UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Software Engineering Curriculum

Karl Kilgi

Code Clone Detection Using Wavelets

Master's Thesis (30 ECTS)

Supervisor: Siim Karus, PhD

Tartu 2014

Code Clone Detection Using Wavelets

Abstract:

For different reasons, developers may produce code that is cloned. It has a negative impact on code quality and code clones are one of the most frequent problems that may appear in a software project. Code clones have an influence on the difficulty of maintaining code, which results in loss of time and money. In this thesis we will propose solution for code clone detection by using wavelet analysis. Wavelet analysis has been found to be extremely useful for clone detection in image processing and financial market analysis. Wavelets have the benefit of allowing comparisons than span different scales and strength. It also benefits a lot from parallelisation, which has become more affordable thanks to GPU computing and cloud computing advances. Thus, it makes sense to evaluate wavelet analysis for solving problems in software engineering as well. The code clone detection algorithm made in this thesis will be language independent and its usefulness will be evaluated in finding different type of clones and compared against existing solutions.

Keywords:

Wavelet analysis, code clone detection

Koodikloonide tuvastamine lainikutega

Lühikokkuvõte:

Erinevatel põhjustel võivad arendajad teha koodi, mis on kloon olemasolevast lahendusest. Sellel on negatiivne mõju koodi kvaliteedile mistõttu on sellest saanud üks levinumatest probleemidest, mis leidub tarkvaraprojektis. Koodikloonid mõjutavad koodi hallatavust, mis põhjustab omakorda kaotuse nii ajas kui ka rahas. Selle töö raames pakume välja lahenduse leidmaks koodikloone kasutades lainik analüüsi. Lainik analüüs on kasutusel ja vägagi kasulik kloonide leidmisel pilditöötluses ja finantsturgude analüüsis. Lisaks saab lainik analüüsis kasutada võrdlusi, mis muutuvad erinevatel skaaladel ja tugevustel ning ära kasutada paralleliseerimist, mis on saanud kättesaadavamaks tänu GPU ja pilvearvutuste arengule. Seetõttu on loogiline lainik analsüüsi hinnata ka tarkvaraarenduses. Töö raames loodav koodikloonide leidmise algoritm on keelest sõltumatu ning selle väljundi kasulikkust hinnatakse erinevate kloonide leidmisel ja võrreldakse olemasolevate lahendustega.

Võtmesõnad:

Lainik analüüs, koodikloonide tuvastamine

Table of Contents

1 Introduction

Over the last few decades the amount and need for software have increased rapidly. Alongside it, many procurers of software want to receive it fast with low costs. Therefore, the ways how to develop software faster are developed as well. There are ways how to gain results faster in the field of software development, but some of them lead into bad code quality. This will result in slower future developments, higher chances of producing faults, bigger maintenance gaps and may end up in the need for rewriting the whole project. One of such ways of getting faster results at the expense of quality is code cloning. This thesis is dedicated to finding these code quality lowering parts in a new way.

1.1 Problem Statement

When writing code, it is sometimes easier to clone code instead of refactoring and using common code. Under the pressure of time, developers may go this way to produce results more quickly. Also in bigger projects, duplicates can emerge from unawareness of the existing code. Every line of code will need maintenance at some part of its life cycle and thus it needs time and money. Furthermore, fixing a bug or extending functionality in one place takes less time and has better fault-tolerance than doing it in multiple places. This is because the developers may not know all the places where the clone is copied and thus the bug fix may be done only partially. Hence, identifying such duplicates is an important step towards improving code quality and maintainability.

Researches have shown that many programs contain remarkable amount of duplicated code. Code clones do not directly cause errors and faults, but the inconsistent changes to them can lead to unexpected behaviour of program. There have been different studies to identify how harmful or harmless the clones are [1]. The general view to code clones is that the less duplicates the software have, the better it is. Also, a large scale study has shown that nearly every second unintentional and inconsistent change to a clone results up in fault [2].

The code clones are substantial problem and a major source of faults, unless the duplicates are detected and taken care of. There are several solutions in the field of code clone detection, but the detectors are mostly language dependant and there is low support of finding clones in scripting languages. Also many of the current algorithms have scaling problems for working in real world code bases [3].

1.2 Why Wavelet Analysis?

Wavelets are functions that satisfy certain mathematical requirements and are used in representing data or other functions. Wavelet techniques have not been thoroughly worked out in such applications as practical data analysis [4]. Wavelet analysis has been found to be extremely useful for clone detection in image processing [5] and financial market analysis [6]. Wavelets have the benefit of allowing comparisons that span different scales and strength, which allows us to prioritize and find the bigger clones first. Wavelet analysis also benefits a lot from parallelisation, which has become more affordable thanks to GPU computing and cloud computing advances. The availability for using parallelisation comes from the nature of matrix multiplications.

Wavelet analysis allows us to do transformations that reduce the amount of data needed to be analysed by code clone detector, significantly. This means that the main problem of many code clone detectors, which is the large amount of data needed to be analysed, can be reduced. For example if one would like to find clones from Linux kernel 1.0 [7] with a

code clone detector, that does not do any normalization on the source code and compares each character in its C code separately, it would have to compare 3 666 796 characters with each other. This would result in a number of combinations that is not scalable. Applying wavelet transforms to the dataset reduces the size of data by 2 with each transform. For example using transforms in six levels, we can bring the number of characters down to 57 293. See Table 1 for details about each level. Second thing that talks in favour of wavelets is that the transformations will smoothen out the small differences that the developer may have added into a clone. Thus, it makes sense to evaluate the usefulness of wavelet analysis for solving problems in code clone detection.

Table 1. Amount of data after wavelet transformations.

Transformation level	**No. of characters**
0	3 666 796
1	1 833 398
2	916 699
3	458 349
4	229 174
5	114 587
6	57 293

1.3 Objective

The objective of this thesis is to evaluate the usefulness of wavelet transformations in code clone detection. For this we first create set of programs that carry out the process of code clone detection, using wavelets. The evaluation will be carried out by analysing the capability of the algorithm finding different type of clones in different levels and comparing the results with alternative code clone detectors.

To use wavelet analysis for code clone detection we first need to design a way to represent code as multidimensional numeric series. This will be done by creating a program that converts each character in the file into its ASCII code form. Then the output will be given to an algorithm which performs wavelet transformations on it. After transformations a match detection algorithm will locate the possible clones and save them into a file. The objective is to find not only 100% exact code clones, but clones that have some syntactic difference as well. This will be achieved by attaching together closely positioned clones. Detected clones will be handed into a program that will generate a HTML view, which shows the clones and indicates the locations of existing duplicates in project. The analysis algorithms are written in R and Java.

We expect the language-independence and performance of our code-clones detection method to compensate for possible loss of accuracy caused by language ignorance. This would make our approach preferred in scenarios with larger volumes of code or language-diversity.

Current thesis consists four major parts. We begin with giving an overview of current code clone detection process and it is described how different methods do it. Additionally, the part will include an overview and examples of wavelet transformation that we are going to use in this thesis. The next part of the thesis has detailed description about the solution made in this thesis. It will provide examples on how to use it and what are the outputs. The description is followed by a two-step evaluation of the solution. In the first step there is a manual analysis of the solution through different scenarios and in the second, there will be comparison with existing solutions and real datasets. Finally, the last part of the thesis contains the conclusion and gives leads to possible future works.

2 Background

We start by describing the relevant work that has been done in code clone detection and in the field of wavelet analysis. Before heading into them, it is important to understand the code clone terminology [8].

1. A **code fragment** is any sequence of code. The granularity can differ from whole functions to simple sequence of statements. A code fragment will be represented as filename, beginning character position and the length of the fragment.
2. A **code clone** is a code fragment that is similar by some given definition of similarity to another code fragment. If there are two similar fragments, they form a clone pair and when more fragments are similar, they form a clone class or clone group.
3. There are two main kinds of code fragment similarity. The code fragments can be similar to their text or their functionality. The first one occurs usually when code is produced by copy-paste method. The other one may happen when programmer don't know the existing solution and implements it differently, but still has the same inputs and outputs. We can classify these similarities to four kind of clone types:
   a) Code fragments which are identical, except variations in comments, layout and whitespaces.
   b) Fragments which are syntactically identical, except for variations in identifiers, literals, types.
   c) Copied identical fragments which have modifications in changed, added or removed statements, literals, identifiers and types.
   d) Two fragments which perform the same computation, but are implemented by using different syntactic variants.

2.1 Clone Detection Process

A code clone detector looks for code fragments that have high rate of similarity. A big problem is, that the detector does not know, which fragments may occur more than once, so it should compare every possible fragment with every other possible fragment. This kind of comparison is computationally expensive thus several things are done before looking for actual clones. The code clone detection process usually follows a process that has six basic steps that can be seen on Figure 1. The steps are briefly described in the following sections. [8] [9]

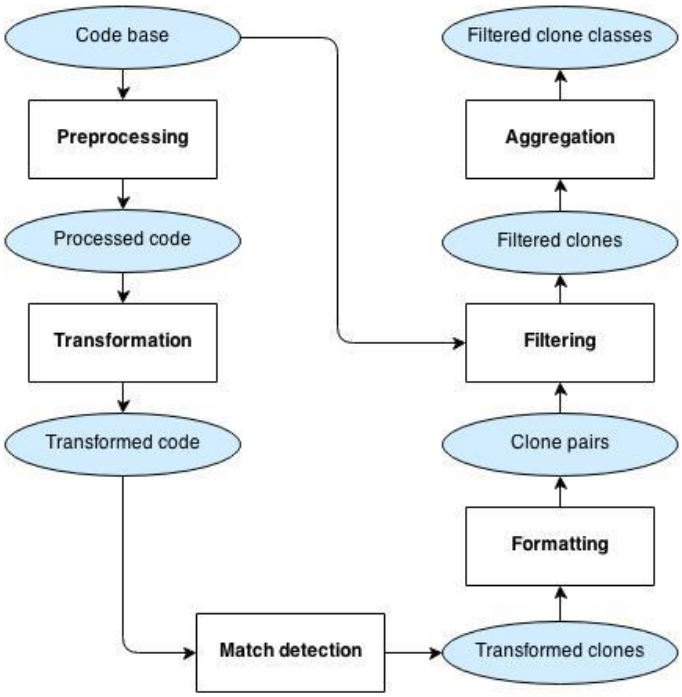

Figure 1. Code clone detection process.

Pre-processing

Pre-processing will help to improve the detector by reducing the amount of data needed to be analysed. There are three main objectives of pre-processing. First thing it should do, is to remove uninteresting parts, such as sections of source code that do not carry the information how a method will work (e.g. import statements in Java). After removing uninteresting code, sets of source units are left remaining. Secondly it has to determine the granularity of source units. Source units can be methods, files, classes, statements and so on. But these source units may need to be partitioned into smaller pieces. Third and final part of pre-processing is to determine the comparison units.

Transformation

Usually the comparison units are different from textual and it is needed to transform them into appropriate form. The transformation to intermediate form is often called extraction in the reverse engineering community and its purpose is to transform the code to suitable form for match detection algorithm. The extraction has roughly three types - tokenization, parsing, and control and data flow analysis. Some tools also use additional normalizations for cleaning up the code. It can vary depending on needs - some are simple and remove only whitespaces, new lines and comments but others also involve normalizing the identifiers, structure and more complex transformations [10]. The normalization can be done before or after extraction.

