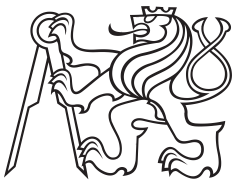
Bachelor Project



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Mutual comparing of source codes

Damir Gruncl

Supervisor: RNDr. Ingrid Nagyová, Ph.D.
Field of study: Open Informatics
Subfield: Software
May 2022

Acknowledgements

Declaration

I do solemnly swear that I compiled the work submitted herein myself and that I did list all used literature.

In Prague, 16. May 2022

Abstract

This work is focused on accurate and traceable detection and displaying similarities of source codes to help detect and prove plagiarism. First, it looks at existing and commonly used solutions, examining their output and the nature of underlying algorithms to see what they can or can not detect and show.

Then it proposes an alternative approach to solve some deficiencies of the examined tools - chief among them the low accuracy and resulting necessity to manually verify results. This is adressed by parsing the compared source codes and using their syntax trees, structure and data flows with the goal of achieving greater accuracy and reliability of results.

Finally, a tool is implemented based on these principles, and its performance is then compared to some existing and commonly used ones, such as Moss and JPlag.

Keywords: plagiate, code comparing, source code analysis, data flow

Supervisor: RNDr. Ingrid Nagyová, Ph.D.

Abstrakt

Tato práce je zaměřena na přesné a trasovatelné hledání a zobrazování podobností ve zdrojových kódech, s cílem napomoci odhalování a prokazování plagiátorství. Nejprve se zabývá existujícími a běžně používanými řešeními, zkoumá jejich výstupy a vlastnosti použitých algoritmů, aby zjistila, co mohou nebo nemohou detekovat a zobrazit.

Dále navrhuje alternativní přístup k řešení některých nedostatků zkoumaných nástrojů - především nízké přesnosti a z toho vyplývající nutnosti ručního ověřování výsledků. K tomu se používá parsování porovnávaných zdrojových kódů s využitím jejich syntaktických stromů, struktury a datových toků s cílem dosáhnout větší přesnosti a spolehlivosti výsledků.

Nakonec je na základě těchto principů implementován nástroj, jenž je porovnán s některými existujícími a běžně používanými programy, jako jsou Moss a JPlag.

Klíčová slova: plagiát, porovnávání kódů, analýza zdrojových kódů, datové toky

Překlad názvu: Vzájemné srovnávání programových kódů

Contents

Figures

Tables

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Gruncl Damir** Personal ID number: **491848**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Mutual comparison of source codes

Bachelor’s thesis title in Czech:

Vzájemné srovnávání programových kódů

Guidelines:

The aim of this work is to analyze the possibilities of current plagiarism detection tools with a focus on detecting plagiarism of program codes and to design and implement a system that can compare two source codes. The purpose of the comparison is to find and mark identical or similar parts of the code. The results of the work will be compared with tools for plagiarism detection in the BRUTE system.
1. Analyze the possibilities of current plagiarism detection tools with a focus on detecting plagiarism of source codes. Familiarize with tools for code obfuscation and their working methods.
2. Design a system for mutual comparison of source codes. Specify modifications to code that plagiarists tend to use and suggest ways how to detect them.
3. Implement a system for comparison of source codes.
4. Analyze the effectiveness of the implemented system. Compare the results with the available tools, focusing on the comparison with the tools in BRUTE.

Bibliography / sources:

1. Pawlik, M., Augsten, N. Efficient Computation of the Tree Edit Distance. ACM Trans. Database Syst. 40, 1, Article 3, 2015, 40 p. DOI: https://doi.org/10.1145/2699485.
2. Votroubek, T. Improving plagiarism detection. Thesis. FEL, ČVUT. 2018.
3. Wroblewski, G. General Method of Program Code Obfuscation. Wroclaw, 2002.
4. Behera, Ch. K., Bhaskari, D. L. Different Obfuscation Techniques for Code Protection, Procedia Computer Science, vol. 70, 2015, pp. 757-763, https://doi.org/10.1016/j.procs.2015.10.114.

Name and workplace of bachelor’s thesis supervisor:

RNDr. Ingrid Nagyová, Ph.D. Center for Software Training FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **07.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

RNDr. Ingrid Nagyová, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Glossary

- Abstract syntax tree - a syntax tree (see 1), but generalised to be independent of grammar or programming language.
- Compound statement - A container that does nothing on its own - that is, does not translate to any CPU instructions - and its sole purpose is to define program structure by grouping statements together. In C-style languages, it is denoted by curly braces: {}.
- (user-defined) Identifier - is any token in the source code that the programmer (user of the language) can define themselves. Typically those are method and variable names.
- Obfuscation - changing source code in such a way that it still has the same function, but it looks different, for example, renaming variables.
- Resistance to (a method of obfuscation) - When a plagiarism detector is said to be resistant to a method of obfuscation, it means that two code snippets, in which the only difference was created by applying said method, will be marked as plagiarised.

- Syntax tree - describes the structure of a program. It contains only the types of expressions used, ignoring their textual representation. Note that the structure of these trees depends on language grammar, and as the same language can be described by multiple grammars, so the same code can have multiple syntax trees depending on what grammar was used. The figure 1 depicts a syntax tree of simple method declaration in C as produced by Antlr:

```
void method(int a,int b){}
```

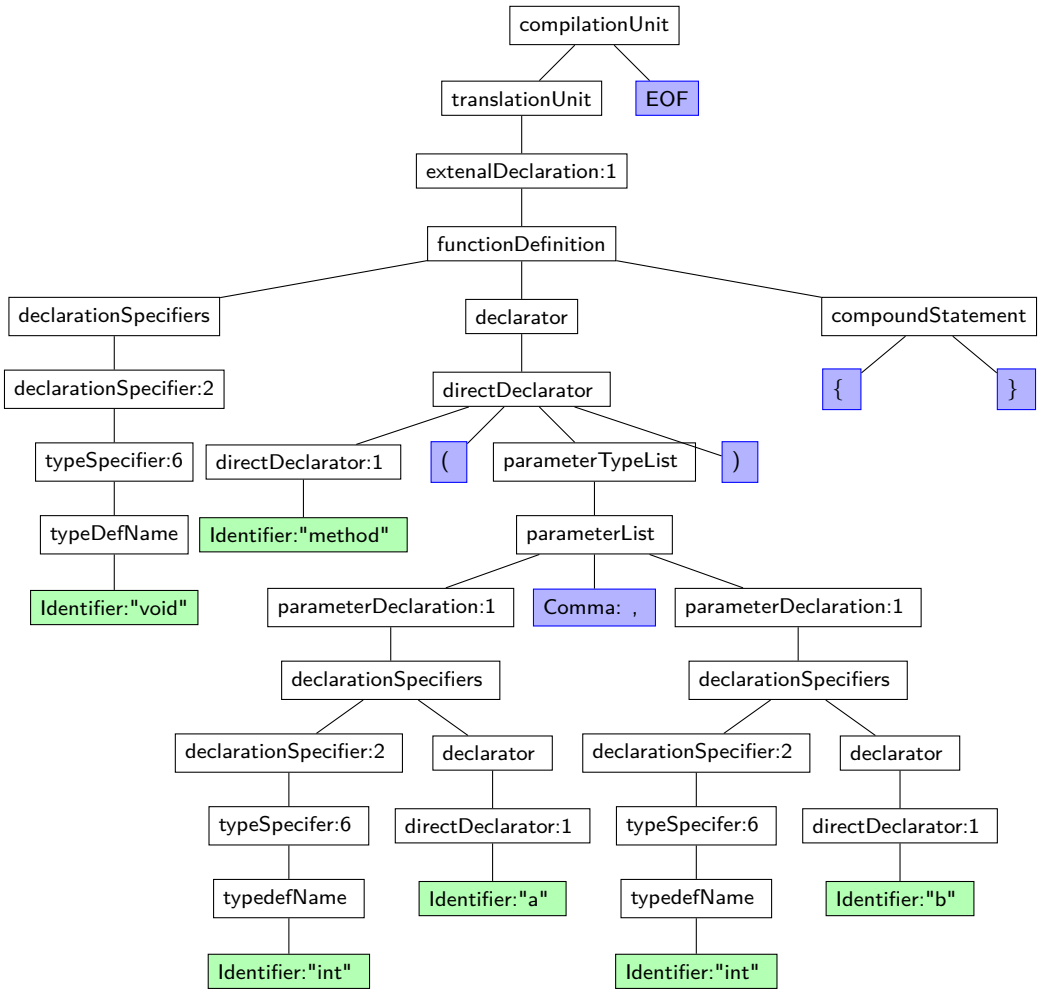


Figure 1: Antlr-generated syntax tree.

- Tree structure of code - source code in almost all languages follows a tree structure - file/class contains methods, which contain expressions that contain operands etc.

- Static data flow analysis - identify inputs of a program - stdio, command line arguments, files opened, etc. - and trace what operations are applied

to data in these inputs. Alternatively. one can start from outputs and trace how they were calculated.

For illustration, the code:

```
void main(int argc,char** argv){
  char* path=argv[0];
  path=strstr(path,"/");
  int x=5;
  if(argc>1){
    int success=sscanf(path,"%d",&x);
    }
  printf(path);
  }
```

might be represented by a graph like the one in Figure 2.

