Investigating the Influence of Different Counting Conditions in Count Matrix Based Code Clone Detection

Lasse Schuirmann, supervised by Prof. Dr. Sibylle Schupp

Institute for Software Systems, August 3, 2015

Abstract

Code clones are well known to be an indicator of architectural issues in a code base. Finding them can help to improve code quality substantially. A novel clone detection technique has been proposed in 2011, which is called the Count Matrix based Clone Detection (CMCD). This technique is able to detect a larger range of code clones than its predecessors. The goal of this thesis is to develop and implement an algorithm on the basis of CMCD, and evaluate the precision of the algorithm and the modifications compared to the original implementation. The final goal is to evaluate if the resulting algorithm is suitable not only for investigating potential code clones in a real code base but for actual day to day production use.

For a production usable clone detector, false positives can be considered even more important than undetected clones. They can also be used for implementing a supervised learning process. By extending the CMCD algorithm and combining traditional scenario based evaluation with evolutionary approaches, it was possible to create a code clone detection algorithm that is able to adapt its understanding of a code clone to a given test dataset. However, a large test dataset is needed in order to be able to use the algorithm to generate a configuration that results in production quality false positive rates in the field. Analysis has shown that the proposed implementation for C is not yet able to deal well with missing include paths. The false positive rate of our generated configuration is unacceptably high. When lowering the threshold manually, the clone detection is still able to identify a great deal of code clones while having a neglectable false positive rate if needed include paths are known. The evaluation of the main modifications to the algorithm using the test dataset has shown that they are beneficial for the precision of the algorithm.

To avoid the stated weaknesses, it is recommended to implement the CMCD algorithm using a lexical approach. Furthermore, an extensive study about the creation of a sufficiently big test dataset consisting out of clone and non clone samples to evaluate arbitrary code clone detection algorithms is needed.

All code written is fully tested, known to work on all major OSes, maintained openly by the coala community, and provides a good basis for future research through its generality and modularity.

Acknowledgements

I hereby want to thank Viviana Gosch, Konrad Kohbrok, Abdeali Kothari, Mischa Krüger, Fabian Neuschmidt, Prof. Sibylle Schupp, Max Schürenberg, Marc Siemering, Udayan Tandon and Marie Wegner for taking the time to review this thesis and providing helpful feedback.

Special thanks go to my supervisor Prof. Sibylle Schupp for supporting and supervising my thesis while helping me to raise the quality through constructive feedback during the whole process.

Especially important for the success of this project was the coala community which helped me through code reviews and proofreading. The help from Abdeali Kothari, Mischa Krüger, Fabian Neuschmidt and Udayan Tandon was and is invaluable for the creation and progress of the coala project which provided a great basis for this work. Their constructive feedback helped me to keep the quality levels up for this code clone detection algorithm.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code clones are usually created when developers either copy and adapt code from various other sources or unknowingly rewrite code already present in the same software product. Clones obviously yield more code than needed; combining two clone fragments into one, more generic, fragment helps to reduce code and simplifies bugfixing, as a developer might not need to fix the same bug in several places. Finally, code clones can serve as an indicator that a code fragment is not written generically enough. All in all, duplicate code leads to more code, less generality, and more bugs.

Production quality tools are already existent that can detect some code clones. However, those are usually limited to finding simple clone types (type 1 and 2, see Scenario Based Evaluation). When Roy et al. evaluated a wide range of code clone detection tools in 2009, none of them was able to detect all the clones presented to them [4].

In 2011 a novel code clone detection technique, the Count Matrix based Clone Detection (CMCD), was proposed. It was shown by different authors that this technique is able to detect all samples given by Roy et al. [1,5]. The CMCD technique, which is outlined in detail in the next chapter, uses variable counts to determine similarities. So called Counting Conditions (CCs) count each variable under different circumstances. Those variable counts are used to compare all possible pairs of functions and determine a difference value for all of them. A major weakness of this technique is the false positive rate when searching for duplicate functions[1]. Furthermore, the understanding of code clones is subjective and the CMCD algorithm in its current form is not easily configurable to match a new understanding of code clones.