Match Detection

The code which is transformed through extraction and normalization are split into units and given to comparison algorithm. Similar comparison units are often joined to form larger units. There are techniques using fixed granularity such as block or function, that have predetermined clone unit, and tools that use free granularity that have comparison units limited with some kind of threshold. The output of match detection is usually pairs of transformed clone candidates and also the coordinates to the clones.

Formatting

The list of transformed code clones are converted back to corresponding clones in original code base. The clones are mapped to their positions in original files.

Post-processing / Filtering

Code clones are analysed and ranked manually or using automated heuristics. Manual analysis contains filtering out false positive clones by human expert. Visualisation of code clone can help speeding up this step. In order to rank or filter out clone candidates automatically, different heuristics of code can be used - length, diversity, frequency, or other characteristics of clones.

Aggregation

Some tools return the pairs of clones directly, but in order to reduce the amount of data or to perform subsequent analysis or gather overview statistics about the code quality and project status, clones can be a aggregated with different methods into classes.

Depending on the detector, it may skip few steps or find an alternative to solve problem differently than mentioned above. Also the code clone detector made in this thesis is going to skip few of them and it is putting focus on improving the step of transformation and analysing the results of transforming the code into new form.

2.2 Current Code Clone Detection Methods

There are many different code clone detection methods proposed, but based on their techniques, these methods can be classified into four categories: semantic, syntactic, textual and lexical. The following sections will describe each of them briefly and give references to existing solutions from which two of them are later on used as comparison point to evaluate clone detection method proposed in this thesis.

Lexical approaches

Lexical approach starts by transforming the source code into series of tokens, using compiler-style lexical analysis. The output is then investigated for similar sub-sequences and if they are found, corresponding original code is marked as clone [8]. For example, a code clone detector CCFinder [11], which is using lexical approach, each line of code is divided into tokens respectively to the lexical rules of programming language. It will also remove the white spaces and comments to keep only the interesting part of the code. The tokens from all source files are concatenated, so that finding clones takes place like in single file. The token sequences are then transformed – removed, changed or added, based on the transformation rules. After that, each identifier that is related to types, variables or constants, will be replaced with a special token. Figure 2 shows the example flow of lexical technique. Due to not taking the syntax into account, token based techniques may find clones that overlap different syntactic units. This can be avoided by using pre-processing [12] or post-processing [13].

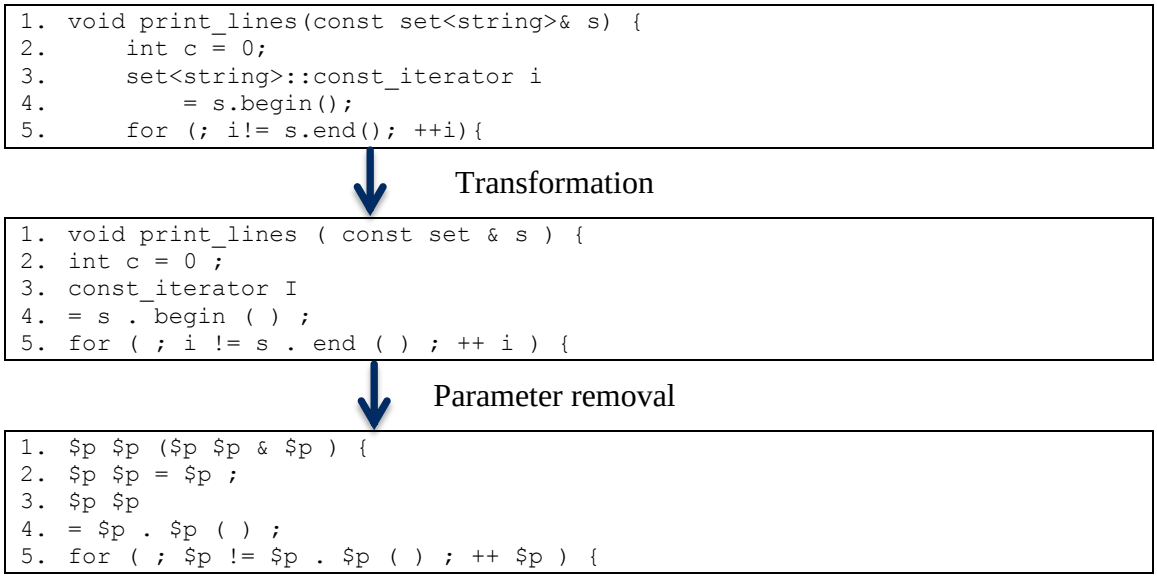

Figure 2. Flow of lexical approach technique [11].

Syntactic approaches

Syntactic approaches use a parser for converting the source code into abstract syntax trees or parse trees, which can be processed by tree-matching or structural metrics for finding clones [8]. Tree-matching contains finding clones by searching similar sub-trees. As one of the pioneering approaches [14], a compiler generator is used to form a constructor for parse trees. The sub-trees can be hashed, which helps to reduce the number of needed comparisons. Compared to parse trees, rather than comparing code directly, metrics approach gathers number of metrics about the code fragment. Then it will compare the results with each other. For example, metrics can be calculated from expressions, names, and control flow of functions [15]. A clone is found when two parts have same or high similarity in metrics values.

Textual approaches

Textual approaches do not use or use just a little transformation before starting actual comparison and match detection, which means that often raw code is used in clone detection process. For example, there is a text-based clone detection approach [9] that uses "fingerprints" on substrings of the source code. In this case, hashing is used to generate fingerprints and whole lines are compared to each other textually. Another technique for textual approach is to use dot plots [10]. It is a chart where both axes represent source entities. Lines of the program are comparison entities and there is a dot at the coordinate, where entities are equal. The dot plots can be visualized to show the clone information – clones are identified as diagonals in plots. This technique can be automated by using pattern matching on dot plots to compare the lines. Some of the approaches, like NiCad [16] are language dependent, which means that they can use their knowledge for selecting comparison units from blocks or functions. Language independent solutions, like Simian [17], are making the comparisons between lines.

Semantic approaches

Semantic approaches are trying to resolve the problem, when the code performs the same computation, like in code snippets in Figure 3, but for example one of them has some extra statements or it is implemented differently. Code clone detector techniques mentioned before are having difficulties finding these types of clones. Semantic clone detection algorithms are trying to locate semantic clones with a less strict, semantics preserving definition of similar

code [3]. Compared to comparing sequences or similar sub-trees, semantic techniques are operating on program dependence graphs [18]. It is a representation in which the statements and flow predicates are as nodes and data and control dependencies are encodes as edges. A clone is found when a similar graph is detected.

```
1.  public int getPow ( int i, int pow) {
2.      int base = 1;
3.      int result = i;
4.      while ( base < pow ) {
5.          result *= i;
6.          base += 1;
7.      }
8.      System.out.Println( "The result is" + result );
9.      return result;
10.}
```

```
1. public int getPow ( int i, int pow) {
2.      int base = 1;
3.      if (n == 0) {
4.         return base;
5.      } else {
6.         return i * getPow(i, pow - 1);
7.      }
8. }
```

Figure 3. A semantic clone pair.

2.3 Wavelet Analysis

Wavelet analysis is analysis of signals that uses the decomposition of signal to wavelet coefficients and scaling coefficients based on defined wavelet filters. The decomposition can be done on the coefficients until the filter length is longer than the resulting wavelet vector [20]. In this section we will describe the properties of selected wavelet analysis and give examples of how does the data will be transformed.

Haar Wavelet

The first discrete wavelet transforms have been used from 1910, when Hungarian mathematician Alfred Haar introduced them [19]. In this thesis we are going to apply Haar wavelet transforms to the data.

Reason for choosing Haar wavelets are due to its simplicity and simple interpretation. Applying wavelet transformations means that we use discrete shift when matching the series with the wavelet [20]. See Figure 4 for example of how does applying the wavelet transformations will change the data. We can consider in this case, that the original data is the ASCII values of code fragment that is 26 characters long. The transformations will start from the last element by adding together this and previous element. Then they are divided by two and we get the first level transformation. From the figure we see that the last two elements in the original data are 78 and 42. Adding them together and dividing by two gives us 60, which we will see in the level 1 transformed data as a last element. But what happens, if there are odd numbers of data that we have to transform? We see that from transforming level 1 to level 2, where the first element of level 1 is left out and not included into level 2 because we don't have any elements to transform it with.

Figure 4. Performing wavelet transforms on data.

Original: 112 29 10 14 82 74 2 42 94 99 94 92 81 84 92 58 27 57 48 84 72 94 92 10 78 42

Level 1: 70.5 12 78 22 96.5 93 82.5 75 42 66 83 51 60

Level 2: 45 59.25 87.75 58.5 74.5 55.5

Level 3: 52.125 73.125 65

The data and possible clones that we are going to transform are not always placed the in the same way, which means that if there are different (odd) number of characters after the clone, it will be transformed differently. See Figure 5 for offset transformations of the same data except it does not have the last element 42. This means that we are making the transformations with one element in offset. If we compere the level 1 of two figures, we see that they are totally different, but if we start going to higher levels, we see that the values are going to get back to closer again. When applying wavelet transformations to characters in code, we will have thousands of elements to transform, which is a lot more than in our example. This means that we can decompose the code to higher than 3rd level and it will help us to smoothen out the differences of being offset. There can be still cases, where the transformations can be offset more than only 1 character, which would Thus we have to find a reasonable amount of offsets that we are going to use for finding code clones.

Figure 5. Performing wavelet transforms on offset compared to Figure 4.

Original: 112 29 10 14 82 74 2 42 94 99 94 92 81 84 92 58 27 57 48 84 72 94 92 10 78

Level 1: 19.5 48 38 68 96.5 86.5 88 42.5 52.5 78 93 44

Level 2: 33.75 53 91.5 65.25 65.25 68.5

Level 3: 43.375 78.375 66.875

Additionally to offset, the small differences between characters will be smoothened out as well. For example characters i and j are next to each other in alphabet, which means their ASCII values are also close – 105 and 106. This means that the difference that they carry on through wavelet transforms would be 0.5 in level 1, 0.25 in level 2, 0.125 in level 3 and so on. The same goes to characters that are next to each other and their position has been changed.

We saw from the sample data, that applying wavelet transforms may produce being on offset or losing some data from the beginning of the wavelets. Thus this is one of the biggest challenges that we are going to face when we are applying wavelet transformations on code. Furthermore, we have to evaluate how does and which kind of data carries on to higher levels and is it possible to find out code clones from that.

3 Code Clone Detection Using Wavelets

In this chapter the process and tool used for code clone detection by using discrete wavelet transformation are described. The process contains four steps - data transformation to numeric form, discrete wavelet transformation, match detection and clone representation. We have chosen to keep these steps as different programs rather than making one big one, because in this way it is easy to replace, improve or reuse them in future works.