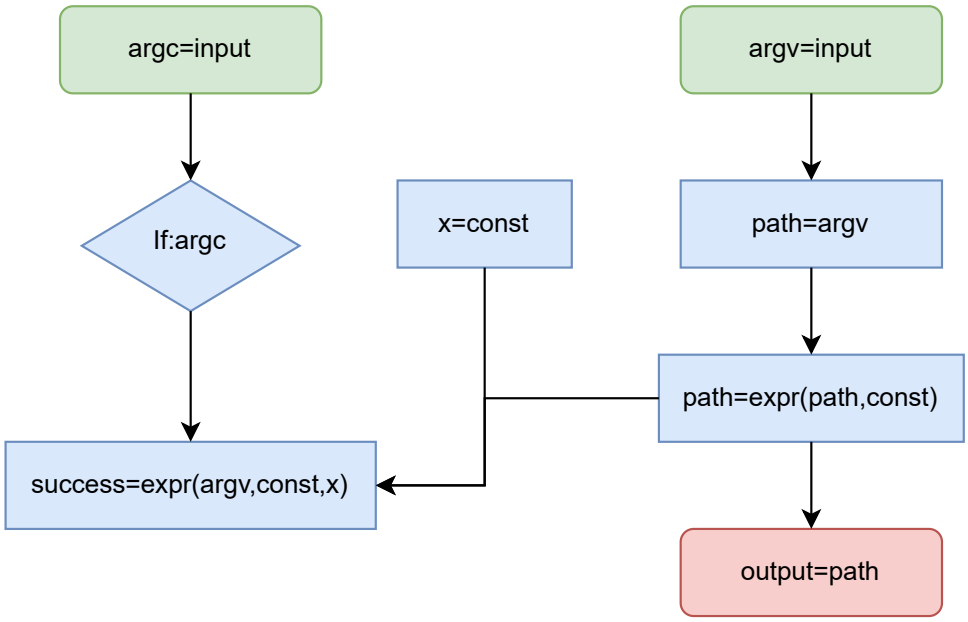


Figure 2: Data flow graph

The graph 2 shows that there is only one path from in (argv) to out (printf), with only two operations on the data; the rest is redundant.

Chapter 1

Introduction

The goal of this work is to design and write a program that can detect and show similarities in source codes as accurately as possible, so as to help in not only detecting plagiarism but also proving it.

In order to establish some context for said design, we will start by defining plagiarism and methods it can use and examining properties of some commonly used tools for its detection.

1.1 On plagiarism

There are two main use cases for comparing source codes:

- Finding differences between versions of the same software. This problem belongs in the domain of version control systems and is outside of the scope of this work.
- Detection of plagiarism, which this work will focus on.

There has been much discussion on what exactly constitutes plagiarism of source code; M.Novak et al. [10] list many definitions, with at least four being fundamentally different from each other rather than variations of the same

principle. The most widely used one essentially says that plagiarism occurs when one uses code written by others without appropriate acknowledgment, thus presenting it as their own work [4]. For the purposes of this text, another useful perspective is
"A plagiarized program can be defined as a program which has been produced from another program with a small number of routine transformations. Routine transformations, typically text substitutions, do not require a detailed understanding of the program" [3].

It is widespread in programming courses due to many people working independently on the same assignment [7]. Dozens of automated tools have been developed for detecting such, with the most popular being Moss, JPlag, and Sherlock[10]. A variety of algorithms is used internally, but invariably the main output is a similarity percentage of the compared files, with files above a certain threshold (around 50-70%) being marked as suspect. It is difficult to objectively define and measure the quality of plagiarism detectors, especially in a single number, but the testing conducted by Tomáš Votroubek [11] suggests that these tools reach reasonably high success rates.

Note the phrase "marked as suspect". An accusation of plagiarism is a serious thing, and naturally, any suspect cases have to be reviewed by an ethics committee or other appropriate authority before disciplinary action can be taken. A simple similarity percentage can not be enough to be certain work was plagiarised; the authors of some tools themselves discourage this[1]. Thus arises the question of how to prove plagiarism, and the obvious answer is to point out similarities in the suspect files.

This would be easy for simply copied works, but it can be assumed that plagiators will take steps to hide the fact that the code is not theirs. This practice is called obfuscation. A search of the extant literature on this topic, most notably[10], failed to yield a comprehensive analysis of what methods are used in real life and how often, but CTU has provided multiple source codes where efforts were made to conceal plagiarism, so the assumption that it does happen is a safe one, and those who seek to detect and prove plagiarism must deal with it.

The aforementioned tools are fairly resistant to obfuscation as far as marking suspicious file pairs goes [11], but another problem arises in proving that, as the person reviewing the suspect files has to find and show the similarities, even though obfuscation methods such as identifier renaming can easily create textually very different file. It would be desirable, then, to have the detection tools aid in this effort by showing the similar parts or possibly even generating a version of the submitted code where the more trivial obfuscations are undone.

1.2 Requirements

Detection of source code plagiarism differs from general text by the specific and exact grammar of programming languages. Commutativity, associativity, arbitrary renaming of user-defined identifiers, changing line count by joining and dividing expressions, adding redundant code, etc. mean that two textually different codes can in fact be structurally and functionally identical. Thus it is not enough to compare code as plaintext; it is necessary to deal with potential obfuscation.

Furthermore, when suspicion of plagiarism arises, it needs to be proven in some manner of disciplinary action - that is, show the identical parts of code. However, as mentioned before, the codes can look very different at a glance, so it is helpful for the detector software to show the similarities as accurately as possible, including any minor changes that might have been made.

Finally, while copying code unusual tokens may be copied, such as words from foreign languages, non-standard compiler extensions, different line ending (combining LF and CRLF in one file), or identical, non-trivial identifiers.

Henceforth, we shall consider two groups of requirements for plagiarism detectors: for detection and for proof.

Detection

A plagiarism detector should mark a pair of files as suspect if they are similar. Of course, this is too vague, and so the second definition mentioned in 1 can be used to clarify: A plagiarism detector should mark a pair of files as suspect if a significant part of them is either identical or contains only formal differences that have no impact on function and do not require understanding the code.

It should be noted that as the files will only be marked as suspect, not declared plagiarised outright, some relaxations of this requirement can be permitted - in other words, the algorithm does not have to be exact.

Following is a list of differences considered formal (obfuscation methods) based on the work of M.Novak et al. [10], who have compiled a list of all

obfuscation methods mentioned in plagiarism research:

- Formatting changes, using characters that do not change program meaning, mostly whitespaces
- Changing strings that do not have an impact on program function, such as identifiers, comments, or output messages.
- Associative and commutative operators (Method parameter separator is also considered commutative here)
- Line swapping.
- Expression join and division.
- Method extraction or merging.
- Addition of redundant lines, or removing optional functionality

Proof

A tool for aiding with plagiarism proof shall mark as suspect exactly those subsections of submitted codes that are either identical or contain only formal differences as per 1.2 and the second plagiarism definition in 1, and only if they are long enough or in such context as to be relevant - standard library methods, for example, are liable to occur many times thorough any code, and so detecting them is of no use unless we also show that they are used in same expressions or on same data.

Unlike in marking suspect files, no relaxation is permitted. The algorithm must be accurate and its result traceable, so if anything is marked as suspect, it must be known why.

It might also be beneficial to be able to revert detected formal differences, the better to show that the codes are same. Detailed statistics of found similarities should also be shown - that is not only the single similarity percentage that most detectors provide[11], but also things like the number of identical identifiers (Among real plagiates used in development there is one where 65 out of 120 identifiers were not renamed), count of formal changes detected, etc. - depending on methods used for comparing.

Chapter 2

Current technologies

We will now look at some existing tools for detecting plagiarism, and how they meet the requirements outlined in 1.2.

Tomas Votroubek, also from CTU, conducted detailed testing [11] of the effectiveness of various plagiarism detection tools in 2018, will be quoted in the following sections, as there is no point in repeating what has already been written. However, his work has focused mainly on the accuracy of marking suspected plagiarism in whole files as per 1.2, while my goal is to help in the proof process (1.2), and thus this work will aim to extend his findings in this direction.

2.1 Test data

To verify the requirements mentioned the following pairs of files were used prepared :

- Real plagiated code ("real1 mod.c" a "real2 mod.c")
- Renaming all variables ("real2 mod.c" a "real2 mod renamed.c")
- Flipping commutative operands ("flip1.c" a "flip2.c")

- Permuting variables ("var falsepos1.c " a "var falsepos2.c ") Note that this is not really a renaming, but permuting - the programs perform completely different calculations, and can hardly be considered plagiarised. They were included in the test suite because supposedly some plagiarism detectors deal with renaming user-defined identifiers by replacing all of them with a constant [1][6], thus losing information and creating a risk of false positives.

In some tools, the length of the files to be compared is limited in the free trial version, so shortened versions were used; this is indicated by the word "short" in the file name.

2.2 Tested tools

2.2.1 Moss

Moss is a program for measuring software similarity. It is free to use and the files to be checked can be sent from the command line, so its use can be easily automated. There are also GUI tools.

The algorithm is text-based. It compares hashes of subsets is order to find (largest) common substring quickly and with resistance to minor changes - that is, the detection is inherently inexact and approximate. Whitespaces are deleted, and identifiers are ignored (renamed to constant). Testing results:

- In a real-life plagiate, it finds 76% similarity (files "real1 mod.c" and "real2 mod.c").
- It is resistant to identifier renaming (97% similarity between "real2 mod renamed.c" and "real2 mod.c")
- Does not detect obfuscation by commutativity (flip1,2.c)
- By ignoring variable names it loses information and creates false positives (96% similarity between "var falsepos1.c " and "var falsepos2.c ")

Moss shows the similarities found in the code, but very roughly, even in blocks of tens of lines. Additionally, it will only show that any two sections are

similar, but if the plagiarism has been masked somehow, it does not remove or mark those changes - it leaves the files exactly as they were uploaded. Finally, ignoring identifiers means that some sections marked as identical are actually false positives. So it will flag suspicious works, but it won't help much in proving it, nor is it meant to do so, as the authors themselves emphasize.

