Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Radek Lano
Source Code Similarity Detection
Katedra softwarového inženýrství
Vedoucí diplomové práce: Ing. Petr Tůma, Dr.
Studijní program: Informatika, Softwarové systémy,
Softwarové Inženýrství
Rád bych tímto poděkoval především vedoucímu své diplomové práce, za ukázkovou trpělivost a příkladné vedení jak při návrhu řešení, tak při testování, psaní dokumentace i vlastního textu diplomové práce.

Také děkuji svým rodičům za významnou podporu v době studia.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne

Radek Lano
\section*{Pictures}

Architecture overview: GCC, parser, reducer, analyzer, comparer and I/O formats................. 15
Divide algorithm: Block of X nodes....................................................................................... 17
Divide algorithm: Block of code according to functions......................................................... 18
SORTIE source code analysis process.................................................................................... 22
The complete analysis process in Columbus.......................................................................... 23
Gasta can be split into four main parts: A parser, a visitor, a builder and an analyzer........... 25
DivideAll division example...................................................................................................... 33
DivideSingle division example............................................................................................... 34
DivideCondition division example......................................................................................... 35
Results of comparison: Similarity of 23 projects.................................................................... 42
HTML output example.............................................................................................................. 48
Title: Source Code Similarity Detection
Author: Radek Lano
Department: Katedra softwarového inženýrství
Supervisor: Ing. Petr Tůma, Dr.
Supervisor's e-mail address: petr.tuma@dsrg.mff.cuni.cz

Abstract
The objective of this thesis is to design and implement a tool usable for detecting similar code in different projects. The tool should be able to locate code pasted from one project to another and should be able to cope with average attempts to thwart the detection such as symbol renaming, changing the order of unrelated entities, moving entities to different files, adding or removing comments, etc.

The tool is implemented in language C++ and is ready to compare source files written in languages C and C++. The tool also enables the comparison of source code written in different languages, which can be compiled by the GNU C Compiler. To obtain good results in these cases, new modules should be added (this is necessitated due to different representations of the GNU C Compiler inner form for different languages).

The first part of this thesis focuses on describing the problem domain, the architecture design and the tools usable for implementation. The second part centers on the implemented solution, a description of data structures and possibilities for application expansion using additional modules. The last part of the thesis sums up the results and outlines future possibilities of implementation.

Keywords: similarity, detection, plagiarism
1 Problem Domain

In this section, we would like to introduce the problem of source code similarity detection as understood in the thesis and the scope of our detection tool – that is, what is and what is not required to be detected as program plagiarism. We start with an introduction describing the motivation for writing a program for source code similarity detection, then we define the primary scope and objectives for the project, and finally we take a look at different types of similarities and requirements for the program's ability to detect such similarities.

**Introduction**

The main motivation for writing this program is closely tied to the Internet, today's most popular information source. More and more frequently, people search the web to find what they need. They seek information about their hobbies as well as data for their jobs. Young students, especially, started using the Internet to find what they needed, including information for their homework. The problem is that they also use the Internet to download and copy their homework from other students so that they can get good grades without having to study - they simply find something similar on the Internet that resolves the same problem and then they change the name of the author. For teachers, it is important to know whether the work was actually done by the student or not.

**Scope**

This project focuses on programming courses. Since programming courses from year to year are fairly similar, students can easily copy the work of their senior schoolmates. Student assignments can be stored by teachers and used later for detection of similarities. Similarities between programming languages are easier to identify than similarities between actual spoken languages, because expression in programming languages is limited by the grammar of the respective language, which is less complex than the grammar of spoken languages. Consequently, the analysis of program plagiarism appears to be simpler and to have a better chance at detecting suspicious work.

**Objectives**

The objective of this project is to develop a tool for detecting source code similarity. The definition of program similarity and various types of similarities are described later in section Similarities. The tool should also take into account the possibility that students know of or will find out about the tool. Therefore, protection against attempts to thwart detection is required. We will realize this protection by adding more types of similarities that are not technically similarities but rather differences that the tool considers attempts at thwarting detection. This means that two programs may be declared similar even when they appear to be different.
1.1 Analyses of Modifications to Detect

Throughout the course of this thesis we will use a floating example to demonstrate possible source code modifications that should be detected by the tool, as well as to demonstrate more complex modifications and, finally, to introduce the tool's usage and output.

Example 1

Our example is source code for a programming course. The students have to program a quicksort algorithm for their homework. Imagine that one student prepares his assignment and writes the following code:

```c
void quicksort(int* items, int l, int r)
{
    int i = l;
    int j = r;
    int x = items[(l + r) / 2];
    int w;

    while (i <= j)
    {
        while (items[i] < x)
        {
            ++i;
        }

        while (items[j] > x)
        {
            --j;
        }

        if (i <= j)
        {
            w = items[i];
            items[i] = items[j];
            items[j] = w;
            ++i;
            --j;
        }
    }

    if (l < j)
    {
        quicksort(items, l, j);
    }

    if (i < r)
    {
        quicksort(items, i, r);
    }
}
```
Similarities

Two programs are considered to have similar source code when there are either no differences between them or there are only differences that we consider to be attempts at thwarting detection. Let’s define the various types of similarities and specify which are considered to be cheating and which are allowed (not required for detection).

Changing the name of variables or functions

Very simple kind of cheating. This must be detected.

Example

The beginning of our floating example:

```c
void quicksort(int* items, int l, int r)
{
    int i = l;
    int j = r;
    int x = items[(l + r) / 2];
    int w;
```

can be replaced by the following code:

```c
void quicksort(int* polozky, int levy_index, int pravy_index)
{
    int pomocna_levy = levy_index;
    int pomocna_pravy = pravy_index;
    int median = polozky[(levy_index + pravy_index) / 2];
    int pomocna;
```

The appropriate parts of later code have to be modified as well.

Changing comments

Comparing comments is as difficult as comparing common language texts, which would require a more complicated solution in order to sufficiently compare them. Moreover, changing comments is very simple and if the student is too lazy to even change the wording of the comments, chances are s/he was too lazy even to change the code. As our program compares code, the similarity will be detected. Therefore, comments can be ignored. Moreover, since we prefer to use a common preprocessing tool for source code analysis, it is possible that we will not even have access to comments at all.

Splitting functions

Changing the structure of the code by splitting functions is quite a simple attempt to thwart detection. If the structure of the code is changed only slightly, then it should be detected without any problem. We require the tool to detect separation of a block of statements into a function called in place of that block. Splitting a sequence of statements into more functions and calling them sequentially should be detected, too. However, any significant modification of the code by splitting functions using code reorganization means that the program will have a different code structure. In this case, we will consider the program to be different.
Example

The next part of our floating example:

```c
while (items[i] < x)
{
  ++i;
}
while (items[j] > x)
{
  --j;
}
```

can be replaced by creating a function:

```c
void moveIndex(int* items, int x, int* index, int direction)
{
  while ((direction * items[*index]) < (direction * x))
  {
    *index = *index + direction;
  }
}
```

and calling this function twice:

```c
moveIndex(items, x, &i, 1);
moveIndex(items, x, &j, -1);
```

Adding unimportant variables or parameters into functions

The addition of unused variables should not thwart detection of similarities; unused variables should be ignored. Adding a few new variables as well as source code that uses them should also not thwart similarity detection. However, when a lot of new variables and source code that uses them are added, it may be difficult to determine which parts are actually relevant to the functionality of the code and which parts were simply added to thwart detection of similarities. Adding variables with source code should only confound detection when the amount of variables and code added doubles the size of the original code.