3.1 Transformation to Numeric Form

To use discrete wavelet transformation, the input data has to be in numeric form. For that a Java program was made, that takes an argument that contains the path to the project. All the files from the folder path are gathered recursively. The program can also take in an optional argument, specifying the extension of files to be converted. For recognizing the file extension in this Java program and the following ones, Apache Commons IO library [21] is used. Depending on the characteristics of the project that is going to be transformed, another optional argument can be given to the converter, which enables removing all whitespaces, tabs and newline characters. This may be reasonable for use with projects, where these types of characters do not carry important information. For example in project Weltab [22], which is written in C, removing whitespaces, tabs and newline characters reduces the amount of converted data about 33%. Less data results in speeding up wavelet transformation and clone detection, without any significant changes in clones that are found. After gathering files and applying the character removal, each file gets an id and every string in it is converted by using (byte) cast on the character into ASCII form. The result is printed into csv file named "data.csv" that has header "pid" - for identifying the file and "feature" - for identifying each character in the file. Arguments of this and other Java programs made in this thesis are handled by Apache Commons CLI library [23]. The arguments are pointed out in Table 2.

Table 2. Arguments for running program "converter".

Option	**Argument**	**Required**	**Description**
-s	/path/to/source	True	Path to source
-e	.extension	False *(default = *)*	File extension
-rm	*No value*	False *(default = false)*	Remove white-spaces / newlines true/false

To run the program, it first has to be compiled. For the compilation and execution commands with this java program and the following ones, users of Windows have to change colons between class path libraries with semicolons. The program can be compiled from the project root directory from command line in following way: "*javac -cp lib/commons-cli-1.3.jar:lib/commons-io-2.4.jar:. -d . src/**". The output will be two class files - Converter.class, which contains the logic of conversion and DataWriter.class which has the purpose of writing converted data to .csv file.

Compiled program can be run in the following way: "*java -cp lib/commons-cli-1.3.jar:lib/commons-io-2.4.jar:. Converter -s /home/karl/weltab -e c -rm*". While running,

the program will print out the filename and id assigned to it. After writing the csv file, the program will print out the running time in milliseconds. See Table 3 for example data which can be found from data.csv. All the Java programs contain in their folder .iml file, which can be used for opening the project in Intellij Idea editor.

Table 3. Example output of converted project.

pid	**feature**
1	112
1	97
1	99
…	…
146	104
146	107
146	99

3.2 Discrete Wavelet Transformation

For doing the discrete wavelet transformations, a R script is used [20]. It is modified from its original to perform transformations on a single column and a feature is added to produce output from offset transformations. The offset calculations are done by removing the last character from each pid. To run the script, the output from the last step, data.csv, has to be in same folder as the script. The script has one library dependency – wavelets [24]. Installing packages in R can be done with simple command *install.packages("wavelets")*. This library must be installed before running the script. Two options (see Table 4) can be passed to the script – offset and levels. Levels will specify which transformation levels we wish to see in output. In case offset is present, the transformations will be made user specified number in offset as well. See Figure 6 for visualization of transformation tree with 6 offsets until 5 levels. If the offset option is not present, no offset transformations will be made and the tree would come straight down.

Table 4. Options for running script wavelet_cum_single_continuous_feature.R.

Option	**Argument**	**Required**	**Description**
offset	Number of offset transformations	False *(default = 0)*	Perform number of offset transformations
levels	Number of levels	False *(default = all)*	Count of levels to be written into output

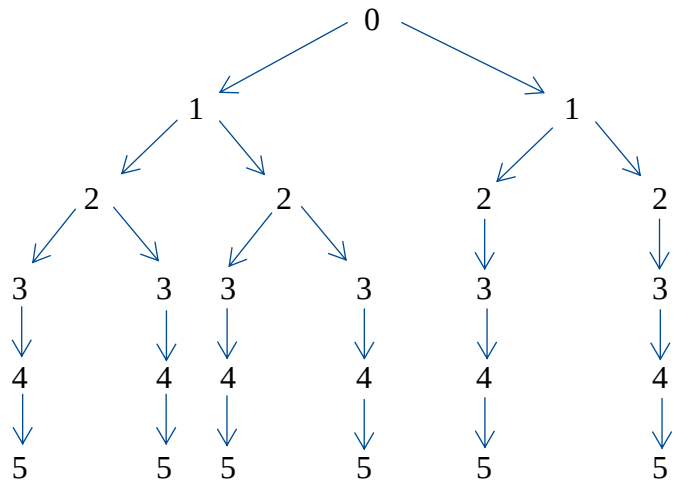

Figure 6. Data transformation tree with 6 offsets

For example the script can be run from command line in following way "*R –vanilla –args offset=2 levels=6 < wavelet_cum_single_continuous_feature.R*". The script will start by creating a folder named wavelet_cum, where the output will be saved. Then it will read in the data and set the column names, which indicate to column that is going to be transformed (feature) and column that contains unique id's (pid). The script will iterate over each pid and performs haar wavelet transforms on its data. After transformations, user specified levels are saved into the csv files. If the user does not specify any levels, all the transformation data will be saved. If the script has offset argument present, it will do the offset transformation as well. All the output files are named accordingly - {offset-value}.Features.dwt.V.feature.csv. If there are no offset used, then the only file will be 0.Features.dwt.V.feature.csv. See Table 5 for example output of transformations.

Table 5. Example output of wavelet transformed data.

x	**seq**	**variable**	**pid**	**coeffi-cient**	**level**	**value**
1	1	feature	1	haar	1	98
2	2	feature	1	haar	1	102
3	3	feature	1	haar	1	102
4	4	feature	1	haar	1	101
…	…	…	…	…	…	…
457	1	feature	1	haar	6	103
457	2	feature	1	haar	6	102
457	3	feature	1	haar	6	98

3.3 Match Detection

To do the match detection, we first have to gather the data from CSV file generated by the previous step. The path to the folder that contains transformed data will be given to the program with an argument. All the arguments for running the match detection program can be seen in Table 6. The match detection program "substring" is written in Java and therefore it starts by putting together a list containing the information about transformations. These lists (see Figure 7) contain strings for each level from output. It is being done by iterating over output file and creating new list or level depending on the change of data. The data from wavelet transforms can have multiple decimal places and therefore it is reasonable to round them. By default the program will round them to two decimal points, but an argument can be passed to the program, which will round it to user specified. Passing the rounding argument 0, the program will transform the integer to alphabetic form by using the ASCII table. By doing the transformation, we will speed up the match detection process, because we have less data to compare. Furthermore, by transforming the data back to alphabetic form, we it will enable a feature that the user can find clones that match accordingly to Soundex algorithm [25].

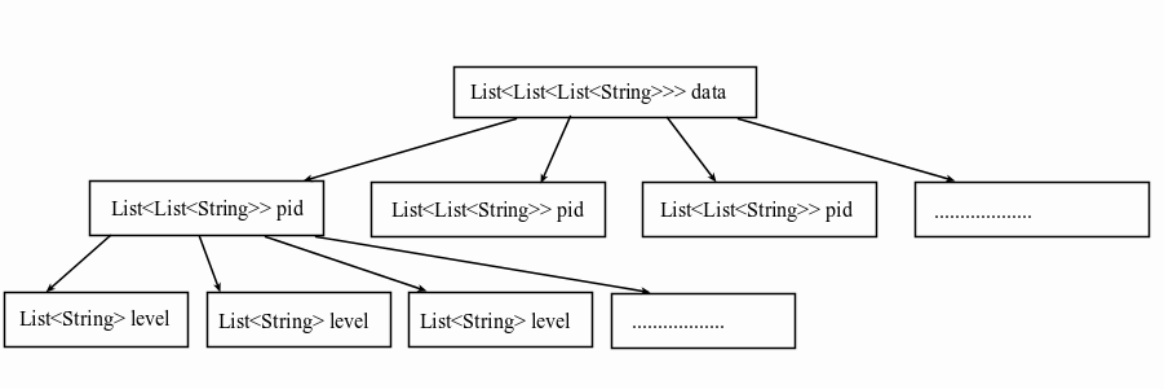

Figure 7. Data structure of output.

After the data is gathered, the algorithm starts comparison from the user specified highest level of wavelet transformation and proceeds to the lower levels if it finds clones. Thus it will have to use less computational power when finding out common substrings. For example, if we start to look for clones from level 6, we will have 64 times less data to search through compared to plain text. When the detector finds a clone in lower level, it will mean that we can be more certain, that we have found a clone. But from which level is it reasonable to start looking clones from? For example in level 10, we will have 1024 characters put together into single one, which means, we may find some false positive clones or we may dismiss lose some smaller clones. Answer for this is given in the next chapter where we experiment our algorithm on different datasets.

Table 6. Arguments for running program “substring”.

Option	Argument	Required	Description
-w	/path/to/wavelets	True	Path to wavelets
-c	filename.csv	False *(default = clones)*	Output file name
-round	Number of decimal points	False *(default = two decimals)*	Round data to given number of decimal points
-soundex	Similarity %	False *(default = false)*	Search for soundex clones with similarity %
-gap	Number of characters	True	Allowed gap between two clones in characters
-set	[level:length:offsets; level:length:offsets]	True	Settings for search level/clone length number of offsets

For searching the clones from wavelet transforms, a simple longest common substring algorithm is used. It is basically an implementation of Naïve algorithm with a modification that when a clone is found we won’t go forward only by one character but we will slide to the end of the clone. Even though it is one of the slowest algorithms and considered as a brute-force method for finding similar substrings, we can still evaluate the usefulness of wavelets when comparing the results from the same algorithm without doing any transformations. See Figure 8 for visualization of substring detection algorithm which is looking for clones with minimum length of 3.

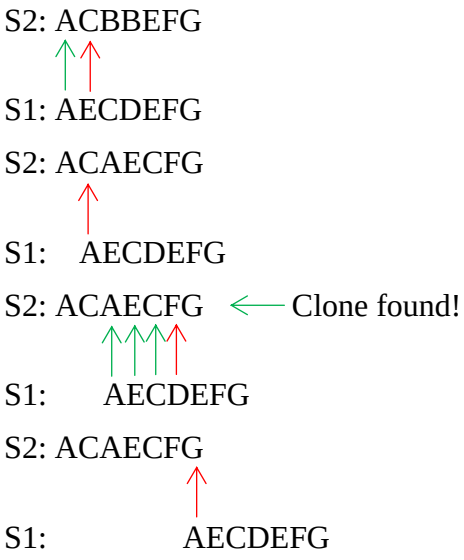

Figure 8. Algorithm used for finding similar substrings.

We will start to compare each element and if there are enough of similar elements in a row, a clone is detected. Due to language independency and different needs of users, we cannot fix any certain length of the clone, so it is left to specify by the user in a required argument that the program takes in. To find the clones, the detector will iterate over the data comparing each feature level with other features same levels. If the offset is specified, it will look clones from offset values as well. The clone detector finds clones from the same feature as well. The clones found from the user specified lowest level are saved to a list. But is it reasonable to start looking the clones with common substring algorithm in higher levels? Like stated before, 6th level has already 64 characters put together, which means changes in code can deviate the results and we won't find the clone in higher levels, even though it exists in lower ones. Let's say that we have transformed data 102.52 and 102.51. These numbers are close, but not the same. This means a common substring won't contain them. To allow this kind of variation, we can use rounding. How much rounding is reasonable to use will be discussed in next chapter.

We have enabled a possibility to use Soundex similarity measures between substrings for finding clones. Note that for using Soundex, we have to use the rounding option back to alphabetic form. The Soundex comparison algorithm returns higher percentage if characters in clones are close to each other and based on that, we will save the clone. We will use existing library SimMetrics [26], for calculating the similarity. This can come useful, if the user needs to find clones that are more imprecise.