2.2.2 Codequiry

Codequiry is a commercial plagiarism detection tool. It can be used via a web application or web API. There is no specific information on the company website about the algorithm used or its parameters, only a claim that it does more than just text matching and that it is therefore resistant to various obfuscations. It is paid, so the test files had to be truncated to fit within free trial limitations. Testing results:

- In a real-life plagiate, it finds 70% similarity (files "real1 mod short.c" and "real2 mod short.c").
- It is resistant to identifier renaming(95% similarity between "real2 mod renamed.c" and "real2 mod.c")
- Does not detect obfuscation by commutativity (files flip1.c and flip2.c)
- By ignoring variable names it loses information and creates false positives (100% match between "var falsepos1.c " and "var falsepos2.c ")

It shows similarities found in the code, but only in large blocks, often more than ten lines at a time. Additionally, it will only show that some two sections are similar, but if the plagiarism has been obfuscated, it will not remove or highlight this - it leaves the files exactly as they were uploaded. In addition, the code windows are quite small in the web application and cannot be enlarged. It shows graphs and statistics of similarities - variance, graph of distance between files, etc. So from a user perspective, it is like Moss with a nicer but not necessarily better user interface. It does find similarities if they are not well masked, but the usefulness for subsequent proofs is limited for the same reasons.

2.2.3 Winmerge

This is a program for comparing two text files. It only detects and shows exact matches of text, which precisely is what we need, and what other tools did not do. However, it has no resistance to obfuscation whatsoever, and so it is only of use for plagiarism by exact copying, which can be proven well enough without specialised tools.

2.2.4 Other

There are many other tools, e.g. JPlag, PMD-CPD, Sherlock, Sim, or Text::Similarity in Perl, which was at least until recently also used in the Brute package. All of these have been tested for detection accuracy and reliability by Votroubek, who writes: "... none of the tested detectors performs parsing, let alone runs the code...." [11]. He further elaborates that they are all internally text-based comparers, relying on statistical methods usually using hashes and removing user identifiers.

This means that while these tools can flag suspicious files with considerable success, they cannot be used in proof for the same reasons as Moss - they do not work with sufficient detail and accuracy, and they deliberately discard some information.

2.3 Conclusion

There are tools capable of detecting suspiciously similar codes with considerable success - or, more precisely, they detect similar subsections of those codes.

However, the use of some vague similarity in proving plagiarism is limited because multiple people can solve a problem the same way, resulting in textually similar code. What is more interesting are sections of code that are completely identical, or contain only simple formal modifications that require no understanding of how the program actually works.

Obviously, this is exactly the kind of detection that is very difficult, if not

impossible, with approximate matching as used in current tools - we can hardly discern, for example, what is just a formal modification and what affects the function of the program if we don't even parse the code, and we can't tell that half of the user-defined identifiers in the code are identical if we ignore them.

Indeed, it turns out that if the currently available tools show suspicious sections at all, they do so very roughly - often matching blocks of more than ten lines.

It would seem, then, that a tool the likes of which this work aims to create, currently does not exist, or is not commonly in use.

Chapter 3

Design of own tool

3.1 Options

There are three basic types of information that can be extracted from source code (and then compared):

- Text. Generally, this means treating the code as a string and searching for some common attributes, such as common substrings, or tokens that can not be changed by the user - keywords, standard library methods, built-in data types, operators, etc. This is used in most comparers 2.2.4[10]. Common programming languages allow for a lot of formal changes (1.2), so the accuracy of this method is limited.

- Syntactic trees (figure 1), whose structure is exactly defined by language grammar, which makes them easy to work with. Requires parsing the code.

- Static data flow analysis, defined in 2. Parsing the code by its grammar will allow detecting all variables and methods, so running the program is not necessary. This method will naturally be most resistant to formal changes, since changing operations on data without breaking the program requires some understanding of how the program works.

3.2 Methods used

From the outset, the main goal was to create a tool that could show similarities of source codes as accurately as possible and so help in proving foul play, rather than just produce a similarity percentage, as many available tools do. This means focus will be on comparing only two files, usually ones that have already been marked as suspicious by another tool.

Now, there are multiple known algorithms for finding identical subsections of strings, so that is easy - the difficult part is dealing with the differences, detecting what is a formality that should still be treated as same and what is not, and doing so exactly, in a deterministic manner, so that for every code subsection that is marked as same, it is known why, as per 1.2.

The considerations made in previous sections (mainly the requirement of accuracy) naturally led to solving this by parsing the source code and building a comparer on tree structure of source code (described in glossary). Doing so should allow to explicitly define what code is considered same and implement a comparer that only accepts exactly that.

Some algorithms based on syntax trees have been proposed, but those are again statistical in nature; the main difference from commonly used tools is that instead of token strings they compare trees.[6][5] In doing so they might mark suspicious pairs more accurately, as the tree retains more information, but it still does not meet the requirements outlined in section 1.2. Furthermore, the program they have developed does not seem available for testing.

We will now consider how to detect the changes listed as formal in the section 1.2 of this work.

- Formatting changes. Because they do not affect program meaning, parsing removes them.
- Identifier renaming. Text comparing obviously can not be used to detect this, so we are left with data types and syntax trees. If identifiers are merely renamed, without change to the algorithm, then the expressions they are used in will remain structurally same, and so by comparing those the identifiers can be matched again.
- Associative and commutative operators. (Method parameter separator

is also considered commutative here). Here the operands will be children of a tree node, so it suffices to ignore order when comparing.

- Line swapping. If the lines are contained in same compound statement, it is almost same as the previous case, as these lines will be siblings in a tree 3.1. If not, the problem becomes more complicated. Possible (partial) solution is two-phase comparing: nodes not matched by reordering will be flatmapped into a list and then compared regardless of position in code, though that is not very accurate. Note the word almost at the beginning. Code lines can not be swapped arbitrarily, and to compare correctly here, data flow analysis 2would be necessary.
- Expression join and split. Here it must be considered that the result of every non-redundant expression must be written and read somewhere, even if it is implicit, such as in method parameters. Then, expressions can be safely divided and split only if all of its parts will write in the same location. Order of read/write operations must also be preserved, lest program behaviour becomes unpredictable. Thus data flow can be analysed to detect which expressions can be joined or split, and then before comparing convert them into basic state by executing all possible joins or splitting everything into atomic operations.
- Method extraction or merging. Extraction and merging are the same problems viewed from opposite sides, so they can be considered as one. Effectively same as moving lines between compound statements, except in the code where the method is extracted, the same variable will exist under two different names. The correct resolution of this would be to attempt inlining methods that can not be matched. A simpler, though less accurate, option would be to add another comparing stage: take the methods (not compound statements, of which methods are a special case) whose content could not be matched normally and try to match it to unmatched lines in other methods.

So theoretically it should indeed be possible to define and with reasonable accuracy detect most obfuscation methods considered in this work.

A comparer thusly built will be able to show exactly which code subsections will be considered same and why, unlike the approximate text-only methods that are usually used. On the other hand, it requires parsing and modifying the compared files, even though for proving foul play, finding exactly identical sections of the original source codes is most valuable.

Finding longest common subsequences of strings is a well known problem, and an algorithm can easily be implemented. It must be considered, though, that the subsequences can be short, heavily overlapping or repeating many

times, such as variable names, and in the context of an entire file insignificant or confusing. So lines are first matched using methods outlined in this section 3.2, and then common subsequences are only searched for in the pairs of matched lines.

3.3 Algorithm

The files are compared in six stages:

- 3.3.1 Parsing and normalisation
- 3.3.2 Identifier pairing
- 3.3.3 Identifier renaming
- 3.3.4 Line pairing
- 3.3.5 Rematching original file

There is also a data flow comparer, but that is a separate algorithm, and only requires files to be parsed and normalised, otherwise it is independent of these stages.

3.3.1 Parsing and normalisation

Code is parsed to syntax tree as per provided grammar. Identifiers are detected. The code is converted to a tree structure copying the structure of lines compound statements, with standardised formatting - hence normalisation phase. Each line also contains its own tree for handling commutative operators.

Thus the following code

```
void method(int a,int b){
    printf("message");
    if(a==b)
        {
        int c=a+b;
        printf("something");
        }
    }
```

is converted to structure like so:

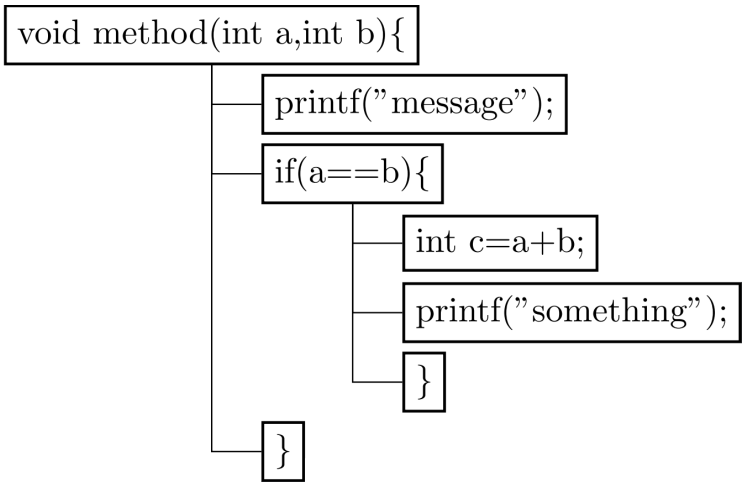


Figure 3.1: Code tree

Henceforth, this structure shall be referred to as *code tree*.

The first line of said code, whose function parameter separator is considered commutative operator, is then internally represented like so:

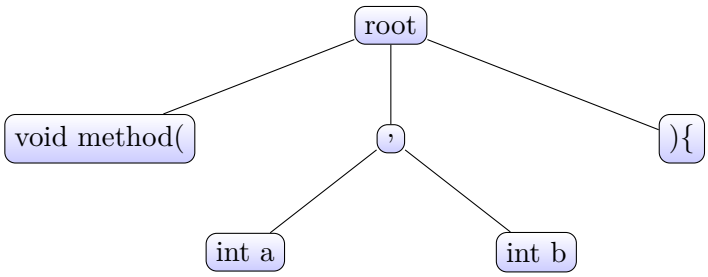


Figure 3.2: Line tree

Henceforth, this structure shall be referred to as *line tree*.