Example

In our floating example, just below the function header:

```c
void quicksort(int* items, int l, int r) {
```

we can add the variable:

```c
int itemsCount;
```

and define some code:

```c
itemsCount = r - l + 1;
```
Slightly more complex examples of source code modifications

Below, you can find slightly more complex examples of source code modification using various combinations of the possible modifications previously mentioned. These examples are used later to evaluate the precision of the proposed tool.

Example 2

The second student wants to copy his friend's homework so that he will have a functional program, but he wants to pretend this homework is his own. Therefore, he changes the comments and the names of variables into local (Czech) names.

```c
void quicksort(int* polozky, int levy_index, int pravy_index) {
    int pomocna_levy = levy_index;
    int pomocna_pravy = pravy_index;
    int median = polozky[(levy_index + pravy_index) / 2];
    int pomocna;

    while (pomocna_levy <= pomocna_pravy) {
        while (polozky[pomocna_levy] < median) {
            ++pomocna_levy;
        }

        while (polozky[pomocna_pravy] > median) {
            --pomocna_pravy;
        }

        if (pomocna_levy <= pomocna_pravy) {
            pomocna = polozky[pomocna_levy];
            polozky[pomocna_levy] = polozky[pomocna_pravy];
            polozky[pomocna_pravy] = pomocna;
            ++pomocna_levy;
            --pomocna_pravy;
        }
    }

    if (levy_index < pomocna_pravy) {
        quicksort(polozky, levy_index, pomocna_pravy);
    }

    if (pomocna_levy < pravy_index) {
        quicksort(polozky, pomocna_levy, pravy_index);
    }
}
```

Later while testing we will see that the similarity detection program declares this code to be exactly the same as the code from Example 1.
Example 3

The third student in our example already knows about the detection tool, so he tries harder to deceive the tool and, in addition to changing the comments and the names of variables, he creates some additional functions, pasting parts of code into them and then exchanging the original parts of code by calling these newly created functions.

```c
void posunIndex(int* polozky, int median, int* index, int smer) {
    while ((smer * polozky[*index]) < (smer * median)) {
        *index = *index + smer;
    }
}
```

```c
void vymen(int* polozky, int leva, int prava) {
    int pomocna;

    pomocna = polozky[leva];
    polozky[leva] = polozky[prava];
    polozky[prava] = pomocna;
}
```

```c
void quicksort(int* polozky, int levy_index, int pravy_index) {
    int pomocna_levy = levy_index;
    int pomocna_pravy = pravy_index;
    int median = polozky[(levy_index + pravy_index) / 2];

    while (pomocna_levy <= pomocna_pravy) {
        posunIndex(polozky, median, &pomocna_levy, 1);
        posunIndex(polozky, median, &pomocna_pravy, -1);
        if (pomocna_levy <= pomocna_pravy) {
            vymen(polozky, pomocna_levy, pomocna_pravy);
            ++pomocna_levy;
            --pomocna_pravy;
        }
    }

    if (levy_index < pomocna_pravy) {
        quicksort(polozky, levy_index, pomocna_pravy);
    }

    if (pomocna_levy < pravy_index) {
        quicksort(polozky, pomocna_levy, pravy_index);
    }
}
```

Later while testing we will see that the similarity detection program declares this code as 40% similar to the code in Example 1 and Example 2.
2 Solution Architecture and its Benefits

This section describes the architecture of the detection program and the key ideas leading to the chosen solution.

2.1 Internal Representation

One of the possible ways to compare source code in different programming languages with block structure using the same solution is to write a compiler from each possible language into the typical format and to analyze the data in the typical format.

The typical format has to contain key words according to programming languages such as C++ and Java:

- if – then – else
- cycles (while, for, ...)
- local variables
- data structures
- function calls
- ...

The key advantage is that it is possible to detect even algorithm plagiarism among different programming languages. However, this kind of plagiarism is not in the scope of this project. We are interested in detecting use of the same code. When only an algorithm is copied and rewritten in a different programming language, most likely a huge amount of work must be done, therefore we do not consider such plagiarism to be cheating.

Another advantage is that all programming languages can be processed and analyzed in a similar way, which makes the solution more flexible (the same analyzing modules can be used for multiple programming languages). This is also a disadvantage, because the same analyzing modules may have different results for different programming languages, but this disadvantage can be inhibited using the approach of testing all modules for each programming language and interpreting the results by picking out only the modules with the best similarity fit.

Similarity detection should only be dependent on program structure, which is specific to each programmer. We decided to ignore comments and the formatting of source code, because these can be changed easily without any knowledge of the copied program. The best choice for analyzing program sources is to analyze the solution instead of the appearance of the code. It is possible to analyze translation units instead of source files, because translation units keep information about program structure and discard the programmer's comments. When analyzing translation units, we can avoid difficult parsing of languages C and C++. We can use a standard front-end for parsing source files and for providing parsed translation units.
During the analysis, we can ignore the header files in the examined source code. There is, however, a good reason for ignoring them. Most of the header files are included in other source files, so they are analyzed along with the source files. The header files that are not included in the source files can be ignored. Theoretically there is a chance that a header file will be used without being included from the source code, but this is not likely and any unusual use of header files would probably be noticed by the professor during work presentation.

Based on the considerations previously mentioned, we decided to use an abstract syntax tree as the analyzed structure, and the obvious way to obtain this is to use the common compiler as a front end. Our detection tool can process an abstract syntax tree produced by the compiler to find differences between it and another abstract syntax tree produced by the same compiler.

We chose the GNU C compiler for the following reasons:

- It is free and the one most commonly used among Linux compilers.
- It is able to parse not only C and C++ sources, but also Java and many other sources, as well. Therefore, analysis of multiple language source code using the same solution is possible.

Our architecture encapsulates the type of solution previously mentioned by using the GNU C compiler as front end, because the compiler is able to parse most of the common programming languages using different front ends and transforms the source code into a structure called an abstract syntax tree. We use the dump of the abstract syntax tree and parse it using our own parser which effectively obtains the important symbols and reduces the abstract syntax tree dump to extract only relevant information. Later, we use the reduced abstract syntax tree for analyzing the source code and for creating hashes for comparison between projects.

2.2 Piped Processing

There are many advantages to using pipelined processing, especially program parallelism. We can process various analyzed projects on different machines and later compare the analyzed results in order to find similarities between projects. It is also possible to reuse analyzed projects and compare them to newly analyzed projects, so piped processing saves the time required for the processing of a large number of projects while searching for similarities.