Often it will happen that the cloned code has something small, like an identifier, renamed or modified. For that, our program takes in an argument that specifies the allowed gap between two clones so that we can put them together. The length of the different data between clones should not be very long, because otherwise we may find too many false positive code clones that we have no interest in. To be more clear how the whole algorithm works, we have outlined the base method (see Figure 9), Soundex detection (see Figure 10) and common substring clone search (see Figure 11) in human readable way.

```
1. for pid in pids
2.   for compare pid in pids
3.     for level in pid and compare pid
4.       if soundex
5.          look for soundex clones
6.       look for clones
7.       if did not find and offset is set
8.        look for offset clones
9.       if clones do not exist
10.       break and go to next compare pid
11. save clones to csv
```

Figure 9. Base method of match detection.

```
1. for in pid size
2.   for in compare pid size
3.     while both are not empty
4.       data from first += current string
5.       data from second += current string
6.       if soundex metric similarity is over than user specified
7.         save soundex clone to list
8.       else
9.         break and start looking clones from next element
10.  return found soundex clones
```

Figure 10. Soundex method for looking possible clones.

```
1.  found clones from level = false
2.  for in pid size
3.    for in compare pid size
4.      while current pid element equals to current compare pid element
5.        add one to counter
6.      if counter is higher than user specified minimum clone
7.        if not highest level and we are checking if we should search lower
8.          return found clones
9.        for in found clones
10.          if clone is before some of clone according to minimum gap
11.            add reference id to clone that is before current
12.          if clone is after some clone according to minimum gap
13.            remember its id
14.      if it is new clone
15.        add it to clones list
16.      if it is new clone but is after some clone
17.        add it to clones list
18.        set its reference id to clone its following
19.      found clones from level = true
20.      set next starting compare element after this clone
21. return found clones from level
```

Figure 11. Finding clones through longest common substring.

When a clone is found by match detection algorithm, a new object called "Clone" is made. The object contains references to the features in which it takes place, beginning character position in features, transformation level of clone, length and reference id. The clone will have a reference id in case it is following some other clone, which will happen if the gap is small enough. Otherwise the reference id will be marked as 0. The program ends by iterating over the list of clones and saving them into a CSV file. See Table 7 example clones.

Table 7. Example of found clones.

id	feature1	feature2	level	length	posFeature1	posFeature2	ref	offset
1	3	15	1	120	125	599	0	false
2	3	15	1	92	263	737	1	false
3	32	49	3	48	725	253	0	true
…	…	…	…	…	…	…	…	…
86	56	79	1	227	1825	211	0	false
87	82	105	4	1347	165	556	0	true
88	89	92	3	71	621	572	0	true

To run the program, it first has to be compiled. The program can be compiled from the project root directory from command line in following way: "*javac -cp lib/commons-cli-1.3.jar:lib/commons-io-2.4.jar:lib/simmetrics_jar_v1_6_2_d07_02_07.jar:. -d . src/**. The output will be three class files - Substring.class, which contains the logic of match detection, Clone.class which has the purpose of holding clone objects and DataWriter.class which has the purpose of writing clones into cvs file.

For example, compiled program can be run in the following way: "*java -cp lib/commons-cli-1.3.jar:lib/commons-io-2.4.jar:lib/simmetrics_jar_v1_6_2_d07_02_07.jar:. Substring -w /home/karl/msc/single/wavelet_cum/ –round 1 -gap 40 –set [1:64:0;5:160:4;6:124:6]"*. This will run the program using rounding to 1 decimal, allowing different character gap of 40 and it is searching clones from level 1 with length of 64, level 5 with length of 160 and using 4 offsets, level 6 with length of 128 and using 6 offsets. While running, the program will print out the current pid the clone is being searched from. After writing the csv file, the program will print out the running time in milliseconds.

3.4 Displaying the Results

To display the clones in human readable way, we will generate a HTML file that displays the clones found in previous step. The library used for escaping the code for HTML is Apache Commons Lang [27]. For running the program, it is important to use same options and argument as used when converting the initial project and specifying the path to clones. See Table 8 for all arguments that can be passed to the program. The program will append together the clones that have reference ids. It was important to keep them separate until now, because this way we can display them differently than identical clones and paint the different part of the clone to red.

Table 8. Arguments for running program "display"

Option	**Argument**	**Required**	**Description**
-s	/path/to/source	True	Path to source
-e	.extension	False *(default = *)*	File extension
-out	output.html	False (*default = clones.html*)	Output file name
-rm	*No value*	False *(default = false)*	Remove whitespaces true/false
-c	/path/to/clones.csv	True	Path to clones
-restore	*No value*	False *(default = false)*	Restore formatting true/false
-min	Size in characters	False *(default = all)*	Minimum size of clone do be displayed

First two columns of the HTML table have links to the files, that the clones were found in and also a reference to the starting character position in file. The third column contains information about the levels the clone was found in. Fourth column displays the length of clone, excluding the length of different part if there are any. The final column has a small two-row table inside of it and it contains the clones that were found. Top one will be from the first and lower from second file. The program will generate different styles for identical clones (see Figure 12) and for clones that are appended together and contain gaps between them (see Figure 13).

lansxx.c:13718	r11tmp.c:13628	1	400	lhead(ioffic,&lines,&ipage,date,time); lprint(filereport); lines += 2; /* print column totals */ smove(longrep,0," ",0,1); lprint(filereport); smove(longrep,0,"0County totals:",0,15); for (k=1;k<(nform+1);k++) { cvicl(form[TOTAL][k],longrep,(short) form[POSN][k],7); form[TOTAL][k] = 0L; }; if (lines+3 >=
				shead(ioffic,&lines,&ipage,date,time); lprint(filereport); lines += 2; /* print column totals */ smove(longrep,0," ",0,1); lprint(filereport); smove(longrep,0,"0County totals:",0,15); for (k=1;k<(nform+1);k++) { cvicl(form[TOTAL][k],longrep,(short) form[POSN][k],7); form[TOTAL][k] = 0L; }; if (lines+3 >=
lansxx.c:14518	r11tmp.c:14426	1	620	*/ int ioffic; int *lines; int *ipage; char date[],time[]; { (*ipage)++; /* print heading */ cvic(*ipage,headc,122,4); lhprint(filereport,headc); /* office line */ blkbuf(130,headO); smove(headO,0,"0Office:",0,8); gtoffi(ioffic,headO,10); lhprint(filereport,headO); /* print candidate heading lines */ smove(head1,0,"0",0,1); lhprint(filereport,head1); lhprint(filereport,head2); lhprint(filereport,head3); /* set lines used for new page */ *lines = 7; return; } /* ==================== *
				*/ int ioffic; int *lines; int *ipage; char date[],time[]; { (*ipage)++; /* print heading */ cvic(*ipage,headc,122,4); lhprint(filereport,headc); /* office line */ blkbuf(130,headO); smove(headO,0,"0Office:",0,8); gtoffi(ioffic,headO,10); lhprint(filereport,headO); /* print candidate heading lines */ smove(head1,0,"0",0,1); lhprint(filereport,head1); lhprint(filereport,head2); lhprint(filereport,head3); /* set lines used for new page */ *lines = 7; return; } /* ==================== *

Figure 12. Displaying identical clones.

The table rows of clones that have gaps inside of them, have blue background colour, to help the user to easily see them. Also the parts that are not part of the clones are painted red. The red part may contain also strings that are identical, but this happens because these parts are two small to be classified as clones. For example in Figure 5 we see that in the first clone pair, both red parts contain similar strings "*fficial Cumulative",0*", this has happened because previous step was configured to find bigger clones than 23 characters long. This may be solved by reducing the minimum size of allowed clone, but it can also result into leading false positive findings that we have no interest in. We will look at adjusting the size of minimum clone in the evaluation part of the thesis.

cumt.c:5378	totl.c:5528	1	974	/* print heading */ smove(report,0,"1",0,1); smove(report,10,"Weltab III",0,10); smove(report,25,"Unofficial Cumulative",0,26); smove(report,57,cmtime,0,8); smove(report,67,cmdate,0,12); cprint(filereport); smove(report,10,"Election:",0,9); smove(report,25,cmelec,0,50); cprint(filereport); /* print number of precincts included in totals */ smove(report,0,"0",0,1); cviczl(cmprec,report,10,3); smove(report,14,"precincts reporting out of ",0,27); cvic(nprec,report,41,3); pcnt = (cmprec * 100.0) / (long) nprec; smove(report,47,"(xxxx.xx %)",0,11); cvec(pcnt,report,48,7,2); smove(report,67,"Page",0,4); cvic(page,report,72,4); cprint(filereport); /* set lines for remaining page */ *lines = 5; return; } /* ========== */ /** showdone **/ void showdone(date,time,initpage) /* produce the un/completed precincts report based on cmprec */ char date[12],time[8]
				/* print heading */ smove(report,0,"1",0,1); smove(report,10,"Weltab III",0,10); smove(report,25,"Official Cumulative",0,19); smove(report,57,cmtime,0,8); smove(report,67,cmdate,0,12); cprint(filereport); smove(report,10,"Election:",0,9); smove(report,25,cmelec,0,50); cprint(filereport); /* print number of precincts included in totals */ smove(report,0,"0",0,1); cviczl(cmprec,report,10,3); smove(report,14,"precincts reporting out of ",0,27); cvic(nprec,report,41,3); pcnt = (cmprec * 100.0) / (long) nprec; smove(report,47,"(xxxx.xx %)",0,11); cvec(pcnt,report,48,7,2); smove(report,67,"Page",0,4); cvic(*ipage,report,72,4); cprint(filereport); /* set lines for remaining page */ *lines = 5; return; } /* ========== */ /** showpoll **/ void showpoll(date,time) /* produce the un/completed precincts report based on cmprec */ char date[12],time[8];
cumt.c:7302	totl.c:7136	1	148	lincr = 1; if (lines+lincr >= 60) dhead(done,&lastun,&lines,&ipage,date,time); lines += lincr; /*
				lincr = 1; if (lines+lincr >= 60) pollhead(&printunit,&lines,&ipage,date,time); lines += lincr; /*

Figure 13. Displaying clones with gaps.

The program can also take in an argument that will restore the formatting of code (whitespaces, newlines) in the HTML form. The spaces are replaced with   and newlines with
 tags. Figure 14 for code with restored formatting. Restoring the formatting will show us the differences between whitespaces and it will give us better understanding where a difference starts and ends. Leaving out the restoring argument can come in handy, when there are big clones and the user wants to see them both simultaneously.