Note that these are indeed two independent tree structures. One is a tree of lines structured according to compound statements, as per Figure 3.1. Each node of that tree represents one line. In the figure, these lines are depicted as simple text for simplicity's sake, but in fact they are complex objects containing various information used for matching, and among this information is another the line tree Figure 3.2, structuring the line according to its commutative operators, so that operand flipping can be detected.

Further information gathered in this stage and saved as part of the line

objects composing the code tree is a syntax tree of the line, the identifiers that it contains and where are they located in the syntax tree.

3.3.2 Identifier pairing

Identifiers are paired by comparing their data types and syntax trees of expressions they are used in. Syntax trees, not line trees, because nodes of line trees contain text - including user-defined identifiers - and those could have been changed, and so can not be used in this stage.

3.3.3 Identifier renaming

Assigning generated names to all paired identifiers. Unpaired identifiers are left as they are. The names should be random, with large edit distances. Line trees of all lines must then be modified to reflect these new names. The reason for this becomes apparent in the next stage.

3.3.4 Line pairing

Iterating through both code trees and pairing the lines using their line trees. The line trees must have same structure except where commutative operators are detected, and their text parts must be within a Levenstein distance smaller than a certain fraction of their length. This fraction is a parameter; here the value of 10% is proposed. The optimal value could be determined by analysing a large enough sample of real plagiarised codes. This stage is done by blocks, where a sufficient percentage of lines within a block must be matched, otherwise, all matches in the block are rejected - thus the matcher gains a rudimentary context-awareness. Currently, blocks are methods. The stricter stages of matching also require that variables within the lines are matched, with the goal of improving context-awareness.

3.3.5 Rematching original file

The objects representing lines are matched with sections of original file that they were parsed from. The parser does not provide this information, so it has to be done separately.

To illustrate, here is the same line of code, first in its original state, then normalised, and finally with identifiers renamed. Note that they are not left-aligned - that is because the indentation has also changed.

```
if(fill_message_buf(msg, msg_buf, size, &length)){
 if ( fill_message_buf ( msg , msg_buf , size , & length ) ) {
 if ( fill_message_buf ( pointer31, integer39, integer37, &
   integer38) ) {
```

Clearly matches found in original state are the most valuable ones for proving foul play, but the modified versions are necessary to resist obfuscation, so we use both.

3.3.6 Common subsequences

Now, for all lines that were paired, the program finds common subsequences in the text they were created from.

3.3.7 Data flow analysis

It has been mentioned 3.2that for some obfuscation methods, data flow analysis is necessary. Also, line pairing 3.3.4compares line to line with minimal regard to context, which decreases accuracy. However, inserting data flow into that algorithm was impractical, and so this is a separate part of the program that can be used once the codes are parsed 3.3.1, independently on results of other stages.

The data flow is generated by breaking program lines into minimal components, called microlines, using an abstract syntax tree, which is essentialy generified version of the syntax tree from Antlr (Figure 1). Each of these can only have one operand, and they are linked together by both their order in

the original program and the way data, contained in variables, flows from one to another. Function call is also considered an operand. Think a graph like in 2, but its elements are also sorted by program order.

Comparing these graphs, while taking into account possible obfuscation, proved rather complicated. After lengthy experimentation, an algorithm was developed that pairs the microlines, but to be considered same, they must not only have same operand, but also all variables used must have been used in same expression before.
For example, last lines of the following codes are not considered same, because in left one, `a` is a result of multiplication, while on the right it is result of division (declarations are ignored in data flow).

```
a=a*b;          a=a/b;
a=a+b;          a=a+b;
```

A disadvantage of this approach is that the first few lines indirectly pair up the variables within them, and if this is done incorrectly, the algorithm will fail to find further matches. For example:

```
a_L=a_L*b;      x=x*y;
x=x*y;          a_R=a_R*b;
a_L=a_L+2;      a_R=a_R+2;
```

Figure 3.3: Counterexample for history-oriented data flow comparing

These codes are obviously same. However, when matching the first lines, the variables in them will have no previous uses, and so `a_L=a_L*b;` and `x=x*y;` will be considered same lines with renamed variables. The same will happen for the second lines in both codes. But then the third lines will be considered different because for them to be same, variables `a_R` and `a_L` must have same history, but variable `a_R` was previously used in the second right line, which is matched to the second left line - and that one does not contain variable `a_L` at all.

That is because microlines are so simple that they are difficult to differentiate without context information - both codes contain two multiplications, and the only difference is in the uses of their operands: `x` appears only once while `a_L` is later used in addition. So to compare microlines properly one has to compare either previous or future uses of their operands. The implemented algorithm defaults to previous uses, because this allows to use dynamic programming approach - when comparing any two lines results of comparing the previous comparisons are available.

This obviously does not work very well on the first few lines of code, as illustrated in 3.3, so if variables used in compared microlines have no previous uses, the future use of these variables is considered instead. Thus the first lines in 3.3 - `a_L=a_L*b;` and `x=x*y;` - would no longer be considered same, because `x` is never used again while `a_L` later becomes an operand of addition `a_L=a_L+2;`

3.4 User interface

Since the main purpose of the work is to help a person decide whether a work will be considered plagiarized, it is obvious that the compared codes should be shown side by side with the similarities found highlighted.

As described in previous section, the comparison is done in several stages, each working with different information (the key domains are matching of variables, lines, and subsequences in the original files). Therefore, the results of each stage should be displayable.

Most stages work with code that is processed in some way, while the user is interested in the original content. Therefore, it would be useful to be able to show, in any given state, the part of the original file from which it was created, for example in the form of a floating window that appears when the mouse hovers over.

Finally, one of the crucial stages is matching identifiers by comparing the expressions in which they occur. It is, therefore, appropriate to allow to show only the expressions in which a selected identifier is present, so that one can see easily whether the match is correct.

Chapter 4

Implementation

4.1 Tools used

The application is developed in Java, because it runs on all platforms, has a native GUI framework, and a wealth of libraries to help with parts of the application generic enough that existing software can be used rather than reinventing the wheel. Two such parts have been delegated to pre-existing tools.

4.1.1 Parsing the code

It is desirable to prepare for eventual support of more languages, and seeing as parsing is both complex and easily extracted into separate component, using an existing tool is obvious. Antlr, an open-source parser generator for provided grammars, is used here. Grammars for all commonly used languages are freely available, and the language they are written in is simple enough to allow for fast and easy modification. It also has a Java version, so it is trivial to integrate. Results of parsing can be extracted in one of two ways - Listener-based or Visitor-based.

Here the former is used, and works like so: Antlr generates a default listener class for given grammar, which contains an empty method for every type of

node defined in said grammar, with that type of node as parameter. Users can extend this class and override any of its methods. Then, upon generating a syntax tree, one can give this tree and listener to a tree walker, also a part of Antlr, which traverses given tree inorder and calls appropriate method from the given listener for each of its nodes.

4.1.2 Comparing syntactic trees

We have special definition of identical trees, where certain formal differences are ignored, and this will necessitate implementing own comparer, but grammars of programming languages are usually extensive and complex, and a minor detail can easily be missed, throwing off the custom exact comparer. Thus, to make the algorithm more robust, a fallback general-purpose algorithm for approximate comparing of trees was included - APTED, which calculates edit distance of ordered trees. Because it compares generic ordered trees, it does not support commutative operators or any other acceptable formal differences. That is why it is only a fallback option.

4.2 Code structure

The core classes of the project are structured by purpose, loosely following the classic MVC architecture, as depicted in Figure 4.1. Yellow blocks are View - GUI, blue corresponds to Control and does the "actual work", that is the comparing, and grey are Model - data structures, which are described in more detail later 4.3.

The comparing algorithms roughly follow the Visitor pattern: they are stateless, with static functions that take data structures representing codes to compare as parameters and proceed to edit these structures to mark the matches found.

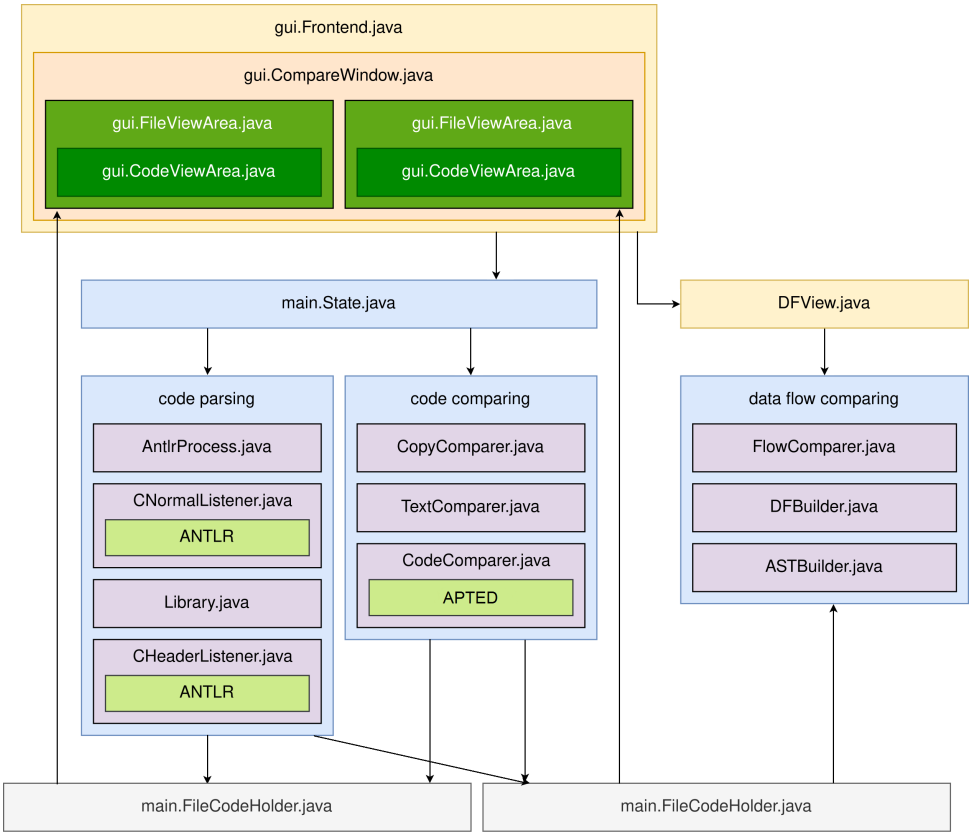


Figure 4.1: Functional blocks

Aside from classes in Figure 4.1, there are classes of the `structs` package, containing data structures that will be expounded upon separately. There are also the `util` and `cmpresults` packages, which contain respectively utility methods and containers for results of comparing stages, and are unimportant.