Because of this, we will use simpler programs to resolve different parts. One program will process the source code before analysis. The next will analyze the processed source code. The final program will compare the analyzed projects. Separation into simpler programs makes the solution more flexible in regard to the experimental addition or removal of modules. When a large amount of source code is being analyzed, it is possible to compile them first using the GNU C Compiler and then to parse the abstract syntax tree dumps received. Afterward, it is possible to do as many analyses of the abstract syntax tree as required. These analyses only need to be done once for each project, then they can be used whenever multiple projects are compared. This means they do not have to be analyzed again for every comparison.
To maximize the options for separating source code analysis from comparison, we decided to use hash signatures. Each hash signature represents some characteristic of a project's source code. The output of each project's analysis is a set of hash signatures for that project. Hash signatures can be created in various ways based on different criteria. Comparing projects is later done using the projects' hash signatures – project similarity is defined as the similarity of the projects' hash signatures. The use of hash signatures enables the processing of source code analysis to be done separately for each batch of projects, the hash signatures of older projects are stored and can later be compared to newly analyzed projects.

Consequently, we use piped processing – that is, instead of one application, we want a separate application for each part of the program. There is an application for parsing the abstract syntax tree, followed by an application for reducing it to a tree with the relevant information extracted. After that, there is an application for analyzing the structures, with the output being a file containing hash signatures. Finally, there is an application for comparing the hash signatures to the hash signatures of another project.

A piped solution saves processing time because it is possible to have the work of older students already prepared and, later, when new projects are accepted, they are parsed, analyzed and compared much faster.
2.3 Overview

Architecture overview: GCC, parser, reducer, analyzer, comparer and I/O formats
2.4 Parser

The parser processes GNU C Compiler abstract syntax tree dumps and produces output in a format based on element types, identification and attributes, etc. Each element is marked as a statement or a connector. A statement means that a node has an important significance in the code. Elements marked as connectors are supposed to be removed from later processing (analyzing and comparing). This step is done in the reducer component; the parser only marks the elements to be deleted. The identifiers of the nodes are kept unchanged so that the divide and hash modules are able to access the information about the real significance of elements from the GNU C Compiler abstract syntax tree. The objective of our parser is to produce a much simpler dump, which can be reduced into a file with maximum information about source code and minimal file size. Parser output structure is used as both parser and reducer output. The structure is described later in this section.

2.5 Reducer

The reducer component is used to remove unimportant nodes marked by the parser as connectors. Before removing the nodes, the reducer walks the element structure, gathers all the required information from connector elements and stores it into statement elements. Rules for additional information gathering are defined in the part about the reducer in section Implemented Solution. The reducer produces output in the same structure as the parser. The structure is described later in this section.

2.6 Analyzer

The analyzer is used to create a file with hashes from the reduced parser structure. Parser structure and the structure of hash files are described later in this section. The analyzer creates a SimilarityForest from elements in the processed file (see section Implemented Solution for details about the SimilarityForest class). Then the analyzer runs each combination of the divide module and hash function on SimilarityForest, producing hashes in the output file.

2.7 Divide and Hash Modules

One of the ideas in the beginning was to enable the comparison process to be done based on modules - something similar to spam assassins which use many independent modules, each discovering another suspicious part of incoming e-mail. The architecture of our program uses two kinds of modules – divide and hash.

2.7.1 Divide Modules

Divide modules represent different techniques for dividing abstract syntax trees, which represent analyzed source code.
**Block of X nodes**

We started with a division based on the following algorithm:

1. Start at the beginning of the source code.
2. Take a block of code consisting of the current node and the next X-1 commands, add them into a separate tree. X is a positive number and is configurable.
3. For each branch continuing from the removed block of code continue in step 2 until no nodes are left in the code.

Divide algorithm: Block of X nodes

The algorithm above has a significant problem when a few lines are inserted at the beginning of the source code, or when some code is moved from the beginning to the end (if possible without changing the code's purpose). In the mentioned cases, the algorithm creates very different subtrees and also produces a different number of subtrees. That is why this kind of solution was unacceptable and we tried to discover other dividing algorithms.
**Block of functions**

The next algorithm we considered was: Divide all nodes belonging to the same function into a different subtree.

![Diagram of the block of functions algorithm](image)

*Divide algorithm: Block of code according to functions*

The above algorithm has a problem when a function is divided into 2 shorter functions. In this case, one subtree cannot be equal to two shorter (divided) subtrees. A similar problem is the joining of two functions used one after another into one longer function. Dividing and joining functions is simple and a student's attempts use these methods to disguise his copying of source code should be detected. For this reason, this divide algorithm was rejected as well.
Implemented divide modules

Neither of the mentioned examples can be fixed, even using very good hashing functions. This leads to the search for an algorithm that can work with code that has been disrupted by actions such as those in the given examples. However, the algorithm should still find similarities later in the parts of the code that the student has not modified, so that only an excessive amount of changes would stop the exposure of plagiarism. A good algorithm, therefore, has to have certain control points - places where it “synchronizes” the subtrees and produces the same subtrees again. We decided to use the conditions (if, switch, for, while) as these synchronization points. The implemented algorithm is described later in section Implemented Solution.

In the end, three modules were implemented as part of this thesis:

- **All** – the abstract syntax tree is taken as is without any division (no module used). The initial graph before any division can be used by hash functions which require some special dividing to be done.

- **Single** – the abstract syntax tree is divided at each basic node, which means each node of the tree is in a standalone tree. A comprehensive division of the tree used for basic comparison of the amount of source code changed.

- **Condition** – the abstract syntax tree is divided by its condition nodes (if, switch, while, for). This module has really good chances for detecting plagiarism, because even partially modified source code is matched with parts that have not been modified.

See section Implemented Solution for details about the implemented divide modules.
2.7.2 Hash Modules

Hash modules represent different techniques on how to hash the divided abstract syntax trees into hash files. Each tree is hashed separately and all the hashes are stored in a file. Hash modules should be the most variable part of the implemented solution. Each hash module has a different approach. We depend on experimental testing to prove hash module ability to detect similarities rather than theoretical proof.

We implemented the following hash modules as part of this thesis:

- **HashArithmeticExpression** – creates a specific hash signature for each arithmetical expression in a given tree
- **HashCount** – has three parts: counts all the nodes in a tree, counts the functions called and counts the modifying nodes
- **HashDataStructure** – creates a specific hash signature for each data structure in a given tree
- **HashInstruction** – creates a specific hash signature for each expression in a given tree
- **HashVariable** – creates a specific hash signature according to the types of variables in a given tree and counts their uses
- **HashVariableType** – creates a specific hash signature according to the types of variables in a given tree

See section Implemented Solution for details about implemented hash modules.

2.7.3 Reliability

If abstract syntax trees with too little or no information relevant to hash modules were hashed, it would make no sense to compare them, because there would be the risk of too many matches for their hash signatures. For example, hash module **HashDataStructure** hashes a number of trees without any specific data structure. In this case, the module would produce a lot of same hashes for different projects. That is why each hash function has the attribute reliability, which contains the ratio of relevant information in the tree being hashed from the hash module's point of view.

See section Implemented Solution for details about implemented reliability in different hash modules.
2.8 Comparator

The tool has two main parts – analysis (previously described) and comparison. During analysis, projects are transformed into sets of hash signatures. During comparison, the sets of hash signatures from different projects are matched and searched for best fits. The comparer component takes two or more hash files produced by the analyzer component for different projects and creates output with comparisons of every two projects.

To prevent random matches, each hash signature has a reliability value - more important hash signatures have higher reliability values than less important ones. Hash signatures with zero reliability are removed from the comparison.