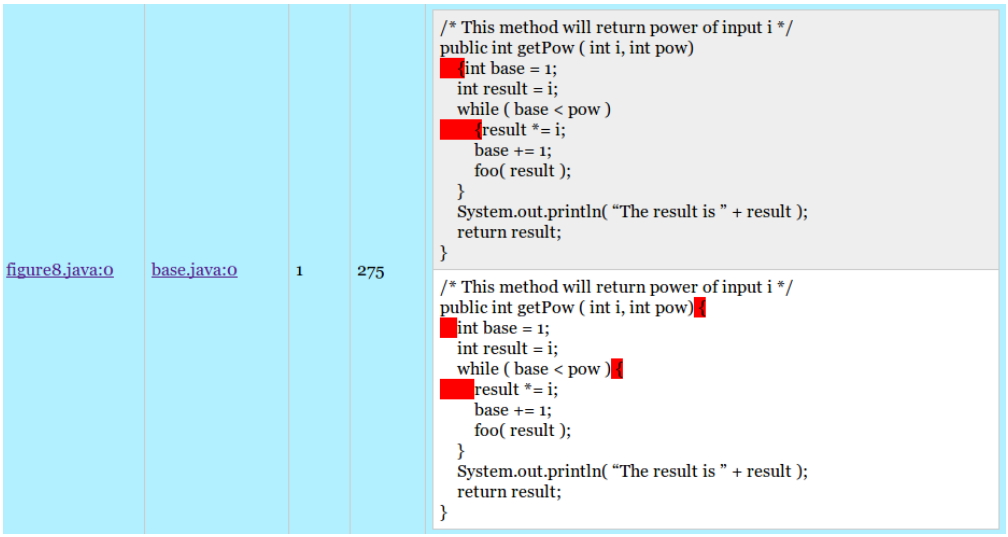

Figure 14. Displaying clones with restored formatting.

To run the program, it first has to be compiled. The program can be compiled from the project root directory from command line in following way: “*javac -cp lib/commons-cli-1.3.jar:lib/commons-io-2.4.jar:lib/commons-lang3-3.3.2.jar:. -d . src/**”. The output will be four class files - Display.class, which contain the main logic for handling the clones, Clone.class, which has the purpose of holding clone objects, SourceFile.class, which has the purpose of holding source files from project and DataWriter.class which has the purpose of writing clones to HTML file.

Compiled program can be run in the following way: “*java -cp lib/commons-cli-1.3.jar:lib/commons-io-2.4.jar:lib/commons-lang3-3.3.2.jar:. Display -c /home/karl/ /msc/substring/clones.csv -s /home/karl/msc/weltab -e c –min 160 -restore*”. This will generate a HTML file with clones that are at least 160 characters long and it will restore the source code formatting in HTML form. The program ends by printing out to the console its running time.

4 Evaluation

To evaluate the clone detection method proposed in this thesis, two different types of evaluations will be carried out. First we will look how the detection works with different types of clones that may occur in everyday life. For this, five scenarios, that contain modifications from small to large, will be composed. In this part of evaluation, the main focus is to understand which kind of clones can be found with our approach and how different changes will affect the results. Secondly we will evaluate how effectively it works with real datasets. The results will be compared with three existing solutions. In this part of evaluation we will use two larger projects and the main focus is to understand how fast and precisely the clones are found. We have chosen this kind of two-step evaluation for two reasons. First, it would be really difficult or even impossible to manually confirm which clones are found in large projects. Secondly, we can use existing large datasets and results of finding clones from them.

4.1 Finding Different Code Clone Types

To evaluate how our method finds different types of clones, a simple fragment of code (see Figure 15) as the base for comparisons, is made. The code is written in Java and it contains a simple function that calculates and returns the input number in a given power. The function also calls out another function foo(int). In the next scenarios this code is modified differently and the results of clone detection are shown and explained. HTML file of the results is included in the appendix. The modifications made to the base code, are not always correct and they may break the functionality. The main purpose of them is to test out our code clone detection algorithm and show the properties of wavelet transformations in practice, rather than being 100% real. Concerning to the clone length, search level and allowed gap between them, we will use the substring program with following arguments: -gap 40 –set [1:32:1;3:32:1;5:32:1;6:64:1]. We will keep the whitespaces, tabs and newlines and we won't use the rounding option. Most of the following scenarios are driven from a study [8] that compares 42 different code clone detection methods. Each code section is transformed by wavelets into 6 levels. Reason why there were only 6 levels is that our code is too small to compose any higher levels.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 15. Base code for evaluation.

Scenario 1: Comments, Whitespaces, Formatting

In the first scenario we will evaluate simple changes in the code. A programmer copies the function three times, each time changing something in it. The first change would be in whitespaces (see Figure 16), second in comments (see Figure 18) and third one in formatting (see Figure 19). All the changes made to the base code in this and following scenarios are marked red.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.       int base = 1;
4.       int result = i;
5.       while ( base < pow ) {
6.            result *= i;
7.            base += 1;
8.            foo( result );
9.       }
10.      System.out.println( "The result is " + result );
11.      return result;
12. }
```

Figure 16. Changes in whitespaces.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.       int base = 1;
4.       int result = i;
5.       while ( base < pow ) {
6.            result *= i;
7.            base += 1;
8.            foo( result );
9.       }
10.      System.out.println( "The result is " + result );
11.      return result;
12. }
```

Figure 17. Results of clone detection with code in Figure 16.

The results of the clone detection with modifications in Figure 16 can be seen of Figure 17. From the results we see, that the whole area where the whitespaces were added are painted red. This has happened because the parts in line 6 and 7, where the base code and modified code are the same, are too small. The next detected clone won't start until line 8, after two whitespaces. This can be resolved by allowing detection of very small clones, but it can result in finding false-positive clones and ones that we do not have any interest in. This problem will reveal in a larger project that we will look at in the next subchapter. More details about the clones we found can be read from Table 9. We can see, that both clones are onset – it is because we added 6 characters to our modified code, which is even. Reason why one of the clones was already found in level 6 and the other one in level 1, is because after first wavelet transform, number of characters we added is 3, which is odd. Thus in the higher levels we end up being deeper offset than by 1.

Table 9. Clones found in Figure 16.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	158	false	1
190	192	102	false	6

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i; //Result
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 18. Changes in comments.

As our code detection algorithm is language independent, the comments are not removed and they are handled as part of the code. The results of the clone detection with modifications in Table 10 are exactly the same as shown in the figure. The clones found by the algorithm can be seen in Table 11. The first clone that was found is in offset because our added comment contains odd number of characters. Again, difference in highest levels can be explained with numbers of added characters.

Table 10. Clones found in Figure 18.

Pos. feature 1	**Pos. feature 2**	**Length**	**Offset**	**Highest level**
0	0	122	True	3
122	130	170	False	6

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow)
3.      {int base = 1;
4.      int result = i;
5.      while ( base < pow )
6.          {result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 19. Changes in formatting.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow)
3.      {int base = 1;
4.      int result = i;
5.      while ( base < pow )
6.          {result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 20. Clones found in Figure 19.

The results of the clone detection with modifications in Figure 19 can be seen of Figure 20. Details about each clone found can be seen in Table 11. Although the human eye detects the change first where the formatting is changed, the real change takes place in the begging of line 3, where the new line is appeared earlier than in the base code. Compared to Figure 16 and Figure 18, we see 3 clones, because gap between two modifications are bigger than 32 characters.

Table 11. Clones found in Figure 19.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	82	false	1
88	88	58	true	5
154	156	134	false	6

Scenario 2: Identifiers, Literals

In the second scenario we will look at renaming of identifiers and literals. A programmer copies the function two times and renames the identifiers and literals. In the first copy (see Figure 21), some of the identifiers name is changed. In the second copy (see Figure 23), the programmer replaces some parameters with expressions.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int j, int pow) {
3.      int base = 1;
4.      int result = j;
5.      while ( base < pow ) {
6.          result *= j;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 21. Changes in identifiers.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int j, int pow) {
3.      int base = 1;
4.      int result = j;
5.      while ( base < pow ) {
6.          result *= j;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 22. Clones found in Figure 21.

The results of the clone detection with modifications in Figure 21 can be seen of Figure 22. Details about each clone found can be seen in Table 12. We can see from the figure that the coloured area is always 2 characters long, although we changed only 1 character. This has happened because a change will affect its surroundings in a wavelet transforms as well. When previous scenarios had different highest clone levels because of the offset, then in this case it is affected by the small size of change, which will be smoothened out in

higher levels. As mentioned in the theoretical part of wavelets, characters i and j are next to each other in alphabet, which means their ASCII values are also close – 105 and 106 and the difference that they carry on through levels gets smaller by every transformation. If we use the rounding option in our match detection, we won't find some of these differences at all.

Table 12. Clones found in Figure 21.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	70	false	5
72	72	48	false	3
122	122	42	false	3
170	170	122	false	6

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= pow * i;
7.          base += 1;
8.          foo( i * i );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 23. Changes in expressions.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *=
```

Figure 24. Clone found in Figure 23.

```
1.                      );
2.      }
3.      System.out.println( "The result is " + result );
4.      return result;
5.  }
```

Figure 25. Clone found in Figure 23.

The results of the clone detection with modifications in Figure 23 can be seen on Figure 24 and Figure 25. The reason, why there are two of them, is because the maximum gap between similar parts is bigger than allowed (40) and the minimum size (32) of clone is also bigger than the part which is same between two differences. This can be fixed by allowing bigger gaps, but it may produce false-positive clones that we have no interest in. For example having language specific start, some user code and language specific end. Searching for small clones and allowing big gap, the detection will mark this as a clone. Details about each clone found can be seen in Table 13.

Table 13. Clones found in Figure 23.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	168	true	3
210	214	82	false	6

Scenario 3: Adding/Deleting a Line

In the third scenario we will look at the effect of adding new line or deleting an existing one. A programmer copies the function two times and changes it by adding (see Figure 26) and removing (see Figure 28) a line.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9           bar( base );
10.     }
11.     System.out.println( "The result is " + result );
12.     return result;
13. }
```

Figure 26. Added a line.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9           bar( base );
10.     }
11.     System.out.println( "The result is " + result );
12.     return result;
13. }
```

Figure 27. Clones found in Figure 26.

Adding a new line to the code won't make any differences to the results of wavelet transforms than making other kind of modifications. The results of the clone detection with modifications in Figure 26 can be seen on Figure 27. Compared to previous results, where whole modifications were marked red, we can see that the ending of the line is not painted red. The reason for this kind of behaviour comes from the previous line – they both have same endings ");". Details about each clone found can be seen in Table 14.

Table 14. Clones found in Figure 26.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	216	true	3
210	230	82	false	6

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int base = 1;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          Deleted line
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 28. Deleted a line.

The clone found from code which has a deleted line won't have anything painted red. It is because it doesn't have anything to paint red. The clone contains of two fragments – before deleted line and after deleted line, without any span between them. Details about clones that were found in Figure 28 can be seen in Table 15. From the details we can see, that the clones found in feature 2 overlap each other. One of the reasons why this has happened, is because the whitespaces before the deleted part are also present in the next line. Second reason is that the line before deleted line also ends with ";" character.

Table 15. Clones found in Figure 28.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	194	true	3
212	188	80	false	6

Scenario 4: Position of the Elements

In fourth scenario we will look at the effect of changing the position of an element. A programmer makes two copies of the program, changing the elements position in the code. In the first copy (see Figure 29), two elements that are next to each other are replaced. In the second copy (see Figure 32), elements that are far from each other, are replaced.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int result = i;
4.      int base = 1;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 29. Reordering two close elements.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      int result = i;
4.      int base = 1;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     return result;
12. }
```

Figure 30. Clones found in Figure 29.

The results of the clone detection with modifications in Figure 29 can be seen on Figure 30. As it can be already expected from the previous results, we will see the different parts painted red. Details about each clone can be seen in Table 16. If we had changed the position of two smaller elements that are next to each other, the clone detection would not find the differences in higher levels. The visualization how the wavelet transform does not care about position of two close characters in higher level, can be seen in Figure 31.

Table 16. Clones found in Figure 29.