Now the function of blocks in the diagram will be explained in more detail.

Frontend

Defines and controls main GUI window. Contains two `FileViewArea` objects, each of which displays data contained in a `FileCodeHolder` object. `Frontend.java` sends user commands, such as load file or start comparing, to `state.java`.

State

Manages execution of user commands. That means loading files into `FileCodeHolder` objects and calling the code that runs comparing stages described in 3.3. Note that it does not do any "real work" - it is just a level of abstraction between UI and core code.

Code parsing

Executes stage 3.3.1. Parsing of source code is done using Antlr and the `CNormalListener.java` class is used to gather the results and format them to be used by rest of the program - meaning it generates the code tree (defined in Figure 3.1) and list of user-defined identifiers (`Var.java` objects).

The `Library.java` class contains names of library functions, so that they can be differentiated from user-defined identifiers. These names are obtained by parsing C header files in `include` directives using Antlr, with `CHeaderListener.java` to extract and format results.

Code comparing

The main part of this work. All the classes here are stateless and contain static functions only.

`CodeComparer.java` does identifier matching (see 3.3.2). It contains algorithms for comparing syntax trees as per requirements of 3.2. The main of those algorithms is a simple recursion for checking tree identity, except some nodes are given special treatment, most notably the roots of commutative expressions, where order of children is ignored.
APTED is also used as a fallback. Matching itself is done in stages - for two variables to be considered same, a sufficient percentage of their occurrences must be same. The class contains several functions that define "same occurence of two identifiers" that vary by strictness. Identifiers are then matched all-to-all first by the strictest method, and what remains is then ran again by less strict ones.

`TextComparer.java` does line pairing. (see 3.3.4). Contains an algorithm

for comparing line trees 3.2, which supports commutativity. The text within its nodes is compared using Levenstein distance. Uses stages similar to variable matching.
`CopyComparer.java` finds common subsequences of the original files (see 3.3.6).

FileCodeHolder

Two instances exists, each bound to a `FileViewArea`. They contain the files being compared and all related data structures - mainly the code tree of the file and a list of user-defined identifiers.

DFView

Displays the result of data flow analysis 3.3.7 in a separate window. Uses instances of `DFBuilder.java` to read the analysed code from and calls `FlowComparer.java` to compare them.

Data flow comparing

`ASTBuilder.java` converts a `Line.java` of code into an abstract syntaxt tree composed of ASTNode.java objects.
`DFBuilder.java` maps a list of `Line.java` objects to `MicroLine.java`, with aforementioned abstract tree as intermediate step and stores the result. It also generates and stores accompanying data structures describing the data flow 2 in given code.
`FlowComparer.java` compares the code in given DFBuilders. It should be noted that data flow is strictly separated from other comparing methods. Its data structures frequently point to structures used for regular comparing, but never the other way round.

4.3 Data structures

Found in `structs` and `dataflow` package. Figure 4.2 is a diagram of structures used for basic code comparing - essentially, they represent the code tree 3.1. Figure 4.1 shows the representation of data flows in code.

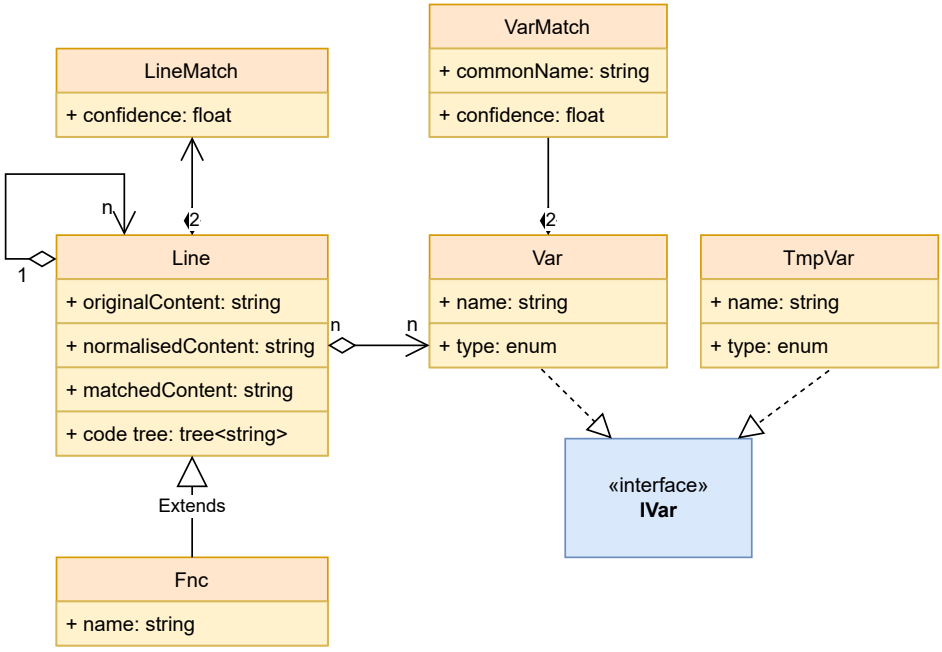


Figure 4.2: Data structures for code comparing

	[IVar] a	[IVar] b	[IVar] c	[IVar] 0	[IVar] 4
[Microline] a=b+c	flow node	flow node	flow node		
[Microline] if(a>0)	flow node			flow node	
[Microline] c=4;			flow node		flow node

Table 4.1: Data flow representation. The flow nodes are interlinked, creating a tree fitted into a table.

- **Line.java** Instances of this class are nodes of code tree (Figure 3.1). Contains line tree, syntax tree, and list of identifiers found in this line (`Var.java`) and their position.
  Has three text representations: original state, then normalised (created in 3.3.1 while parsing), and finally with identifiers renamed (used in 3.3.4). The difference between the three is explained in 3.3.5.
  After lines matching (3.3.4), this structure contains links to their counterparts in the other code tree.

After matching original files (3.3.6), if a line has a match, positions of common substrings in the original text will be added.

- **Var.java** Is a user-defined identifier. Has name, data type a list of its occurrences - `Line.java` instances.
  After matching identifiers (3.3.2), they contain links to their counterparts in the other code tree.

- **VarMatch.java** Used to link together identifiers matched in 3.3.2. During line matching (3.3.4), each instance is assigned an autogenerated name used to undo identifier renaming.

- **LineMatch.java** links together matched lines. However, this is known to contain bugs and may change in future.

- **MicroLine.java** contains minimal lines with only one operator, used to work with data flows.

- **TmpVar.java** is a simplified version of `Var.java`, used for temporary variables in data flow.

- **DFNode.java** is a node of data flow, as in 4.1

- **IVar.java** is a common interface for `Var.java` and `TmpVar.java`, used in data flows.

4.4 User interface

The application is controlled through dropdown menus in top left corner.

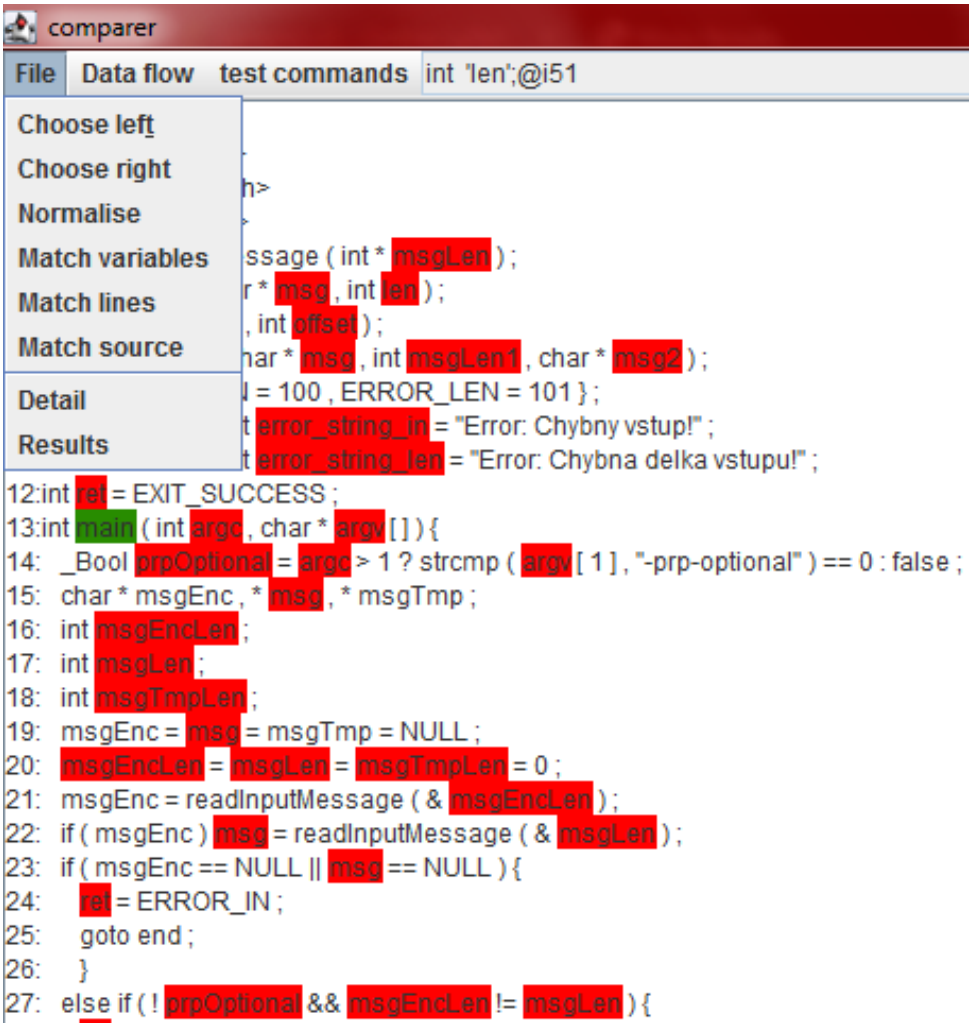


Figure 4.3: Main menu

The items are used as follows:

- Choose left, Choose right - select files to compare
- Normalise - excecutes 3.3.1
- Match variables - excecutes 3.3.2
- Match lines - excecutes 3.3.4
- Match source - excecutes 3.3.6
- Detail - opens another window containing only occurences of selected identifiers.