Project one is X% similar to project two when X% of project one's hash signatures can be found among the hash signatures of project two. When all the hash signatures of project one are found among the signatures of project two, the whole of project two or its part was probably copied.

Project one is X% reliability similar to project two, when X% is the sum total reliability of project one hash signatures found in project two over the total reliability of all of project one's hash signatures.

For comparer usage and options see section User Manual.
3 Related Work

3.1 Usable Components

In this section, we would like to introduce the tools that can be used as part of any (even an open source) project that focuses on parsing and/or analyzing program source code. We originally planned to use some kind of tool as a part of our project, and we tried few of them, too. This section describes our motivation for using such tools as well as our experience with the tested tools. We will elaborate on their strong points and the key disadvantages that in the end kept us from using them.

3.1.1 SORTIE Code Analysis

From the very beginning it was obvious that SORTIE was not a tool usable as a part of the solution, but it was good for information. The authors described the source code analysis process and the architecture of the processing system. Their theories and results can serve as an inspiration to us.

We are interested in two parts – parsing source files and the graphical representation of received outputs.
3.1.2 FRONTENDART Columbus/CAN

Columbus/CAN is a tool for reverse engineering and the reverse engineering process.

The complete analysis process in Columbus

The tool looks great and we wanted to try it for our own purpose, but then we found out that distribution is only available for Windows, which was not acceptable for our solution.
3.1.3 XOGASTAN

As the first possible part of the solution, we tried the software XOGastan, developed by the University of Sannio in Benevento, Italy. To get a glimpse of the XOGastan project scope and architecture, take a look at the citation from XOGastan's homepage.

XOGastan means XML-Oriented Gcc Abstract Syntax Tree ANalyzer. Wahoo, it's a very long name! Yes, but its interpretation is very simple.

The name is composed of three parts:

- XML Oriented
- Gcc Abstract Syntax Tree
- ANalyzer

The first part, XML Oriented, gives us information about the technology XOGastan uses: the "new" XML technology.

The second part, Gcc Abstract Syntax Tree, tells that XOGastan interacts with the output file of gcc. More precisely, it uses the file produced by gcc that contains the ast of a C program. This file, dumped before XOGastan is used, is successively translated into an intermediate XML representation.

The third part, ANalyzer, gives us an hint about the purpose of XOGastan: it analyzes its input file and produces some data.

Hence: "XOGastan is a program that analyzes the file containing the AST generated by gcc. The result is information about the C program represented by the AST. The analysis and the results are formatted and presented using the XML technology."

In short, XOGastan converts the AST dump to XML, which is better to analyze than the original dump for the following reasons:

- The XML output is GNU C Compiler independent, enabling unproblematic usage of new compiler versions. New versions may use different symbols than previous versions, but the XML will be the same.
- XML is easier to read and to work with than the AST dump, because it is quite difficult to analyze all AST dump parts. Why not use the work of someone who has already done the analysis?
- XML is a great technology, which is used more and more often nowadays.
- It uses perl scripts that transform the AST dump into GXL (graph exchange language), so it is possible to use standard tools for graph analysis and visualization.

XOGastan was too complicated to use as a part of this project. It requires preparation in advance of source code, followed by perl processing and resulting in a thousand-node graph, so that later reductions have to be used in order to make GXL usable. Therefore, we started looking for another solution.
3.1.4 GASTA

This seemed to be a great tool after considering all the mentioned disadvantages of XOgastan. It produces a usable control flow graph and is written in C, so we expected simpler implementation as a part of this project.

Let's focus on gasta architecture:

![Gasta Architecture Diagram]

Gasta can be split into four main parts: A parser, a visitor, a builder and an analyzer

As you can see from the picture, we could use ASG in memory produced by gasta and write our own visitor and builder. We did so. We wrote our own visitor and builder as required in gasta and used them to produce control flow graphs, then we used the control flow graphs to analyze similarities between source code. We used gasta headers rewritten into C++ and gasta's code linked as a static library. Making gasta functional as a part of a C++ program was boring work, but it seemed to be a usable solution.

Everything worked well until in the test phase of the project we discovered a fatal problem – gasta was only written to support C sources. It was able to parse C++ programs, but in most cases it ended with a “segmentation false” error. We discovered that gasta fails each time any namespace declaration occurs. After communication with gasta's author, we discovered that further work on the project had been stopped at that time and we could not expect gasta to parse anything other than C source code in the near future.

3.1.5 Nokia Source Code Analyzer

Not tested, because it is a paid program.
3.1.6 SiSSy / Recoder

A tool usable for transformation of some programming languages (Java, C++, Delphi) into a meta model stored in a SQL database. It provides useful tools for AST parsing and manipulation. However, SiSSy was not released yet when we worked on our tool.

More information can be found in the document about SiSSy [6].

3.2 Other Solutions

In this part, we discuss programs usable for similarity detection, their advantages and disadvantages.

3.2.1 SIM

Software written by Dick Grune used for lexical similarity in texts. It can be used to detect plagiarism in software projects. We did not know about this software when we worked on our tool.

We ran a few tests on the SIM program, for the examples mentioned in section Analyses of Modifications to Detect. SIM declares Example 2 is a 100% copy of Example 1, and Example 3 is 36% copied from Example 1. Our tool declares a 100% similarity in the first case and 40% in the second case.

One problem with SIM we know about is that it can be easily thwarted by adding commented source code. SIM compares the entire text (including comments), so commented parts are matched along with source code. This can be prevented by separating source code and comments and comparing each separately. Our tool ignores comments, so we do not have any problem with attempts to thwart detection by using source code in the comments.

Results for SIM and our tool are pretty similar and it would be interesting to do some more tests on actual data. Solutions based on a combination of SIM and our tool during a student's works analysis is possible, too. SIM is usable as a standalone solution, especially for smaller projects where the whole source code is revised by the teacher, so any attempts to thwart detection using code in comments is easily discovered.

More information about SIM can be found on web page [7].
4 Implemented Solution

4.1 Data Structures

4.1.1 AST Dump

We use the standard AST dump produced by the GNU C Compiler front end by using option \textit{-fdump-translation-unit}.

More information about terms used in the dump can be found in the GNU C Compiler documentation, in the section about intermediate representation used by the C and C++ front ends\cite{1}.

Format example:

```plaintext
@1 namespace_decl name: @2 srcp: <built-in>:0
dcls: @3
@2 identifier_node strg: :: lngt: 2
@3 function_decl name: @4 mngl: @5 type: @6
srcp: quicksort1.cc:1 chan: @7
args: @8 link: extern body: @9
```

4.1.2 Parser Structure

The parser works with \textit{ParserElement} class objects and the structure produced by the parser is a list of these objects. Each object is written on a separate line. Each line consists of parts separated by a semicolon.

These parts are: type, identification, separator identification, identifier, source line, followers and parameters:

- **Type**: Possible types are STATEMENT, CONNECTOR and SEPARATOR.
- **Identification**: Element's identification, a positive number.
- **Separator identification**: represents a container for elements within the same parsing space. Elements with the same separator identification are processed together; they can have links in between them. Elements with one separator identification cannot be connected in any way to elements with a different separator identification.
- **Identifier**: text that represents the element's exact purpose.
- **Source line**: a line in the source file where the element occurred, if it is known.
- **Followers**: a comma-separated list of identifications of the element's followers.
- **Parameters**: a list of parameters separated by the number sign (\#). The list contains parameter name, type and value. See the \textit{ParserParameters} class for details.
See section Implemented Solution for details about classes ParserElement, ParserParameters and the meaning of the attributes previously mentioned.