Pos. feature 1	**Pos. feature 2**	**Length**	**Offset**	**Highest level**
0	0	92	false	5
122	122	170	false	6

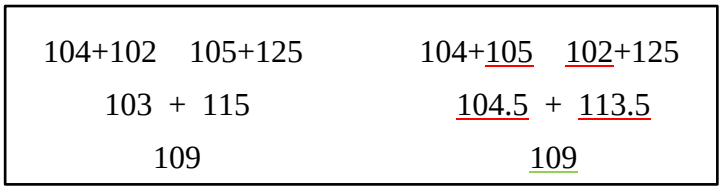

Figure 31. Position change effect of two characters.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      return result;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     int base = 1;
12. }
```

Figure 32. Reordering two elements far from each other.

```
1.  /* This method will return power of input i */
2.  public int getPow ( int i, int pow) {
3.      return result;
4.      int result = i;
5.      while ( base < pow ) {
6.          result *= i;
7.          base += 1;
8.          foo( result );
9.      }
10.     System.out.println( "The result is " + result );
11.     in
```

Figure 33. Clones found from Figure 32.

The results of the clone detection with modifications in Figure 32 can be seen on Figure 33. From the figure we will see, that the ending is cut off, which can be expected, because the second difference is in the end and there is no clone following to it. But why the clone contains last two characters "in" when the base code has "re" in this position? Reason for this comes from the ASCII table, where the codes for i, n are 105, 110 and for r, e are 114, 101. The wavelet transform will have the same result – (105 + 110) / 2 = 107.5, (114 + 101) / 2 = 107.5. Details about each clone found can be seen in Table 17.

Table 17. Clones found in Figure 32.

Pos. feature 1	Pos. feature 2	Length	Offset	Highest level
0	0	88	false	5
102	102	176	true	1

Scenario 5: Semantic Clone

In fifth scenario we will look at the effect of rewriting the code in different way. A programmer does not know that there is an existing solution for finding power of number and rewrites a new one (see Figure 34).