- Results - show results of executed stages

Variables can be selected by clicking on them; lines are selected by clicking on their number. Selected elements are highlighted in yellow.

Matched identifiers are highlighted in green; selecting one will highlight their match in yellow. Opening the detail window when a matched variable is thusly selected will show only the lines where it occurs in a new window, as proposed in 3.4.
Matched lines will have their number highlighted in green, and selecting one will again highlight its pair in yellow. Images are in the appendix C.

4.5 Data flow display

In the **Data Flow** menu of the main window, **Data flow comparing** will open new window with the selected lines of code disassembled into microlines 3.3.7. If the selected lines are the start of a compound statement (i.e. contain a }), the entire compound statement will be included in the flow. User-implemented methods are inlined. In the new window **Actions->Compare data flow** then starts the matching. **Actions->Compare data flow LMR** does the same but takes into account how the original lines were matched.

Figure 4.4 shows an example of data flow comparing. The files are `HW07_21_91/grep_93.c` and `HW07_21_91/grep_110.c`, entry line is `int main(int argc,char* argv)`. Corresponding excerpt from the former follows after the figure.
Displayed are the two main functions, with inlined calls to user-implemented methods, disassembled to microlines. Paired microlines are written in green and connected by black lines. Also, clicking on a matched microline will highlight its match. The lines of original code from which each microline was generated are shown to the sides, but were cut out in the figure due to insufficient page width.

Figure 4.4: Data flow comparing view. The code on right side follows.

```
void otevrit_soubor(FILE** f, char* jmenoSouboru){
  *f = fopen(jmenoSouboru,"r");
  if(*f == NULL)
    exit(1);
  }
void init_radek(str_radek* mujRadek){
  mujRadek->radek = (char*) malloc(sizeof(char));
  mujRadek->velikost = 0;
  mujRadek->kapacita = 1;
  mujRadek->radek[0] = '\0';
}
int main(int argc, char *argv[]){
  str_radek radek;
  FILE* f;
  int vystup = 1;
  otevrit_soubor(&f,argv[2]);
  int konec_cteni = NE_KONEC_RADKU;
  while (konec_cteni != KONEC_SOUBORU){
    init_radek(&radek);
    }
}
```

Chapter 5

Results

5.1 Testing and comparing with other tools

Having implemented a tool, the time came to ascertain whether it works. The first step of that should be testing of correctness, that the algorithms work as intended. This part was easy because, with the graphical visualization of results, it could be seen at a glance whether the result is correct, which naturally led to test-driven development - a stage of matching could not be declared implemented before it showed expected results on both real plagiarised works (mainly those from folder `sem` in 5.1) and several handwritten files with various obfuscation methods (folder `Votroubek_avoidmoss` in 5.1 are a good example of how such files might look).

The next step was to compare the tool's performance to some similar programs.

Now, as written earlier 2.3, main output of most plagiarism detectors is a single similarity percentage, which is not the purpose of this tool, so the comparison can not be direct. Nonetheless, the percentage of matched lines is similar enough to similarity percentages provided by other tools, which is better than no comparison at all. Thus the similarity percentage of my tool is defined as count of lines matched as per 3.3.4, divided by line count of the smaller file after normalisation 3.3.1.

When deciding what tools to compare with, the work of Tomas Votroubek[11], who is cited several times here, seems a natural choice. However, the provided executable file could not be launched, having failed to open required .dll libraries, and upon compiling the source codes instead, an arduous task that included manually deleting dead links and incompatible attributes in the .fsproj file (for those unfamiliar with the ways of .NET, it is the equivalent of Makefile, but IDE-generated and the user is not expected to ever open it), the resulting executable reported no similarities even in identical files.

Thereby, Moss was used instead, as it is among the best known and performing tools [11][10]. Also, most of the plagiated works provided by my supervisor have a similarity percentage found by the course's detection tool (JPlag [12]).

5.1.1 Testing on suspect datset

The first testing dataset consists of student-submitted codes suspected of plagiarism by faculty members. Table 5.1 contains similarity percentages of all these files(see Appendix A), divided into pairs. In most cases, one folder contains one group of suspect files, with two exceptions:

Folders where name starts with "cross" contains solutions of the same assignment, but from different groups - thus they should not be plagiates of each other, and we expect a low similarity percentage to be found.

The folder "Votroubek_avoidmoss" contains a pair of files created by Votroubek specifically to avoid detection by Moss.

Highlighted cells are those which are within 10% of JPlag result. Again note that this is not necessarily a good thing, especially with my tool, as will be elaborated later 5.2.

The `Verdict` column contains the result of human arbitration, if available: Yes means was declared plagiarised, No passed inspection, Other is that both codes used a third-party source that is not present in this table, typically from the internet. So they are similar, and plagiarised, but not from each other.

Folder	Files	JPlag	Moss	My tool	Verdict
HW05_20_p30	main_30, main_71_47	47	37	31.7	Yes
HW05_20_p30	main_30, main_93_99	99	88	98.6	Yes
HW05_20_p30	main_30, main_137_76	76	41	72.4	No
HW05_20_p30	main_30, main_171_50	50	39	69.5	Yes
HW05_20_p21	main_21, main_37_76	76	7	37.2	Yes
HW05_20_p21	main_21, main_116_48	48	20	37.2	Other
HW05_20_p21	main_21, main_132_42	42	0	42.1	Other
HW05_20_p21	main_116_48, main_132_42	-	4	53.3	Other
HW05_20_60	main_82, main_123	60	50	58.4	No
HW05_20_59	main_169, main_182	59	36	82.4	Other
HW04_20_100	main_98, main_172	100	48	92.3	Other
HW03_20_100	main_159, main_10	100	0	100	Yes
HW02_20_95	main_98, main_172	95	89	95.6	Yes
HW02_20_94	main_193, main_185	94	88	90.4	Yes
HW07_21_78	linked_list_112, linked_list_103	78	0	38.7	No
HW07_21_78	linked_list_103, linked_list_38	78	0	30.1	No
HW07_21_91	grep_93, grep_110	91	48	53.5	No
HW07_21_91	grep_110, grep_159	91	27	55.3	No
HW07_21_90	grep_113, grep_156	90	77	77.6	No
HW07_21_90	grep_156, grep_168	90	57	78.7	No
HW07_21_87	grep_34, grep_117	87	75	64.4	No
HW07_21_87	grep_34, grep_143	87	67	65.5	No
HW07_21_87	grep_34, grep_152	87	77	55.9	No
HW06_21_83	main_118, main_156	83	4	29.6	No
HW05_21_84	main_158, main_5	84	0	77.6	Yes
HW05_21_82	main_69, main_1	82	7	92.9	Yes
sem	prg-sem mod,semestral_project mod	78	79	75.9	Yes
sem	prg-sem mod,main_3 mod	-	74	70.1	Yes
sem	prg-sem mod,my_screen	-	46	49.1	Yes
cross05_21_82_84	main_69, main_158	-	0	14.1	
cross07_21_90_91	grep_159, grep_113	-	32	71.3	
Votroubek_avoidmoss	Avoid_Moss.c,Avoid_Original.c	-	0	88.8	

Table 5.1: Results of comparing plagiated codes using various tools.

5.1.2 Testing on raw dataset

The second testing dataset consists of all files submitted by students in a single assignment, ungrouped (see Appendix A). The goal was to compare all-to-all and see what happens. However, with 113 submissions, that would be too many tests to sort through, and so the files were divided into buckets

by size. The file sizes were around 5kB, and each bucket contained a 300B range. This resulted in much more manageable 488 pairs to compare, of which 119 had similarity above 20%. List of these can be found in attachments in folder containing the compared files, under the name `select_summary.txt`.

Human examination of these files quickly showed that under 40% similarity was negligible - usually the only significant similarity in these files was a function named `read_message` or a variation thereof, which was similar in most of the 119 "interesting" pairs, even if the rest was completely different. It appears that it was a simple function to read input, and a skeleton of it was provided by lecturers.

This eventually left seven notable pairs of files, with the last two being almost certainly plagiated in the author's estimation:

- **a107/main.c, a27/main.c** Similarity: 43%. The `read_message` functions, with circa 25 lines, are almost identical save for a few letters. Even with skeleton of function being provided, this is little strange, but the rest is completely different.
- **a107/main.c, a31/main.c** Similarity: 48%. Same as previous case.
- **a27/main.c, a31/main.c** Similarity: 41%. Same as previous case.
- **a107/main.c, a94/main.c** Similarity: 59%. Despite high percentage, the similar parts are only `read_message` and a few trivial utility function, while the real work of the program is done differently.
- **a18/main.c, a84/main.c** Similarity: 56%. Just as in previous case, the similar parts are only the unimportant parts of the code, but `a84` uses an enum to define constants, but then uses a literal in the code anyway, which hints that some parts may have been copied.
- **a66/main.c, a67/main.c** Similarity: 61%. Here, similarities can be seen throughout the whole code, including shared typos in comments (e.g "chyba vtupu"). Here, plagiarism was confessed, though this was only revealed to the author of this work after having declared it suspect.
- **a102/main.c, a92/main.c** Similarity: 72%. Most of the code is same, with renamed identifiers at most.

