lead us to think about entirely new plagiarism detection algorithm called MAA and described in chapter 7.

10 References

[1] Smith v. Little, Brown & Co., "Merriam-Webster Online Dictionary," Merriam-Webster, 10 4 2017. [Online]. Available: https://www.merriam-webster.com/dictionary/plagiarize. [Accessed 10 4 2017].

[2] Free Software Foundation, "GNU Operating System," 1 1 2007. [Online]. Available: https://www.gnu.org/licenses/agpl-3.0.en.html. [Accessed 15 4 2017].

[3] N. L. P. Group, "Sam's String Metrics," University of Sheffield, 20 12 2008. [Online]. Available: http://web.archive.org/web/20081224234350/http://www.dcs.shef.ac.uk/~sam/stringmetrics.html. [Accessed 12 5 2017].

[4] D. N. P. M. F. C. F. C. J. van RIJSBERGEN B.Sc., "INFORMATION RETRIEVAL," University of Glasgow, 1 1 1979. [Online]. Available: http://www.dcs.gla.ac.uk/Keith/Preface.html. [Accessed 18 5 2017].

[5] G. M. M. P. Lutz Prechelt, "JPlag: Finding plagiarisms among a set of programs," Universite at Karlsruhe, 28 3 2000. [Online]. Available: http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf. [Accessed 20 5 2017].

[6] E. S. Kshitiz Gupta, "Source Code Plagiarism Detection using Multi Layered Approach for C Language Programs," National Institute of Technology Kurukshetra, 12 12 2014. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.800.4260&rep=rep1&type=pdf. [Accessed 15 4 2017].

[7] G. D. J. D. C. C. David Grove, "ACM DL," University of Washington, 5 6 1997. [Online]. Available: http://dl.acm.org/citation.cfm?doid=263700.264352. [Accessed 10 3 2017].

[8] R. Pike, "The Sherlock Plagiarism Detector," The University of Sydney, [Online]. Available: http://www.cs.usyd.edu.au/~scilect/sherlock/. [Accessed 5 4 2017].

[9] Ohjelmistotekniikan laboratorio, "Plaggie," Ohjelmistotekniikan laboratorio, 8 12 2006. [Online]. Available: https://www.cs.hut.fi/Software/Plaggie/. [Accessed 27 3 2017].

[10] D. Grune, "The software and text similarity tester SIM," VU University Amsterdam, 31 11 1989. [Online]. Available: https://dickgrune.com/Programs/similarity_tester/. [Accessed 20 2 2017].

[11] W3C, "W3C schools," W3C, 20 3 2017. [Online]. Available: https://www.w3schools.com/xml/xml_whatis.asp. [Accessed 20 3 2017].

[12] Software Testing Fundamentals, "Software Testing Fundamentals," 18 4 2017. [Online]. Available: http://softwaretestingfundamentals.com/unit-testing/. [Accessed 18 4 2017].

[13] Oxford, "Oxford Living Dictionaries," Oxford University Press, [Online]. Available: https://en.oxforddictionaries.com/definition/plagiarism. [Accessed 5 4 2017].

[14] Oxford, "Oxford Living Dictionaries," Oxford University Press, [Online]. Available: https://en.oxforddictionaries.com/definition/source_code. [Accessed 4 5 2017].

[15] RKHRamsay, "https://en.wikipedia.org/wiki/Barrister," Wikipedia, 20 12 2001. [Online]. Available: https://en.wikipedia.org/wiki/Barrister. [Accessed 6 4 2017].

Content of CD

- source-code //Folder with source-code
  - barrister // Barrister
  - prosecutor_api //Other software in Prosecutor family
  - prosecutor_php //Other software in Prosecutor family
  - prosecutor_evalviewer //Other software in Prosecutor family
- builds //Folder with builded barrister
- tested_software //Folder with JPlag, Moss, Plaggie, SIM, and Sherlock
- xml
  - xml_schema
    - BarristerInput.xsd
    - BarristerOutput.xsd
    - ProsecutorCoreLibrary.xsd
    - oxygenproject.xpr // Oxygen project for xsd
  - xslt
  - examples
- scripts // Scripts for anonymization
- data-sets // Testing data sets
  - artificial_sets
  - dsa_anonymized
  - omo_anonymized
- documentation
  - Bachelor_Thesis.docx
  - Bachelor_Thesis.pdf
  - xml_schema //Documentation for xml schema