```
1. public int getPower ( int num, int power) {
2.     int base = 1;
3.     if (num == 0) {
4.         return base;
5.     } else {
6.         return num * getPow(num, power - 1);
7.     }
8.  }
```

Figure 34. Rewriting code differently.

Our code clone detection algorithm won't find this kind of clones. In fact, the similar parts are so small, that we won't find anything similar between this and base code. Clones doing the same things, but are implemented differently are very hard to detect and most of the clone detectors do not find them [3]. For finding this clone, it would be important to compose a dependence graph or to compare compiled binary trees.

From the scenarios we have seen how the wavelet based clone detection algorithm handles different types of scenarios that may happen in real life. Also, the scenarios reveal, that finding the clone in different levels can span due to offset, positioning and similarity of the characters. Each test found a part of the clone already in level 5 or 6, which means thanks to wavelet transforms, our match detection algorithm had to compare 32 or 64 times of

less data. Therefore we can identify places to look for other parts of the clone faster and go to details only in these parts of the code.

4.2 Comparison with Existing Clone Detectors

The comparison with existing solutions will be done with two different code clone detectors. They are both text-based like the one composed in this thesis. One of them is also language dependent and therefore its detection algorithm knows and can use shortcuts that can be used for finding clones faster and more precisely. For example, knowing how a function looks like, can lead into using bigger comparison units than a character, which will save lots of time. Also, parts that are not important can be left out (getters, setters, interfaces etc.). The clone detectors are listed in Table 18.

Table 18. Existing clone detectors for comparison.

Name	**Language dependency**	**Method**
NiCAD 3.5 [16]	Java, C	Text-based
Simian 2.3.35 [17]	None	Text-based

Performance

The performance test is done with 1.0.0 Linux kernel, which has 141 361 lines of code in its 282 C files. These files contain 3 666 796 characters, that we are going to convert, transform and look clones from. The variables of environment, where the test was carried out are included in the appendix.

The conversion to ASCII form of these characters will be done by using our converter program with a parameter (-rm) that will remove unnecessary whitespaces, tabs and newlines. This will reduce the number of characters that are going to be converted to 2 857 155. The conversion of them to ASCII form and writing them into a csv file takes about 1.1 seconds.

Next step is to perform the wavelet transformations. As seen from the previous chapter, where we looked for small clones, some of them appeared in level 6, 5 but some due to offset, only in level 1. Considering that the transformed code from long files may be in offset of its clone due to surrounding environment, we cannot be certain if checking level 6 or 5 is enough to say that we do not need to look them from upper levels. Also, it is not reasonable to check level 1 always, because in this way we lose the purpose of wavelets that reduce our amount of data. To be sure, that we find our clone in higher levels, we need to calculate wavelet transformations with different offsets. In this project, one transformation through all levels takes time about 67 seconds. This means, if we want to do it in 7 different offsets, we have to multiply it by 7. See Table 19 for increase of time spent due to offset.

Table 19. Duration of wavelet transformation.

Number of offsets	**Duration**
0	67.5s
1	135s
2	202.5s
…	…
6	472.5s
7	540s

Although doing multiple offset wavelet transformations looks like long, we should not forget one of the wavelets properties – it can be done parallel. Usually developers do not use much of their GPU while programming, which means that we can use its cores for our transformations. For example NVIDIA Quadro 6000 [28] graphics card has 448 cores, what can be used for doing transformations. We cannot use all the cores, but for example, using 400 of them, the time spent on transformations with 7 offsets would theoretically be only 1.18 seconds. Furthermore, when a developer makes changes to a file, only this one has to be transformed, not the whole project.

Before we can head into evaluation of our performance, we should first try out, which are the best settings for our detection algorithm and how many offset transformation we need for finding most of the clones in higher levels. For this, we are going to test the algorithm with different numbers of offsets, rounding and search levels. The thing that we want to achieve is to know early as possible, if there is a possibility of finding a clone. One is certain, if we start to look from the highest level and go up every time by one step, we will use almost the same amount of computing power as looking the clones from plain text. See Table 1 for amount of data needed to be analysed in each level. Thus we will first use the argument for specifying in which levels we are going to look for clones.

From the view of performance we should check the highest level and if we won't find any clones, we should end our searches and move on to the next file and if we do find a clone, we should confirm it in the lower level. One approach is to go to check lowest level after finding a possibility for a clone in a higher level. This would perform well, if we choose a level and conditions that will ensure finding clone in lower level as well. But if the conditions are too concrete, we may have false-negative outcomes. This means that we won't find some of the clones. But if our conditions make too many wrong decisions by detecting false-positive clones, we end up checking too many low levels that contain no clones, therefore spending meaningless time. We should avoid having false-negative results and for this we have to find a point where we rather have false-positive than false-negative results. We will evaluate the number of false-negative results in the next sub-chapter. For now we will compare the false-positive results only between different settings.

In case of false-positive we should check another lower level before checking the lowest. Thus it is important to identify, from which level we should start looking and how long the clone should be in this level, so we can be sure, that we have a clone lying in lower level.

For understanding, which is the best level to confirm the possibility of a clone, series of tests were carried out (see Table 20). The table contains information about the search level, number of characters looked for and how many false-positive checks was made, which means that how many times the algorithm thought that there is a clone in lowest level, but there wasn't any. The last column contains the count of clones that were found from the highest level. We carried out tests in each level until the time spent on looking or number of false-positive results grew drastically.

Table 20. Results with different settings.

ID	Level	Segment length in level	Segment length in code	False-positive results	Time	Count of clones
1	4	8	128	0	446.3	1107
2	4	9	144	0	352.0	913
3	5	4	128	3	174.1s	422
4	5	5	160	0	99.7s	290
5	5	6	192	0	77.0s	174
6	6	2	128	392	1603.5s	508
7	6	3	192	2	37.4s	97
8	6	4	256	0	28.8s	84
9	7	2	256	74	428.5s	202
10	7	3	384	0	11.7s	88
11	7	4	512	0	11.4s	43

From the Table 20 we can see that the number of found clones can vary from 43 – 717, which means that in some cases we won't reach to the places, where the clone lays. The variation of the number of clones found can be explained with false-negative results and the sizes of clones that we are looking for in each level. Lower levels tend to find more small clones than higher levels. Depending on the users need, it may be reasonable to leave out the smaller clones. Most of the times we are interested of finding bigger clones, since these parts have a larger effect on code quality as well. One of the future works with this algorithm can be optimizing in a way like modern search engines work – they will display us the bigger and more interesting results before while searching smaller and less interesting results in the background. For example, the developer may want to start refactoring the bigger clones first and then proceed to the smaller ones.

When we compare the clones that were found by tests 1, 2, 3 and 4, we see that the change of clones was mostly in smaller ones in sizes of 64-160 characters, but the time spent looking for them increased rapidly. We say mostly, because also some larger clones were left out, but this was due to higher number of offsets, that we are going look at after finding the best levels and rounding for our algorithm. Therefore, we can say that the amount of

false-negative in clones with size bigger than 160 is the same with settings 1, 2, 3 and 4. We can leave out settings from tests 1, 2, 3 from our performance test if we are not interested of finding smaller clones. Comparing tests 4 and 5 shows us that 5 has false-negative outcome which means that it has left out some larger clones. It found large clones, that have a lot in common, but if the large clone contains smaller parts, the settings from test 5 did not find them. Also, we see the same thing when comparing results from 6 to 7, 8 and results from 9 to 10, 11, where 7,8,10 and 11 have left out larger clones. Considering that we want to avoid false-negative results, we will exclude these settings from our future tests.

We see from tests 6 and 9 that we would need additional confirmation from lower level before looking into lowest, because both got many false-positive results. Reason why this has happened with them is because when we have 128 characters crunched together, like in level 7, the generalization may give same results, although we won't have the same code behind it. See Figure 35 for two clone pairs that were found in level 7, but have totally different meaning in code and thus giving as false-positive result and therefore we should confirm it.

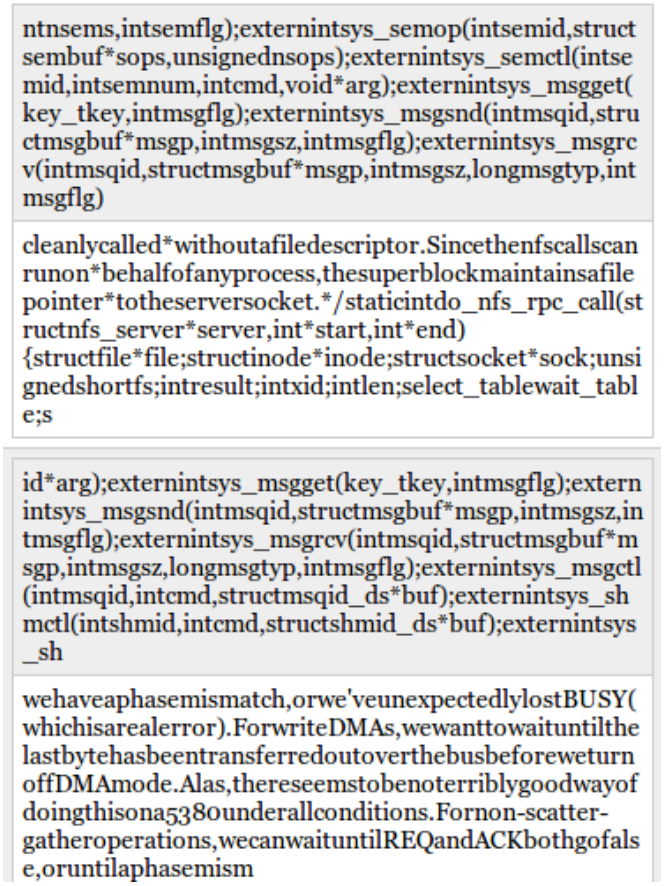

Figure 35. Clone pairs found in level 7.

For confirmation of clones in lower level, we use the level 5, because it did not have any more false-negative results in bigger clones and compared to level 4 and it won't spend as much time. To be more specific, we have chosen to use settings from test 4 because it almost 44% faster than 3, having only the small clones left out. The results of confirmation with test 4 settings for tests 6 and 9 can be seen from Table 21. We see that adding confirmation to test 6, reduced the time almost 43 times due we checked our results in level 5 rather than in lowest level. It also reduces the number of clones found, but it only removes

the small clones. When test 12 has also high number of large clones found, we see that 13 did not find high number of clones because the starting amount from test 9 contained already small clones that were removed by confirmation. Thus we will eliminate the settings from test 9, because it lacks the count of large clones and therefore it has false-negative results.

Table 21. Results from confirmations in level 5.

ID	Previous test ID	Confirmation test ID	False-positive results	Time	Count of clones
12	6	4	0	37.7s	217
13	9	4	0	11.4s	64

Next thing that we can try for improving the count of clones found is rounding. We shall try the option of rounding the results to one decimal point. This would allow us to do more decisions for looking clones from lower level. The test is done with settings that we have not eliminated yet and have the best results – 4, 12 and 13. The results are listed in Table 22. We can see that in each case, the amount of clones found is bigger than without rounding. We also see that the time spent on looking them has increased. Especially in test 15, where the increase was almost 60 seconds. We can suppose that this is because there was high number of false-positive decisions made in level 6 that needed confirmation from level 5. But which types of clones were found thanks to rounding? The manual analysis of them shows, that we have found more clones that are sizes between 64 and 128 characters, which are not close to each other. This was expected to happen, because if a smaller part contains also something else in the beginning or in the end, it transforms little bit differently than its clone. By rounding, we will remove these differences and find them anyways. Thus enabling the rounding can be set by user, depending on the needs. We saw the effect of rounding, but we won't use it in our performance test. It is an optional feature that should be used only in need and right now our main need is not to find small clones.

Table 22. Results of rounding to one decimal point.

ID	Previous test ID	False-positive results	Time	Count of clones
14	4	0	99.1s	314
15	12	0	95.4s	247

Next thing that we are going to look at is applying different amount of offsets to clone detection. For this we use settings from each level that we have not eliminated yet. The test settings are with ID's 4 and 12. See Table 23 for results that list the number of offsets used and the amount of clones found from them with settings from test 4 and Table 24 for results with settings from test 12. We see from both tables that the number of clones found grew with each offset. But in case of adding offsets to test 4 the clones found after using 4 offsets are mostly small. The same goes to adding offsets to test 12, but this time after using 6 offsets. Few larger ones that are found are put together from multiple smaller pieces having lots of differences in them. Although it would be good to see these kinds of clones, we have to put lot of computation power to detect them. For example if we compare tests

23 and 24, we found only found two more clones that are put together from many smaller pieces and between sizes of 512 – 1024 characters. At this point there comes a question if the clones are interesting for us. Can we refactor them and improve the code quality? Most of the times, we cannot, because they have too many different parts mixed in and the similar parts are to take out are too small. Thus we will let it decide by the developer how many offsets to use based on specific language and needs. Due small number larger clones were discarded and from this point forward we did not see any considerable improvement after using 4 and 6 offsets. Thus we will look into details of clones found in tests 17 and 22.

Table 23. Effect of applying offsets to test 4.

ID	Number of offsets	Time	Clones found
16	2	291.4s	1111
17	4	532.6s	1652
18	6	725.8s	1945
19	8	977.7s	2113

Table 24. Effect of applying offsets to test 12.

ID	Number of offsets	Time	Clones found
20	2	159.4s	722
21	4	284.4s	1130
22	6	421.9s	1449
23	8	560.1s	1710
24	10	6117.2s	1910

For selecting the best settings for our performance test, we look at the clones found in test 17 and 22 in detail. We distribute the clones into five categories – smaller than 128, 128 – 256, 256 – 512 and bigger than 512 characters. See Table 26 of which amount different types of clones we found. From that we see, that we found the most interesting – biggest, clones quite equally in both levels. Like said before, depending on the needs, we might want to see these clones, but taking into account, that we had to spend more than 100 seconds for finding them, we won't use these settings in our performance test. So we can say that we achieved the best results considering clone to time ratio by searching 2 characters in level 6, confirming them in level 5 and using 6 offsets.

Table 25. Different sizes of clones found.

ID	**… - 128**	**128 - 256**	**256 - 512**	**512 - …**
18	850	483	200	119
22	755	406	173	115

As we seen from Table 20, that checking level 5 like we did in test 4, it won't make any false-positive results so we are sure that we are going to find a clone below. Thus is it reasonable to check level 1 for confirmation, if we know for sure that there is a clone lying under? Confirmations on level 1 are taking the most time while looking for clones. The benefit that they are giving is showing us where the clone exactly is and which characters are the same and which are different. If the user of the clone detector accepts that the small differences between clones are not always marked and the position of clone may be shifted a little bit, we can save a lot of computation time. For example, from the best result that we have selected (test 22), we can reduce the time spent on looking clones from 412.9s to 90.2s. The last step of our process is generating a HTML file from clones that we have found. This takes about 1.2 seconds. Final arguments for running wavelets transformation are "*offset=6 levels=6*". We have not specified how big the gap should be between two clones so that they would be merged together and in this point it not important to specify either, as its purpose is to give better overview of clones to the user. Through the tests we have used gap of 40, like we used in our scenarios. To sum it up, we are using following arguments for running match detection "*-gap 40 –set [1:64:0;5:160:4;6:124:6]*".

Table 26. Performance results.

Detector	**Total time**	**Clones found**
Wavelets	896.7s	1449
NiCad	19.3s	152
Simian	6.5s	262

Although it seems that our solution is multiple times slower than others, we have to take into account which comparisons we make for finding clones. For example, NiCad first selects blocks or functions for comparison units, which means it has multiple times less comparisons to make. This cannot be done with language independent solution as we do not know, how does a function look like or what is the definition of a code block. Minimum granularity for a unit can be ten lines and based on that, NiCad finds 8788 different blocks to compare. Simian on the other hand, uses lines for comparison units. The data it has to process in Linux kernel would be about 141 361LOC. Also it will remove the lines that are not significant (empty lines, comments, etc.) and ends up with line count of 85 728LOC. When we removed all whitespaces, newlines and tabs, the amount of data we had to process is 2 857 155, which is about 33 times larger compared to 85 728. Simian makes minimum comparisons with two lines, which means me can divide the units by 2. The reason, why other clone detectors do not use character level comparison units, is because without doing a large scale pre-processing, for example like wavelet transformations, the clone detection would not scale.

Another thing, why they get faster results is because the detectors that we used for comparison did not use the basic Naïve algorithm for finding similar parts. For example, NiCad uses an algorithm that is similar to the algorithm used by the Unix *diff* utility [29]. Also we should not forget, that the wavelet transformations can be done parallel, which may reduce, depending on the hardware, almost half from our time total time. Taking all of it into account, it shows why our solution takes so much longer than theirs. But with smaller comparison units we have a benefit that we can pinpoint out the exact characters where a clone starts and where it ends, while other solutions can only reference to functions, blocks or lines. With smaller comparison units we can look also for smaller clones and if we detect small clones near to each other, we can attach them together. The higher number of clones found with our solution comes from small sizes of clones we found and the fact that we did leave anything out – all the comments, include statements etc. can be found from our clones. To give a comparison point what it means to use character based comparison units without the wavelet transformations, we used the same algorithm to search clones from plain text and it took close to 9 hours.

Clone Detection

For evaluating clone detection with wavelet based approach a project named Weltab [22] is chosen. This is an election results program that has approximately 11460 lines. Main reason of selecting Weltab is that it has already been used as in a code clone detection studies [30][16]. Also, Weltab it is written in C that is supported by the selected detector NiCad for comparison. Due to its size, we can analyse the clones manually and look which differences come out. For running the clone detection, we use the same parameters for detection process as previous step, where the performance was evaluated. See Table 27 for count of clones found with each detector.

Table 27. Detection results.

Detector	**Clones found**
Wavelets	1662
NiCad	162
Simian	253

For evaluating, which clones are found with different detectors, we will look at clones that are bigger than 10 lines. NiCad also offers maximum difference threshold level, which we set to be 10%. See Table 28 for the results, where the green marks as found, yellow partly found and red not found. The clones in the table are put together as same clone classes and if there are two or more classes in the same files, this result is put together as well. A clone class forms, when it is found in multiple places. Before that, it is a clone pair. From the table we see that most of the times our clone detector found the same clones as others, but in two cases we performed worse than NiCad and Simian. This was due the specifics of clones, why they did not reveal themselves in higher levels that we used to confirm if it is reasonable to look from down. If we would use only 1 character smaller confirmation in level 5, we would find these two. The clones that NiCad did not find are mostly because it is searching for cloned blocks or functions. For example, when clone is 50 lines long and it is inside of a 100 line block, the clone will be unrevealed. Reason, why Simian did not find all the same clones, is because if a clone contains something different or has some additional information line before 10 equal lines, then it is not detected as a clone. Alt-

hough Simian separated some clones from the non-equal parts, we will still counted it as it found the clone equally as others.

Table 28. Clones found from Weltab.

ID	Files	Results of detectors		
1	vedt.c, xfix.c, vfix.c	Wavelets	NiCad	Simian
2	cnv1a.c, cnv1.c	Wavelets	NiCad	Simian
3	cnv1a.c, canv.c, cnv1.c	Wavelets	NiCad	Simian
4	vful.c, vtot.c	Wavelets	NiCad	Simian
5	ejcn88.c ,lans.c, rsum.c, lansxx.c, r01tmp.c, r11tmp.c, r101tmp.c, r51tmp.c, r26tmp.c, rsumxx.c	Wavelets	NiCad	Simian
6	ejcn88.c, lans.c, lansxx.c	Wavelets	NiCad	Simian
7	rsum.c, r01tmp.c, r51tmp.c, r101tmp.c, r26tmp.c, r11tmp.c, rsumxx.c	Wavelets	NiCad	Simian
8	rsumxx.c, lansxx.c	Wavelets	NiCad	Simian
9	cnv1a, vedt.c	Wavelets	NiCad	Simian
10	canv.c, cnv1.c, cnv1a.c	Wavelets	NiCad	Simian
11	totl.c, cumt.c	Wavelets	NiCad	Simian
12	xfix.c, vedt.c, vfix.c	Wavelets	NiCad	Simian
13	ejcn88.c, r51tmp.c, r11tmp.c, r26tmp.c, rsumxx.c, rsum.c, r01tmp.c, r101tmp.c, lans.c, lansxx.c	Wavelets	NiCad	Simian
14	cnv1.c, vtot.c	Wavelets	NiCad	Simian
15	spol.c, poll.c	Wavelets	NiCad	Simian
16	ccibmx.c, ccibm.c	Wavelets	NiCad	Simian
17	cnv1a.c, lansxx.c	Wavelets	NiCad	Simian
18	sped.c, samp.c	Wavelets	NiCad	Simian
20	rcvr.c, linecoun.c, spol.c	Wavelets	NiCad	Simian

21	baselib.c, baselib.c	Wavelets	NiCad	Simian
22	poll.c, spol.c	Wavelets	NiCad	Simian
23	prec.c, samp.c	Wavelets	NiCad	Simian
24	xfix.c, prec.c	Wavelets	NiCad	Simian
25	lans.c, r51tmp.c, rsumxx.c, r11tmp.c, r26tmp.c, lansxx.c, r101tmp.c, r01tmp.c, rsum.c, ejcn88.c	Wavelets	NiCad	Simian
26	spol.c, xfix.c, vfix.c, vedt.c, poll.c	Wavelets	NiCad	Simian
27	vtot.c, prec.c	Wavelets	NiCad	Simian
28	vedt.c, prec.c	Wavelets	NiCad	Simian

Due to the fact, that our detector is capable of finding smaller clones that the other detectors did not find, we are going to look at them separately. The smaller clones that we found contain small statements, comments, smaller parts with same computation done, include directives etc. All of them are not important to refactor concerning the code quality and some of them even can't be. But let's look at a case, when detecting such a small clone can help us to increase the readability and fault-tolerance of our program. See Figure 36 for a code snippet with simple if statement that can be refactored. From the figure we see, that in some cases the code must call out the method doSomethingSpecial(). Now if we find clones that contain the same statements in if, it would be reasonable to move these checks to foo's model, rather than writing the conditions always out. For proposed solution see Figure 37. If the conditions are written inside of a statement then if the conditions for doSomethinSpecial() change we have to change all the places where we perform these controls. Therefore there is a higher probability of introducing a bug.