5.2 Summary of testing

We can see that percentages reported by the various tools are sometimes wildly different. JPlag is the only one with consistently high percentages, but these are heavily biased seeing as it was used to detect these plagiates in the first place, so that means little. Also, folder *cross07_21_90_91*, which contains files from same assignment in same semester, contains files that were put in different groups by JPlag, meaning that they sohould be different, yet our tool reported 71.3% of matched lines. Review of the files confirmed they are indeed very similar(notably functions `found_you`) and `find_in_line`, where only difference is renamed identifiers. Thus, it can be assumed that the higher percentages of JPlag reports are indeed result of said bias, and thus they provide no information about its performance compared to other tools.

Accordingly, Moss shall be used as a reference, as it has no such bias here. It can be seen that percentages reported by our tool are similar or higher, which serves as a good sanity check.

Comparing the similarity percentages to the final verdict, we see that most of the pairs that were judged to be plagiated have high similarity percentages - usually above 60%, and higher than Moss.

The same can be said for the pairs that were declared not plagiated, which might make them out as false positives, but on human review the reported percentages turned out to be well grounded in reality. For example, the files `grep_156` and `grep_168`, with reported 78.7% of lines matched, are almost identical even in plaintext, which can be clearly seen in the program GUI. It is unknown why they were judged not plagiated, but at any rate it is not a false positive - the program correctly showed the similarities, and judgment is left to human.

Thus it can be said that the developed program successfully detects similarities in code, while showing them with greater accuracy than commonly used tools (down to line and variable, as shown in appendix C). This is exactly the goal set in 1.

Chapter 6

Conclusion

Analysis of available plagiarism detection tools has shown that all are based on various approximate text comparing algorithms and the main output is a single similarity percentage 2.3, while the reason behind the reported similarity percentage - that is, the similar and different parts of code - is shown roughly or not at all. Past testing[11] has shown that this approach is sufficient for marking suspicions pairs, which is particulary useful when dealing with large file sets, but these still have to be reviewed by a human to confirm or reject, and there greater accuracy would be helpful.

It was noted that the text representation of a program is easy to change without much understanding or effort 1.2, and accordingly it was proposed to focus on properties of code that are more difficult to change, such as syntax trees, data flows, operands, etc. Assuming that most plagiators will not make the effort to understand the code, and thus they will be limited to formal changes - essentially refactoring - it was also proposed to declare the compared codes or their subsets similar only if they were same or had only some predefined formal differences, called obfuscation methods.

On listing the available obfuscation methods, it was concluded that they could be exactly defined in-program and detected 3.2. However, in actually developing the comparing algorithms, some relaxations were made due to time constraints - there were too many special cases to account for, and so some degree of approximation was allowed.

Accuracy of comparing is difficult to quantify, as lower but true positive percentage can have greater value than a higher similarity percentage at cost

of false positives, but testing 5.1 has shown performance comparable with or better than commonly used tools (Moss, JPlag), even finding similar file pairs that were overlooked by JPlag, while displaying found similarities in more detail; the most other tools do is highlights whole blocks, while our tool shows exact lines, variables and substrings that were matched (images in appendix C).

Data flow analysis, which is a rather experimental feature, has been implemented and works well for strict comparing(shown in 4.5 and Figure 6.5), but its results are difficult to quantify and summarise, so at this point it is only for huma use; however, it does work, and could be also used as internal part of the other comparing algorithms.

6.1 Possible development

The tool only works with C language. To make it multi-language, it would have to internally use abstract syntax trees only, and convertors to this abstract tree would have to be written for every language to be supported. Only data flow analysis currently uses abstract syntax trees, and so the main comparers would have to be refactored. This should also improve the accuracy of detection, as it could help remove some formal differences - for example, conditional operators v. if-else. Also the Antlr grammars, and thus the derived syntax trees, are not meant for comparing, creating issues such as addition and subtraction both being represented with same node type, which have to be resolved with ad-hoc workarounds.
The support of C also is not complete - for example macros are ignored, so running the files through a C preprocessor would be helpful, and the handling of structs is simplified, etc.

Finally, the configuration of comparing stages (4.2) could be explored more in-depth, and it would be worth trying to use data flow analysis as part of the main comparer instead of the completely separate algorithm it is now to gain more context awareness, and reduce the reliance on lines.

Bibliography

[1] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing", *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*, pp. 76–85, 2003.

[2] T. Parr, "Antlr," *ANTLR*. [Online]. Available: https://www.antlr.org/. [Accessed: 04-May-2022].

[3] A. Parker and J. O. Hamblen, "Computer algorithms for plagiarism detection", *IEEE Transactions on Education*, vol. 32, no. 2, pp. 94–99, 1989.

[4] G. Cosma and M. Joy, "Towards a definition of source-code plagiarism", *IEEE Transactions on Education*, vol. 51, no. 2, pp. 195–200, 2008.

[5] D. Ganguly, G. J. Jones, A. Ramírez-de-la-Cruz, G. Ramírez-de-la-Rosa, and E. Villatoro-Tello, "Retrieving and classifying instances of source code plagiarism", *Information Retrieval Journal*, vol. 21, no. 1, pp. 1–23, 2017

[6] D. Fu, Y. Xu, H. Yu, and B. Yang, "WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection", *Scientific Programming*, vol. 2017, pp. 1–8, 2017.

[7] R. Fraser, "Collaboration, collusion and plagiarism in computer science coursework", *Informatics in Education*, vol. 13, no. 2, pp. 179–195, 2014.

[8] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance", *ACM Transactions on Database Systems*, vol. 40, no. 1, pp. 1–40, 2015.

[9] M. Pawlik and N. Augsten, “Tree edit distance: Robust and memory-efficient”, *Information Systems*, vol. 56, pp. 157–173, 2016.

[10] M. Novak, M. Joy, and D. Kermek, “Source-code similarity detection and detection tools used in Academia”, *ACM Transactions on Computing Education*, vol. 19, no. 3, pp. 1–37, 2019.

[11] T. Votroubek, "Improving Plagiarism Detection", *CTU* [Online]. Available: https://dspace.cvut.cz/handle/10467/73914 [Accessed: 04-May-2022].

[12] Jplag, “JPLAG/JPlag: Detecting software plagiarism and collusion since 1996.”, *GitHub*. [Online]. Available: https://github.com/jplag/JPlag. [Accessed: 04-May-2022].

Appendix A - List of attached files

The following is also a part of this work:

- ./test_codes/ - Files used in testing properties of plagiate detectors (2.1)
- ./comparer/ - Maven-compiled Java 14 project containing implemented comparer.
- ./comparer.jar - Compiled jar of said project.
- ./plag_codes/- Student-submitted codes suspected of plagiarism 5.1.1.
- ./HW03-nonplagiated/- A collection of all solutions submitted by students in a single assignment 5.1.2.
- Appendix B - Installation manual - explains how to compile or run the application.
- Appendix C - GUI - contains images of the application interface

Appendix B - Installation manual

The application is built as a single .jar file using JDK 16, bytecode version 14. It uses the following supporting files:

- A XML settings file `"./comparer_settings.txt"`, but if it can not be loaded, hardcoded defaults will be used. The file contains comments, so the possible settings will not be detailed here.
- C library header files. `assert,termios,unistd,pthread,stdio,stdlib` are part of the .jar, others will be searched for at location given in the settings file. If they are not found, the program will still work, although comparing accuracy will suffer somewhat.

Thus nothing but the .jar file and a sufficiently recent version of the JVM should be needed for the application to work properly.

For compilation, there are the following dependencies:

- APTED comparer, which is part of project source code, so no setup work is needed.
- ANTLR-generated C parser, also part of source code.
- ANTLR library, version 4.9.2[2] in the form of a .jar file which has to be linked as an external library.

There is one known caveat - the ANTLR jar can also be used on its own from the command line, and for this purpose, it contains a `MANIFEST.MF` file that defines, among other things, the entry point of the application like so: `Main-Class: org.antlr.v4.Tool`. So one needs to make sure the main project's manifest file is used instead. A failsafe way to do this is to open the `MANIFEST.MF` file of the compiled .jar and make sure it contains this record: `Main-Class: main.Main`.

Appendix C - GUI

Here are screenshot of the GUI in various phases of comparing, to show how are detected similarities shown. The program shows both compared codes side by side, but the image would be too wide for an A4 page, so these images are cropped to include only one side.

The colors are coded as follows:

- **red** - not matched
- **light green** - matched, clicking will highlight its counterpart in other file
- **dark green** - strongly matched (identical text)

In the `Results` window (Figure 6.4), the most important figures are the following:

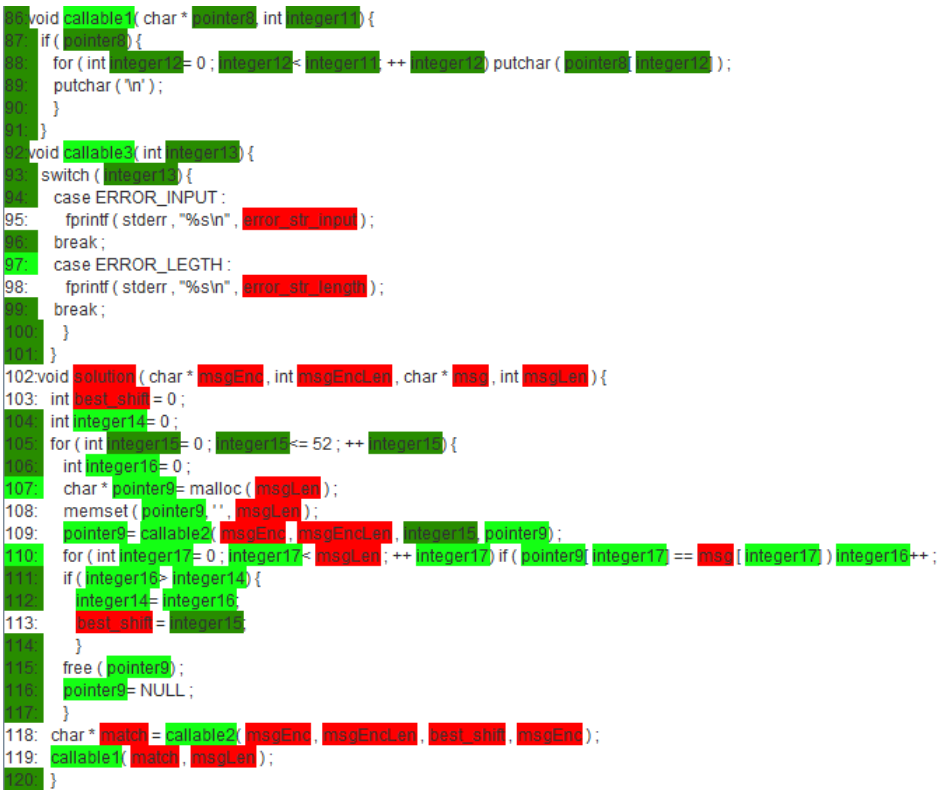
- **Lines** - Original count and count after normalisation3.3.1.
- **Identifiers** - Count of variables.
- **Identifiers matched** - Count of matched variables.
- **Unrenamed matches** - Count of matched variables that have same name in both files.

```
86:void printMsg ( char * msg , int len ){
87:  if ( msg ){
88:    for ( int i = 0 ; i < len ; ++ i ) putchar ( msg [ i ] ) ;
89:    putchar ( '\n' ) ;
90:    }
91:  }
92:void printError ( int error ){
93:  switch ( error ){
94:    case ERROR_INPUT :
95:      fprintf ( stderr , "%s\n" , error_str_input ) ;
96:    break ;
97:    case ERROR_LEGTH :
98:      fprintf ( stderr , "%s\n" , error_str_length ) ;
99:    break ;
100:   }
101: }
102:void solution ( char * msgEnc , int msgEncLen , char * msg , int msgLen ){
103:  int best_shift = 0 ;
104:  int best_match = 0 ;
105:  for ( int i = 0 ; i <= 52 ; ++ i ){
106:    int match_counter = 0 ;
107:    char * new_msg = malloc ( msgLen ) ;
108:    memset ( new_msg , ' ' , msgLen ) ;
109:    new_msg = shiftMsg ( msgEnc , msgEncLen , i , new_msg ) ;
110:    for ( int j = 0 ; j < msgLen ; ++ j ) if ( new_msg [ j ] == msg [ j ] ) match_counter ++ ;
111:    if ( match_counter > best_match ){
112:      best_match = match_counter ;
113:      best_shift = i ;
114:      }
115:    free ( new_msg ) ;
116:    new_msg = NULL ;
117:    }
118:  char * match = shiftMsg ( msgEnc , msgEncLen , best_shift , msgEnc ) ;
119:  printMsg ( match , msgLen ) ;
120: }
```

Figure 6.1: Variable matching

- **Matched lines** - Count of matched lines.
- **Text identical** - How many matched lines have same text representation after normalisation after normalisation and assuming matched variables to have same name.
- **Identical lines** - How many matched lines are identical in original source code.
- **Partially identical** - How many matched lines have common substrings of total length larger than certain threshold (15% is default) in original source code.

In data flow comparing view (Figure 6.5) is shown disassembly of `solution`



```
86:void callable1( char * pointer8, int integer11) {
87:  if ( pointer8) {
88:    for ( int integer12= 0 ; integer12< integer11; ++ integer12) putchar ( pointer8[ integer12] ) ;
89:    putchar ( '\n' ) ;
90:    }
91:  }
92:void callable3( int integer13) {
93:  switch ( integer13) {
94:    case ERROR_INPUT :
95:      fprintf ( stderr , "%s\n" , error_str_input ) ;
96:    break ;
97:    case ERROR_LEGTH :
98:      fprintf ( stderr , "%s\n" , error_str_length ) ;
99:    break ;
100:   }
101: }
102:void solution ( char * msgEnc , int msgEncLen , char * msg , int msgLen ) {
103:  int best_shift = 0 ;
104:  int integer14= 0 ;
105:  for ( int integer15= 0 ; integer15<= 52 ; ++ integer15) {
106:    int integer16= 0 ;
107:    char * pointer9= malloc ( msgLen ) ;
108:    memset ( pointer9, ' ' , msgLen ) ;
109:    pointer9= callable2( msgEnc , msgEncLen , integer15, pointer9) ;
110:    for ( int integer17= 0 ; integer17< msgLen ; ++ integer17) if ( pointer9[ integer17] == msg [ integer17] ) integer16++ ;
111:    if ( integer16> integer14) {
112:      integer14= integer16;
113:      best_shift = integer15;
114:      }
115:    free ( pointer9) ;
116:    pointer9= NULL ;
117:    }
118:  char * match = callable2( msgEnc , msgEncLen , best_shift , msgEnc ) ;
119:  callable1( match , msgLen ) ;
120: }
```

Figure 6.2: Line matching

function from previous figures, with inlined `shiftMsg`, compared to its match from the other code. The original lines are shown to left and right, but had to be cut out here due to narrow page.

```
86:void printMsg(char * msg, int len){
87:if(msg) {
88:for(int i = 0; i < len; ++i){      putchar(msg[i]);
89:putchar('\n');
90:}
91:}
92:void printError(int error){
93:switch (error){
94:case ERROR_INPUT:
95:fprintf(stderr, "%s\n", error_str_input);
96:break;
97:case ERROR_LEGTH:
98:fprintf(stderr, "%s\n", error_str_length);
99:break;
100:}
101:}
102:void solution(char *msgEnc, int msgEncLen,char *msg, int msgLen){
103:int best_shift = 0;
104:int best_match = 0;
105:for (int i = 0; i <= 52; ++i){
106:int match_counter = 0;
107:char *new_msg = malloc(msgLen);
108:memset(new_msg, ' ', msgLen);
109:new_msg = shiftMsg(msgEnc, msgEncLen, i, new_msg);
110:for (int j = 0; j < msgLen; ++j){ //count the number of matching letters      if (new_msg[j] == msg[j])      match_counter++;
111:if (match_counter > best_match){
112:best_match = match_counter;
113:best_shift = i;
114:}
115:free(new_msg);
116:new_msg = NULL;
117:}
118:char *match = shiftMsg(msgEnc, msgEncLen, best_shift, msgEnc);
119:printMsg(match, msgLen);
120:}
```

Figure 6.3: Original text matching

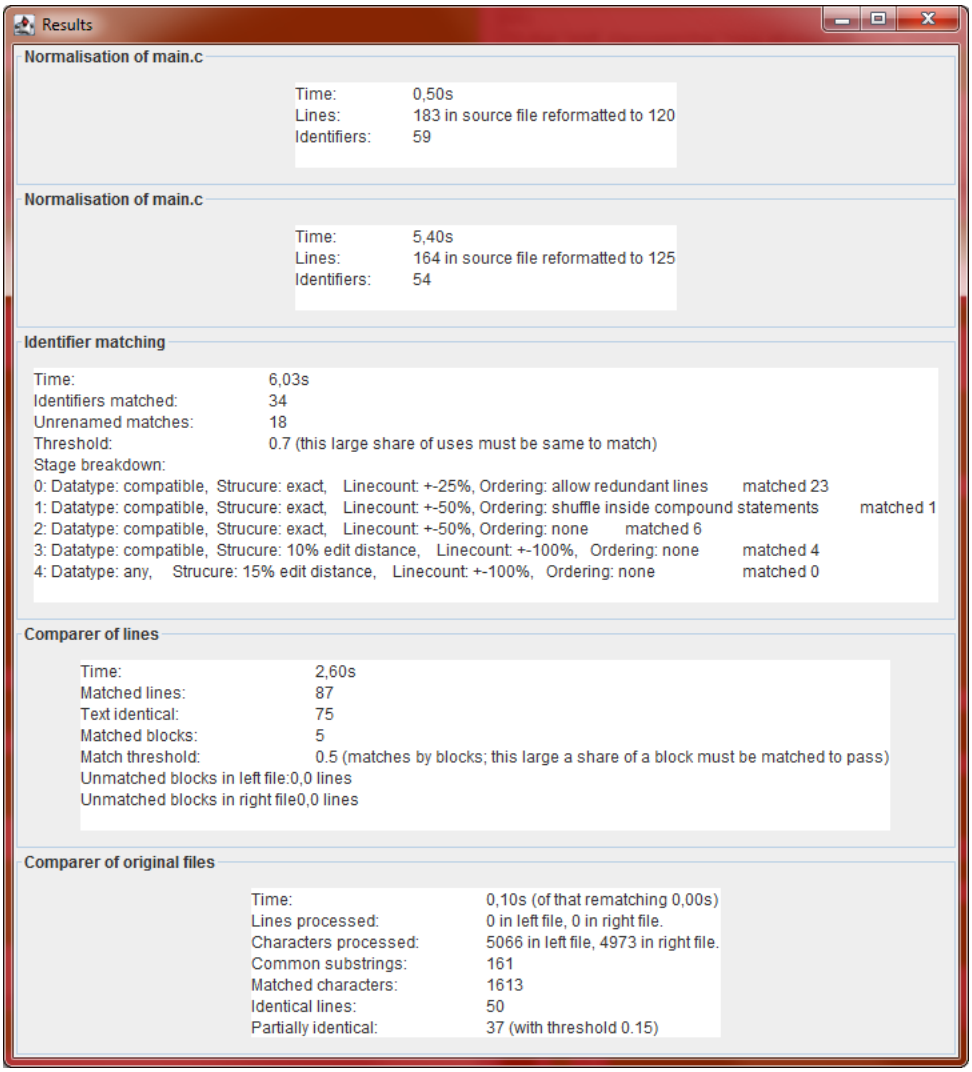


Figure 6.4: Results window

Figure 6.5: Data flow comparing. Code on right side is the `solution` function from previous images.