4.1.3 Analyzer Structure

The analyzer reads data in the parser structure, creates similarity nodes and connects the nodes to produce a similarity forest. For details, see classes SimilarityNode and SimilarityForest in section Implemented Solution.

The output of the analyzer is a hash file with hashes – one hash per line. A hash consists of the following parts:

- Source – name of the file processed by the analyzer while creating the hash
- Hash type – hash type according to the module used to create the hash
- Divide type – division type according to the module used to create the hash
- Root identification – identification of the node used as a root for the hash (usable for tracing the original source code part back from the hashes)
- Reliability – reliability of the hash based on hash complexity according to the hash module used
- Hash – the hash produced by the hash module

See section Implemented Solution for details about classes SimilarityNode, SimilarityForests and the meaning of the attributes mentioned above.
4.2 Parser

When we found out there were no tools available for the type of source code processing we needed, we decided to take the abstract syntax tree dump produced by GNU C Compiler and process it using our own tool, which means parse it, reduce the meaningless nodes which were only there for the compiler's needs and produce a simple tree structure with all important information such as node types, argument lists, argument values and so on.

The original translation unit dump contains too many lines with no informational value. For a general overview, see the attachment GAST dump – there are over 5000 lines of output for this source file:

```c
int main()
{
    return 0;
}
```

The parser is written using the bison generator utility version 2.2, for implementation details see file src/Parser.y.

4.2.1 Parser States

The parser works as an automated machine with 4 states.

(0) Initial state

The parser expects the new valid part of the GAST dump as input. It reads the lines for one parser input from the file into a buffer (multiple lines per input). The parser checks whether the line is correct and decides whether the line type is a separator or not. If it is a separator, it produces output, otherwise it continues processing the line after switching to state 1.

(1) Read action

The parser determines the action to be taken, which determines the type of node (statement or connector). Then the state is changed to 2.

(2) Read attributes

In this state, the parser reads a list of parameters. Parameters in the dump are written as name:value, where name has exactly 4 chars (well, it can have 3 or less chars, but in that case, the missing chars are filled in by spaces). The parser reads the parameter's name and then switches to state 3 to get the parameter's type and value.

(3) Read parameter's type and value

Parser just reads the parameter's value, decides the parameter's type and switches back to state 2.
4.2.2 Identifiers and Attributes

In the original GAST tree dump, we focused only on the following identifiers and their attributes.

<table>
<thead>
<tr>
<th>Structures and types</th>
<th>Control flow</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>translation_unit_decl</td>
<td>if_stmt</td>
<td>line</td>
</tr>
<tr>
<td>tree_list</td>
<td>while_stmt</td>
<td>init</td>
</tr>
<tr>
<td>identifier_node</td>
<td>for_stmt</td>
<td>decl</td>
</tr>
<tr>
<td>type_decl</td>
<td>scope_stmt</td>
<td>expr</td>
</tr>
<tr>
<td>field_decl</td>
<td>compound_stmt</td>
<td>cond</td>
</tr>
<tr>
<td>function_decl</td>
<td>return_stmt</td>
<td>then</td>
</tr>
<tr>
<td>void_type</td>
<td></td>
<td>else</td>
</tr>
<tr>
<td>boolean_type</td>
<td></td>
<td>next</td>
</tr>
<tr>
<td>integer_type</td>
<td></td>
<td>body</td>
</tr>
<tr>
<td>real_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>record_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pointer_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>function_type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decl_stmt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr_stmt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next</td>
<td></td>
<td></td>
</tr>
<tr>
<td>body</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.3 Reducer

The reducer works on groups of elements with the same separator identification. At first a list of followers is transformed from identifications into pointers. In this constructed tree, important information is obtained from the list of followers (see part Fetch parameters below).

When all information is gathered into statement elements, the reducer removes all connector elements. For each connector, previous elements (elements to which this connector is a follower) are linked to each connector's follower, so dependency is still stored in statement nodes. Connectors are afterwards deleted.

List of elements – now only statements and separators – is written into an output file and processed later by the analyzer (see part Analyzer later in this section).

4.3.1 Fetch Parameters

Variable types

Variable types are stored in different elements than variable declarations (var_decl), but for our analysis, we need the variable type to be accessible from the variable element. That is why the reducer finds a link to variable type and stores the variable type into text parameter VariableType of the variable element (a new parameter is created).

Field types

Like variable types, field types are stored separately from field declarations (field_decl), too. The reducer finds a link to field type and stores the field type into text parameter FieldType of the field element (a new parameter is created).

4.4 Types of Similarity Nodes

It is pretty simple to determine the purpose of each type from that type's name (for example, an IF_STATEMENT for the beginning of an IF control flow action), so we do not need to show a list of them with their meanings. Similarity node types are defined in file SimilarityNodeType.h; check it for more information about similarity node types.

4.5 Divide Module Interface

The interface for the divide module is realized by class DivideAlgorithm defined in header file DivideAlgorithm.h.

```cpp
class DivideAlgorithm
{
public:
    DivideAlgorithm();
    virtual ~DivideAlgorithm();

    virtual string getName() const = 0;

    /**
     * Divide trees in given forest into subtrees.
     */
    virtual SimilarityForest* divide(SimilarityNode* tree) const = 0;
};
```
Each written divide module must be a child of class *DivideAlgorithm*. There are two virtual functions to implement, excluding module constructor and destructor.

The first function called getName() just returns the module name, which is stored in the file with hash signatures.

The second virtual function called divide is the key part of the divide module. It implements the divide algorithm required by the module. **The input tree should not be changed!** Nodes should be copied into a newly created divided tree. Modules are not supposed to change the input tree; however, they can set help flags by calling the setHelp() function of class SimilarityNode to mark already accessed nodes of the tree. Before calling the divide module, help flags of the given tree are cleared (set to 0). The main algorithm runs just one module on a single similarity tree at a time, which means that the getHelp() and setHelp() functions can be used safely while going through the tree.

In the constructor, there must be initialization of whole module structure. The module is initialized and disposed of each time it is used. Divide modules are initialized for each source separately during analysis of sources.

The complete module has to be added to the *SimilarityAnalyzer*. See information in source file src/SimilarityAnalyzer.cpp to find current information about inserting a new module.

### 4.6 Hash Module Interface

The interface for the hash module is realized by class *HashAlgorithm* defined in header file HashAlgorithm.h.

```cpp
class HashAlgorithm
{
  public:
    HashAlgorithm();
    virtual ~HashAlgorithm();

    virtual string getName() const = 0;
    virtual Hash hash(SimilarityNode* tree) const = 0;
};
```

Each written hash module must be a child of class *HashAlgorithm*. There are two virtual functions to implement, excluding the module constructor and destructor.

The first function called getName() just returns the module name, which is stored in the file with hash signatures.