The major aim of this thesis is to determine if a production quality code clone detection algorithm that is adaptable to the user's understanding of a code

[1] Yuan and Guo do not state any false positive rate for function duplicates. The authors of a variant state a rate of 15%.

clone can be developed using the Count Matrix based Clone Detection (CMCD) technique. Important requirements for such an algorithm are a low false positive rate while keeping the ability of detecting a large range of clones, including syntactically different code fragments that perform a similar computation.

To achieve the stated goal, the introduction of weightings for the CCs, a novel normalization method, and postprocessing for the difference value are proposed. In addition, evaluation and tweaking algorithms are introduced to automatically determine suitable thresholds and CC weightings. To finally determine if a production quality code clone detection using the CMCD technique can be developed, the following steps were followed:

1. Implement the CMCD technique with the proposed modifications as well as the proposed evaluation and tweaking algorithms. The modifications can be turned off individually.
2. Set up a basic test dataset using the clone samples by Roy et al.
3. Use the proposed tweaking algorithms to retrieve a good configuration.
4. Apply the clone detection algorithm to a code base.
5. Reproduce false and true positives in the test dataset.

Running the code clone detection on real code has helped to identify gaps in the test dataset and extending it iteratively.

The test dataset was used to evaluate the algorithm and each modification done to it. Results indicate that the proposed modifications are beneficial for the quality of the clone detection. Furthermore, the algorithm was evaluated by determining false positive rates on real life code bases. We have seen high false positive rates with the configuration generated by our algorithms and concluded that a larger test dataset is needed for better results. We also determined that the algorithm is not able to deal with unknown include paths because the implementation uses a full abstract syntax tree for analysis. By lowering the similarity threshold for clones manually, we were able to improve the results drastically. The algorithm was still able to detect many clones of all types while no false positives were raised at the tested projects with no unknown include paths.

We especially recommend further research on test datasets for code clone detection software which should cover a large base of true as well as false positive samples. Furthermore, we see the adaption of the CMCD technique to be able to detect clones at sub function level as the next evolutionary step.

Code samples given in this thesis follow the python syntax.

Chapter 2

Fundamentals

This chapter introduces the most important terms, the scenario based evaluation method, as well as the original Count Matrix based Clone Detection (CMCD) algorithm and a variant. The approaches presented here are combined, implemented, and evaluated in this thesis.

2.1 Scenario Based Evaluation

In its most generic form, a code fragment is considered a clone of another code fragment if both are similar by a definition of similarity. This generic definition leads to the conclusion that a tangible definition of similarity is needed, to write a proper code clone detection. However, as no such objective similarity definition exists for this purpose, it is feasible to create an abstraction over the code for which similarity can be measured. This model can be evaluated through applying it to code samples. Such an evaluation is called Scenario Based Evaluation [4].

Code clone samples can be classified into several types: type 1 clones have only comment and whitespace variations, type 2 clones feature additional identifier variations while still being syntactically equal. Type 3 clones are copied fragments with further modifications (changed or deleted statements) and type 4 clones are arbitrarily different code fragments that perform the same computation. Each of those types represents a different kind of challenge for a code clone detection algorithm. Samples representing different scenarios[1] for all of those types exist [4] and are used among others to evaluate the algorithm developed in this thesis. None of the code clone detection techniques evaluated at that time was able to detect all samples as clones.

[1] See files matching `s[1-4][a-e].c` on https://github.com/coala-analyzer/coala/tree/master/bears/tests/codeclone_detection/clone_detection_samples/clones

Roy et al. did not do research about non-clone samples and thus provides no full dataset for the evaluation of a code clone detection algorithm: obviously an algorithm that classifies every arbitrary function pair as a clone passes this scenario based evaluation because their evaluation had no restrictions on false positives.