```
if (foo.getBar() == null && foo.getCount() != 1 && foo.getWord().equals("bar")){
    doSomethingSpecial(foo);
}
```

Figure 36. Simple statement.

```
if (foo.isSomethingSpecial()){
    doSomethingSpecial(foo);
}

class Foo {
    …
    public boolean isSomethingSpecial() {
       return this.bar == null && this.count != 1 && this.word.equals("bar");
    }
}
```

Figure 37. Refactored statement.

Another type, that the other clone detectors would never find is a clone containing simple string which is hardcoded multiple places into code. For example, if a developer does not know, that there is a string defined somewhere and instead of using it, the value is written into the code again. Now, if in the future some other developer changes the value of the

variable, thinking that this is the only place where it is defined, a bug is introduced because someone had unknowingly duplicated the value which is now left unchanged.

Thanks to language independency, our code clone detection can detect clones not only from programs written in different languages, but it can also find similarities from a project, where for example, some of the configuration is written into xml but some of it is hardcoded in code. Even though there are others language independent solutions, like Simian, they would not find the clone, if they are using a row as a comparison unit. This is because in case of xml, the tags will change the looks of the row. From that, there comes out another benefit of using wavelets transformation and character based comparison units. In the end, this clone detector can be used for any textual similarities, not depending if it is a code. For example it can be used for analysing theses from similar fields to find out if there are any cases of plagiarism or for gathering interesting statistics of how many times was some phrase used.

5 Conclusions

Alongside with the increase of the amount of software produced, particular problems are emerged as well. One of these is code cloning, which is done to achieve faster results at the cost of code quality. Leaving the clones ignored may result in bigger costs in maintenance and future developments. Furthermore, it will lower the fault-tolerance of the system, when making changes to existing code base.

The aim of this thesis was to evaluate the usefulness of the wavelet transformation for code clone detection. This was done by creating multiple tools that helped us to carry out this process. We created a program for transforming code from textual form to numeric, modified a script for performing discrete wavelet transformations on a single column, created a program for finding similar substrings form wavelets and created a program that will generate a HTML file from the clones found. Evaluation of the results was done by going through different scenarios that can happen in real life, testing the performance by trying out different possible settings and comparing the clones that it is capable of finding with alternate solutions.

The evaluation of the solution showed us that even if this approach is slower than the others code clone detectors, we can find the more clones and therefore produce outputs that the others cannot. By using wavelet transformations, we can reduce the amount of data in a way that we can use character level comparisons units, while other clone detectors are using blocks, functions or rows. When having smaller comparison units we can attach the similar parts that are near to each other and pinpoint out the differences between them. Finding smaller clones can be also beneficial for finding the cloned statements, definitions and configurations that are scattered and can be refactored and unified. In conclusion of the evaluation, it can be said that the wavelet based code clone detector produced especially good results in finding small clones fragments with reasonable time.

Current solution made in this thesis is not ready for production use as its purpose was to carry out the evaluation of using wavelet transformations for code clone detection. Therefore future works for extending its usability and efficiency would be optimizing it in different ways. First thing to optimize is giving the results as a modern search engine, where important results are displayed first, while looking for non-important in the background. In our case, important results are bigger clones. Secondly the time spent on doing wavelet transformations can be optimized by calculating the needed levels and offsets only in need. The third thing that can be optimized is the substring search algorithm. Current algorithm is a straight-forward brute-force implementation that does not have the best time efficiency.

Even though current solution may not be the most efficient and user friendly, it can be already used for detecting the clones from a software project. Furthermore, it can be used for analysing the quality of software by looking over the HTML output that our solution generates. I would especially recommend it, if there is a need for finding smaller clones. Due to character based comparison, we can detect such small clones, while others cannot. Although our solution finds out the larger clones as well, there are currently other solutions that may find them faster and more convenient. Since there are only a small selection of language independent code clone detectors, this solution may be also a good tool for finding clones from language that is not widespread and does not have a language specific solution.

This work and algorithm can also be extended to find the wrong usages of code. Like finding the code clones with wavelet analysis, it can be also used for looking how the code is being used. If there are different usages of the same code parts it may be a sign of wrong usage and the user should be warned. For example in one place there is a check for NULL but in another place there is not. Furthermore, the algorithm can be extended to analyse the architecture of the existing code base as - does it follow the requirements? Is it homogeneous? For example if a program has service-oriented architecture and somebody writes the code which is not following its principles, the algorithm would notify the developer.

6 References

[1] C. Kapser and M. W. Godfrey. “Cloning considered harmful” in Proc. WCRE ’06. IEEE, 2006

[2] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, Do Code Clones Matter? Proc. 31st IEEE Int’l Conference Software Engineering, 2009.

[3] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In ICSE 2008, pages 321–330, 2008

[4] Amara Graps, An Introduction to Wavelets, IEEE computational science and engineering, vol. 2, no. 2, 1995.

[5] N. G. Kingsbury, Image processing with complex wavelets, Philos. Trans. Roy. Soc. London Ser. A 357 (1999), 2543–2560.

[6] F.In, S.Kim, The Hedge Ratio and the Empirical Relationship between the Stock and Futures Markets: A New Approach Using Wavelet Analysis, The Journal of Business 2006.

[7] L.Torvalds. Linux kernel 1.0 https://www.kernel.org/pub/linux/kernel/v1.0/ (1994)

Last accessed 12.05.2014.

[8] Chanchal K. Roy and James R. Cordy and Rainer Koschke Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach, 2009

[9] J. Johnson, Identifying Redundancy in Source Code Using Fingerprints, in: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 1993, pp. 171–183 (1993).

[10] S. Ducasse, M. Rieger and S. Demeyer, A Language Independent Approach for Detecting Duplicated Code, in: Proceedings of the 15th International Conference on Software Maintenance, ICSM 1999, pp. 109-118 (1999).

[11] T. Kamiya, S. Kusumoto and K. Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, 28(7):654-670 (2002).

[12] J.R. Cordy, T.R. Dean and N. Synytskyy, Practical Language-Independent Detection of Near-Miss Clones, in: Proceedings of the 14th IBM Centre for Advanced Studies Conference, CASCON 2004, pp. 29-40 (2004).

[13] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue, On Software Maintenance Process Improvement Based on Code Clone Analysis, in: Proceedings of the 4th International Conference on Product Focused Software Process Improvement, PROFES 2002, pp. 185-197 (2002).

[14] I. Baxter, A. Yahin, L. Moura and M. Anna, Clone Detection Using Abstract Syntax Trees, in: Proceedings of the 14th International Conference on Software Maintenance, ICSM 1998, pp. 368-377 (1998).

[15] J. Mayrand, C. Leblanc and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, in: Proceedings of the 12th International Conference on Software Maintenance, ICSM 1996, pp. 244-253 (1996).

[16] C.K. Roy and J.R. Cordy, NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC 2008, pp. 172-181 (2008).

[17] S. Harris. Simian - Similarity Analyser. 2003

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3), 1987

[19] Radomir S. Stanković and Bogdan J. Falkowski, "The Haar wavelet transform: its status and achievements," Computers & Electrical Engineering, vol. 29, no. 1, pp. 25-44, 2003.

[20] S. Karus, "Automatic Means of Identifying Evolutionary Events in Software Development", ICSM, 2013

[21] Apache Commons IO 2.4, http://commons.apache.org/proper/commons-io/

Last accessed 04.04.2014.

[22] S. Bellon and R. Koschke. Detection of Software Clones: Tool Comparison Experiment. http://www.bauhaus-stuttgart.de/clones/ (December, 2007)

Last accessed 06.05.2014

[23] Apache Commons CLI 1.3, http://commons.apache.org/proper/commons-cli/

Last accessed 04.04.2014.

[24] E. Aldrich. wavelets: A package of functions for computing wavelet filters, wavelet transforms and multiresolution analyses. (December 2013)

[25] Zobel, J. and Dart, P. Finding approximate matches in large lexicons. Softw: Pract. Exper., 25: 331–345, (1995).

[26] J. Botha. SimMetrics Library http://sourceforge.net/projects/simmetrics/ (2012)

Last accessed 22.05.2014.

[27] Apache Commons Lang 3, http://commons.apache.org/proper/commons-lang/

Last accessed 04.04.2014.

[28] NVIDIA Quadro 6000 - http://www.nvidia.com/object/product-quadro-6000-us.html

Last accessed 01.05.2014.

[29] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Technical Report 41, Bell Laboratories, 1976.

[30] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. IEEE TSE, 33(9):577-591, 2007.

Appendix

I. Glossary

pid Determines the file
feature A character in file

II. Resources

The source code of the tools for wavelet based code clone detection can be found from the Appendices.zip attached to the thesis. Extracting the zip will have output of 4 folder:

1. converter – contains the program “converter” for converting the data to ASCII form
2. single – contains the R script for running wavelet transforms
3. substring – contains the program “substring” for finding clones
4. display- contains the program “display” for generating HTML view of clones
5. scenarios – contains the HTML file scenarios.html which displays the clones found when doing evaluation in section 4.1

III. Environment

The performance tests were carried out in following environment.

Software:

- Operating system: Ubuntu 14.04 LTS (Trusty Tahr)
- JDK: 1.7.0_45
- R: 3.0.2

Hardware:

- Processor: Intel Core i5-3320M 2.6GHz
- RAM: 16GB RAM 1600MHz PC3-12800 DDR3
- Hard drive: 128GB SSD SATA
- Graphics: Intel HD Graphics 4000

IV. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Karl Kilgi (date of birth: 19.11.1989),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, of my thesis "Code clone detection using wavelets", supervised by Siim Karus.

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 26.05.2014