The second virtual function called hash is the key part of the hash module. **The input tree should not be changed!** If tree modifications need to be done before creating the module's output, nodes should be copied into a newly created tree. Modules are not supposed to change the input tree; however, they can set help flags by calling the setHelp() function of class SimilarityNode to mark already accessed nodes of the tree. Before calling the hash module, the help flags of the given tree are cleared (set to 0). The main algorithm runs just one module on a single similarity tree at a time, which means that the getHelp() and setHelp() functions can be used safely while going through the tree.
In the constructor, there must be initialization of the entire module structure. The module is initialized and disposed of each time it is used. Hash modules are initialized for each source separately during analysis of sources.

The complete module has to be added in the `SimilarityAnalyzer`. See the information in source file `src/SimilarityAnalyzer.cpp` to find current information about inserting a new module.

4.7 Divide Algorithms

4.7.1 DivideAll Module

The abstract syntax tree is taken as is without any division (no module is used). The initial graph before any division can be used by hash functions that require some special dividing.

![DivideAll division example](image-url)
4.7.2 DivideSingle Module

The abstract syntax tree is divided into basic nodes, which means each node of the graph is in a standalone tree. Total division of the graph is used as a basic comparison of the amount of source code changed.

*DivideSingle division example*
4.7.3 DivideCondition Module

The abstract syntax tree is divided by its condition nodes (if, switch, while, for). This module has really good chances for detecting plagiarism, because even partially modified source code is matched by the parts that have not been modified.

For implementation details, see the source code in the directory divide.
4.8 Hash Algorithms

Each hash algorithm should capture one or more characteristics of a program. Below, you can find implemented hash modules and information about their theoretical resilience to source code modifications. However, each hash signature is limited to a block of code according to the divide algorithm. Final similarity recognition is a combination of divide algorithms and hash algorithms.

For implementation details, see the source code in the directory hash.

4.8.1 HashArithmeticExpression Module

Creates a specific hash for each arithmetical expression in a given tree.

Maximum reliability is set for trees with at least 11 expressions. When one or no expression is used in the whole tree, reliability is set to 0 and the hash is discarded.

The hash is resilient to the moving of expressions to different places in the same block of code (blocks defined based on the divide algorithm). The hash is resilient to the moving of expressions into separate functions if this function is a part of same divide block as the original position of the expression. The hash is resilient to the adding or removing of code excepting the addition or removal of arithmetical expressions.

4.8.2 HashCount Module

The hash has three parts: a count of all the nodes in a tree, a count of the functions called and a count of modifying nodes.

Maximum reliability is set for trees with at least 30 nodes. For 5 or less nodes, reliability is set to 0 and the hash is discarded.

Nodes in the block of code are mostly dependent on the divide algorithm. Within the limits of the division, hash signatures created by this module are position independent, meaning the structure of the block can be changed as much as possible and the module will produce the same hash signature.

The hash is not resilient to the addition or removal of any kind of code.

4.8.3 HashDataStructure Module

Creates a specific hash for each data structure in a given tree.

Maximum reliability is set for trees with at least 23 type or field declarations. For 3 or less type or field declarations, reliability is set to 0 and the hash is discarded.

The hash is resilient to any moving of data structures to different places in the same block of code or to the moving of a whole set of data structures from one divided block of code into a different one, respectively changing the sets of data structures in two blocks of code (blocks as based on the divide algorithm). The hash is resilient to the addition or removal of code but not to the addition or removal of data structures.

The hash is not resilient to the splitting of data structures in between different blocks of code (divided blocks based on the divide algorithm).
4.8.4 HashInstruction Module

Creates a specific hash for each expression in a given tree.

Maximum reliability is set for trees with at least 25 expressions. For 5 or less expressions within the tree, reliability is set to 0 and the hash is discarded.

The hash is resilient to any moving of the parts of source code to different places in the same block of code (blocks based on the divide algorithm).

The hash is not resilient to the splitting of instructions in between different blocks of code (blocks are based on the divide algorithm). The hash is not resilient to the addition or removal of source code.

4.8.5 HashVariable Module

Creates a specific hash according to the types of variables in a given tree and the count of their uses.

Maximum reliability is set for trees with at least 10 variable definitions. For 2 or less variable definitions, reliability is set to 0 and the hash is discarded.

The hash is resilient to the moving of variables to different places in the same block of code (blocks outlined according to the divide algorithm) or the moving of an entire set of variables from one divided block of code into a different one, respectively changing the sets of variables in two blocks of code. The hash is resilient to the addition or removal of code, but is not resilient to the addition or removal of variables and code using the variables.

The hash is not resilient to the splitting of variables in between different blocks of code (blocks are outlined according to the divide algorithm). The hash is not resilient to the changing of variable type.

4.8.6 HashVariableType Module

Creates a specific hash according to the types of variables in a given tree.

Maximum reliability is set for trees with at least 10 variable definitions. For 2 or less variable definitions, reliability is set to 0 and the hash is discarded.

The hash is resilient to any moving of variables to different places in the same block of code defined by the divide algorithm or to the moving of an entire set of variables from one divided block of code into a different one, respectively changing the sets of variables in two blocks of code. The hash is resilient to the adding or removal of code but not to the addition or removal of variables.

The hash is not resilient to the splitting of variables in between different blocks of code as defined by the divide algorithm. The hash is not resilient to the changing of variable type.
4.9 Scripts

For implementation details, see the source code in the directory scr.

4.9.1 compile_tu

Parameters

- projectDirectory – directory to be processed
- projectName – name of the project (output directory name as well)

Behavior

The script looks for all header files and source files in directory projectDirectory. Header files are files with suffix “.h” or “.hpp”, source files are files with suffix “.c”, “.cc” or “.cpp”. For each source file, the GNU C compiler is run in order to get translation units. Header file directories are used as include paths, so include directives can be satisfied. Final translation units are stored in directory projectName; if this directory does not exist, it is created.

The script should be run in a directory with no files, because *.tu units are stored in an active directory and in one part of the script, all files with suffixes “.tu” and “.o” are moved or deleted.

The tested GNU C Compiler version for providing the required abstract syntax tree dump is 4.1.1. For version 3.4, use script compile_original, which has the same parameters and behavior.

4.9.2 compile_original

The script has the same parameters and behavior as script compile_tu, but instead of the option -fdump-translation-unit and a dump produced with suffix “.tu”, it uses option -fdump-tree-original and produces a dump with suffix “.original”.

The tested GNU C Compiler version for providing the required abstract syntax tree dump is 3.4.6. For version 4.1, use script compile_tu.
4.9.3  analyze_similarities

Parameters

- inputDirectory – directory containing project directories to be processed

Behavior

The script analyzes each directory in the input directory as a project. At first, the compile_tu script is run to create the translation units for the project. Project translation units are then parsed, reduced and analyzed. Finally, correct source names are set using the script set_source_names. The output of this script is a text file with the name of the processed directory and suffix “.hash” containing hashes for the whole project.

There are 2 versions of this script available

- analyze_similarities_gcc_4.1.1
- analyze_similarities_gcc_3.4.6

Each version was tested on the corresponding version of the GNU C Compiler.

The correct script version must be run for each GNU C Compiler version because of incompatibility in between abstract syntax tree dumps in different versions.

4.9.4  set_source_names

Parameters

- hashFile – a file with hashes, where sources have proxy names (source_file_X)
- srcFile – a file containing source file names with a whole path

Behavior

The script replaces source files in file “hashFile” with the real names of the files and whole paths from file “srcFile”.


5 Cases Tested

Basic tests were done on the floating example and on slightly more complex examples of modifications to detect from section Analyses of Modifications to Detect.

Example 2 represents a basic attempt to copy another student's work by changing the comments and variable names. This attempt does not require any programming or technological knowledge.

Example 3 represents a more sophisticated attempt to copy the work by changing program structure using various techniques such as adding variables, moving unrelated blocks of code into different places and splitting parts of the program into separate functions.

The main purpose of this thesis was to prepare a tool to test similarities in projects committed to Operating Systems Seminar. Regrettably, projects are not allowed to be published; however, we present the test results in section Results.
6 Results

This section presents the test results from the test cases mentioned in the previous section. A brief evaluation of the implemented tool for source code similarity detection has also been prepared outlining its strong and weak points.

Example 2 was marked as 100% similar to Example 1. That is the expected result because no structure changes were made. This means that students will not be able to thwart the detection tool by changing comments or modifying variable names.

Example 3 was marked as 40% similar to Example 1. Example 3 is copied from Example 1, but only about 25% of source code is the original and 75% was modified. Therefore 40% similarity of Example 3 is within the expected range for the correct result.

When compared to each other the first time, the projects from the Operating Systems Seminar found to be the most similar were projects based on the same kernel (the source code provided by the teacher at the beginning of the seminar). This kernel source code represents the main part of the program; the student's work only adds the minor parts. That means the tool is correctly able to detect projects with the same main part of the source code.

However, we are interested in comparing student work, so kernel hash signatures should be removed from the comparison. There is, however, a problem with the Operating Systems Seminar that works are based on different kernels and not all the kernels are available. See section Possible Extensions and Future Work for more information about this problem's solution.
Below, you can find a report of the comparison of projects from the Operating System Seminar based on known kernel source code. The results show that in 23 projects the similarity differs from 0% to 100%.

As you can see in the graph, most of the projects are completely different – almost 200 combinations of projects have a similarity of less than 5% – or very different – the next 200 combinations of projects have a similarity of less than 20%. This means that only 22% of all comparisons are more than 20% similar. These represent the set to be checked by the teacher to find out if the projects are truly copied from each other.
6.1 Summary

We fulfilled the objectives of thesis to implement a tool usable for the detection of similarities between projects written in programming languages C and C++. The tool is resilient to source code modifications as defined in the objectives.

The tool was prepared primarily for C and C++, because these are the languages used for the Operating Systems Seminar work that we focused on. The architecture of the tool enables the comparison of projects in different programming languages with block structures and provides mechanisms for efficient usage such as reusing analyzed projects to compare to new works.

From testing a set of 506 comparisons, less than 20% of the comparisons done were more than 20% similar. Similarities in the tested projects shows that most of the projects are different. However, we found also projects with similarities of over 60% which should be investigated to determine the project's uniqueness or plagiarism.
7 Possible Extensions and Future Work

7.1 Unknown Kernel Source Code for Operating Systems Seminar

It is important to have all the source code you want to exclude from comparison. However, we do not have all the kernel source code used in various seminars and we want to compare code from various seminars together.

We can exclude the kernel source code from known kernels, but that does not seem to be enough to get correct results.

A possible solution is to create a tool that is able to gather all the kernel source codes from the projects. The tool should locate all source files that are 100% similar to other source files with the same name in multiple projects. These files should later be checked by the professor and confirmed as kernel sources to be excluded from the project's comparisons.

7.2 Java

Our tool is able to process projects written in Java, but comparing such projects will not produce good results. The reason is that Java source code produces different identifiers than C and C++ source code. What is necessary for effective Java source code comparison is to update the parser component to mark important Java identifiers as statements and update all modules so that they correctly use Java identifiers to produce hash signatures.

7.3 Longterm Maintainability

The solution prepared using the GNU C Compiler enabled faster implementation without the necessity of implementing programming language parsers. However, it has some disadvantages, too. One of the biggest disadvantages we are aware of is the incompatibility of different abstract syntax tree dumps for different versions. With version 4.3.6 we used compiler option -fdump-tree-original and the dump was created in a file with suffix ".original". In version 4.1.1 we used compiler option -fdump-translation-unit, which produces a similar dump in a file with suffix ".tu". In version 4.3.1 we did not find any similar dump.

Our solution uses its own representation of the abstract syntax tree to analyze and compare source code. But for a different abstract syntax tree dump, the whole parser and probably also the reducer have to be rewritten to accept a new format of the dump and to produce the required inner structure of the abstract syntax tree. This is a considerable amount of work which has to be done each time the dump is changed in the new GNU C Compiler version.

The solution to this problem is to use a 3rd party tool to parse source code into a defined structure and use that structure or parse and reduce it into our inner structure (but only once). At the time of preparing our tool, we tried to find a good 3rd party tool, but did not find any. Hopefully the situation will improve in the future. See section Related Work for tested tools and the tools available now, which did not exist or we did not know about at the time we prepared our tool.
7.4 Graphical User Interface (GUI)

One of the possible extensions is the preparation of GUI, which will allow the user to choose a directory with projects as well as an output file path and the setup of analysis and comparison options. However, the main focus of the application is usability for Operating Systems Seminar and no such interface was required.
8 User Manual

This section describes how to use the program to analyze the source code and find similar source code in different projects. By project we mean a set of source code located in the same folder.

8.1 Structure of Projects

All projects are located in the same directory. Each project can have a different inner structure (different files, folders, ...).

8.2 File similarity.incpath

In each project directory, there can be a file created with the name similarity.incpath containing include paths to all the header files used in the project but that are not located inside the project directory. Each include path has to be on a separate line.

Example: 3 directories are included: /include1, /include2 and /include3. The file similarity.incpath will contain 3 lines as below.

```
/include1
/include2
/include3
```

8.3 Running Analyses for a Single File (GCC version 4.1.1)

1. Prepare translation units by running `gcc -Wall -fdump-translation-unit -c filename`, where filename is the name of file you want to analyze.

2. Parse the received translation unit by running `bin/parser filename > parsed_file`, where filename is the name of the translation unit from step 1.

3. Run `bin/reducer parsed_file reduced_file` to receive reduced output.

4. Run `bin/analyzer reduced_file` to receive the file with hashes.

5. Run `bin/comparer -s -f hashfile1 hashfile2 > results.html` to produce html output with results.

8.4 Running Analyses for a Single Project

The easiest way to create a file with hash signatures for a single project is to create a special directory, placing the project into it and then continuing to analyze the whole directory with projects (only 1 project in this case). We did not create any scripts for only one project, because the objectives outlined in the thesis are massive directories with several student projects.
8.5 Running Analyses for the Whole Directory (GCC version 4.1.1)

1. Create a new folder and enter it.

2. Run script `scr/analyze_similarities_gcc_4.1.1 directory`, where the directory is the place where the projects are located. The script should be run only from an empty directory, otherwise it will delete files with the suffixes “.o” and/or “.tu”.

3. During script processing, some errors may appear, typically for files with missing includes. Check these errors and fix them to get the best results. This step is optional; program will continue with valid translation units only.

4. Run `bin/comparer -f -s *.hash > results.html` (for more options check the comparer manual). The command will compare all files with project hashes and produce a web page with the results sorted from projects whose comparisons were exactly the same to those that were totally different.

8.6 Excluding Shared Code

In some work, the teacher provides some source code at beginning of the assignment. Students are allowed and furthermore supposed to use the given source code. These could cause false alerts and decrease the quality of comparisons.

All source code that was passed on to the students should be placed into a separate directory and analyzed as a common project (running `scr/analyze_similarities` – see part Running analyses for a single project in this section). The final file with hashes can be passed to the comparer as a kernel hash file using option `-k`. Hashes in such a file (passed through the -k option) are skipped in the comparer and the kernel parts included in projects are ignored.
8.7 Comparisons

The comparer component takes two or more hash files produced by the analyzer component for different projects and creates output with comparisons of every two projects. Both text and HTML outputs are supported (text by default, HTML by using the -f option). For sorted output beginning with the best matches in between projects, the user can use the -s option. The number of best fits shown by the comparer can be set using option -b. For excluding some specific hash signatures from comparison, the user can use option -k and set the file including its hash signatures to be excluded.

Usage

`bin/comparer [-k kernelHashFile] [-s] [-f] file1 file2 ...

Options

- -k file File with hashes to ignore (kernel).
- -b fits Best fits to print (positive number).
- -s Sort results before printing.
- -f Format output as HTML page.

Comparison of projects: skupina-1 and skupina-6

Comparing without reliability

Project skupina-1 has 70% matches with project skupina-6
Project skupina-6 has 43% matches with project skupina-1

Comparing with reliability

Project skupina-1 has 54% reliability matches with project skupina-6
Project skupina-6 has 38% reliability matches with project skupina-1

Best sources fits

<table>
<thead>
<tr>
<th>Percent1 of2</th>
<th>Percent2 of1</th>
<th>RelPercent1 of2</th>
<th>RelPercent2 of1</th>
<th>skupina-1</th>
<th>skupina-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>vtable.c</td>
<td>vtable.c</td>
</tr>
<tr>
<td>70</td>
<td>100</td>
<td>40</td>
<td>100</td>
<td>vmalloc.c</td>
<td>vmalloc.c</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>vlist.c</td>
<td>vlist.c</td>
</tr>
</tbody>
</table>

HTML output example
9 Programmer's Manual

9.1 Divide Module Tutorial

The example of the module below (DivideSingle) cuts each node separately. The header file only defines functions getName and divide as mentioned in section Divide Module Interface, the source file defines behavior. For divide decisions, you can use the properties of class SimilarityNode, especially type, identifier and properties.

9.1.1 Example of the Module's Header File DivideSingle.h

```cpp
#include "SimilarityForest.h"

class DivideSingle : DivideAlgorithm
{
    public:
        string getName() const;
        SimilarityForest* divide(SimilarityNode* tree) const;
};
```
9.1.2 Example of the Module's Source File DivideSingle.cpp

```cpp
#include "DivideSingle.h"

string DivideSingle::getName() const
{
    return "DivideSingle";
}

/**
 * This module divides each node into its own tree.
 */
SimilarityForest* DivideSingle::divide(SimilarityNode* tree) const
{
    SimilarityForest* forest = new SimilarityForest();
    list<SimilarityNode*> queue;
    SimilarityNode* node;
    SimilarityNode* ins;

    queue.clear();
    queue.push_back(tree);

    while (queue.size() > 0)
    {
        node = queue.front();
        queue.pop_front();

        if (node->getHelp() != 0)
        {
            continue;
        }

        node->setHelp(1);

        // Make work.
        ins = new SimilarityNode(node);
        forest->storeNode(ins);
        forest->addTree(ins);

        for (list<SimilarityNode*>::const_iterator iter = node->getSuccessors()->begin(), last = node->getSuccessors()->end(); iter != last; iter++)
        {
            if ((*iter)->getHelp() == 0)
            {
                queue.push_back(*iter);
            }
        }
    }

    return forest;
}
```
9.2 Hash Module Tutorial

The example of the module below (HashCount) counts the number of different types of nodes. The header file only defines functions getName and hash as previously mentioned, the source file defines behavior. For hash decisions, you can use the properties of class SimilarityNode, especially type, identifier and properties.

9.2.1 Example of the Module's Header File HashCount.h

```cpp
#include <string>
#include "SimilarityForest.h"

class HashCount : HashAlgorithm
{
  public:
    string getName() const;
    Hash hash(SimilarityNode* tree) const;
};
```

9.2.2 Example of the Module's Source File HashCount.cpp

```cpp
#include "Utils.h"
#include "HashCount.h"

string HashCount::getName() const
{
  return "HashCount";
}

/**
 * Hash has 3 parts: count of nodes, count of called nodes, count of modified nodes.
 */
Hash HashCount::hash(SimilarityNode* tree) const
{
  list<SimilarityNode*> queue;
  SimilarityNode* node = NULL;
  int count = 0;
  int call = 0;
  int modify = 0;

  string output;
  int reliability = 0;

  queue.clear();
  queue.push_back(tree);

  (continue on the next page)```
while (queue.size() > 0)
{
    node = queue.front();
    queue.pop_front();

    if (node->getHelp() != 0)
    {
        continue;
    }

    node->setHelp(1);
    ++count;

    if (node->getType() == CALL_EXPRESSION)
    {
        ++call;
    }

    if (node->getType() == MODIFY_EXPRESSION)
    {
        ++modify;
    }

    for (list< SimilarityNode* >::const_iterator iter = node->getSuccessors()->begin(), last = node->getSuccessors()->end(); iter != last; iter++)
    {
        if ((*iter)->getHelp() == 0)
        {
            queue.push_back(*iter);
        }
    }

    output = itos(count) + ":" + itos(call) + ":" + itos(modify);
    reliability = min(MAX_RELIABILITY * max(count - 5, 0) / 25, MAX_RELIABILITY);

    return Hash(tree->getNumber(), reliability, output);
}
10 Literature and Other Sources

[1] GCC: GNU C Compiler
   - http://gcc.gnu.org/onlinedocs/gccint

[2] GXL: Graph eXchange Language
   - Ric Holt, Andy Schürr, Susan Elliott Sim, Andreas Winter
   - http://www.gupro.de/GXL/tools/tools.html

[3] XOgastan: Xml-Oriented Gnu AST Analyzer
   - Gianluca Masone, University of Sannio, Benevento, Italy
   - http://www.ing.unisannio.it/villano/students/masone/

   - Gasta User's guide

   - http://swag.uwaterloo.ca/~cppx/

[6] SiSSy/Recoder/Java2PCM

[7] SIM
   - Dick Grune, VU University Amsterdam

[8] Artistic Style
   - http://astyle.sourceforge.net/

[9] Doxygen
   - Source code documentation generator tool
   - http://www.stack.nl/~dimitri/doxygen/
11 Attachments on CD

- Formatted program source code [8]
- Program binaries
- Program documentation in HTML format [9]
- Thesis in PDF format
- Examples of same and similar programs
- GAST dump example