2.2 Count Matrix Based Code Clone Detection

The Count Matrix based Clone Detection (CMCD) technique was chosen because its authors have shown that it can evaluate all given scenarios mentioned in the previous section. This section describes the CMCD approach as it was originally proposed [5].

The CMCD technique creates Count Vectors for each variable in each code fragment. Those vectors provide an idealized abstraction over the code - nonessential information for clone detection is consciously left out. A Count Matrix represents one code fragment and is made up of a set of Count Vectors. Based on those matrices, a comparison can be performed to generate a difference or similarity value. This value can be compared to a threshold to sort the fragment pairs into clones and non-clones.

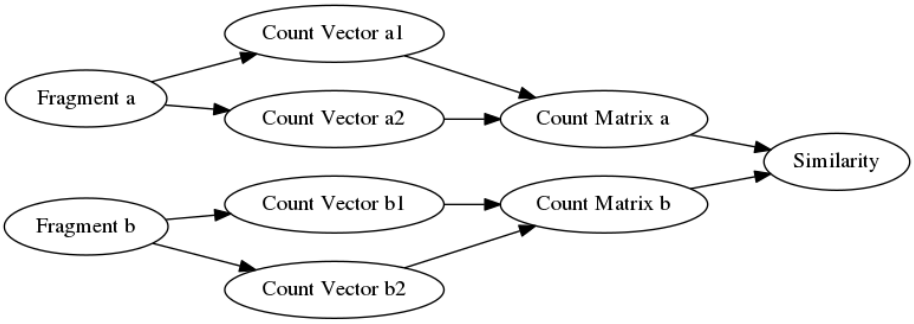

Figure 2.1: Visualization of CMCD Data Transformations

2.2.1 Creating Count Vectors and Matrices

Each variable within a code fragment is represented by the count of further specified usages. As usages are only counted, the order of usages and the name of the variable is not reflected by this metric - this inherently makes the algorithm suitable to detect all type 1 and type 2 clones as they will yield exactly the same vectors. Because of this, similarity of two Count Vectors (CVs) can indicate that the associated variables are used in similar ways.

The usage counts held in each CV are determined by so called Counting Conditions (CCs). All CCs are evaluated for each occurrence of each variable and have their own position in the CV. For example, the code `i++; return i;` with the CCs (Used, Returned) results in a CV of $(2, 1)$ for the variable `i`.

The table below lists the CCs used in the original proposal. They are targeted at Jimple, an intermediate representation for Java.

Table 2.1: Original Counting Conditions

CC ID	Short Description
1	Used
2	Added or subtracted
3	Multiplied or divided
4	Invoked as a parameter
5	In an if statement
6	As an array subscript
7	Defined
8	Defined by add or subtract operation
9	Defined by multiply or divide operation
10	Defined by an expression which has constants in it
11	In a third-level loop (or deeper)
12	In a second level loop
13	In a first-level loop

While the creation of the vectors obviously requires some understanding of the target language, the further steps of the algorithm are language-independent.

The CMCD technique creates a so called Count Matrix (CM) for each code fragment, which is a set of CVs. The CMs are created for functions only; other possible code fragments are not taken into account. It is a limitation of this algorithm, that only duplicate functions are processed.

It is worth to note that many small type 4 clones exist in many code bases (e.g. getters and setters). Because identifying those as clones is not useful for improving a software, functions of that size are simply excluded from the code clone detection.

2.2.2 Comparing Count Matrices

CVs are compared using a normalized euclidean distance. The euclidean distance is normalized by a heuristic function that produces a value in the interval $[0, 1]$, which expresses the similarity. 0 means variables are not similar, 1 means their CVs are equal.

While the authors did not disclose the normalization function [5], it is worthwhile to note that it performs a special treatment for several corner cases like both vectors being "small" or differing in many dimensions of the CV. Furthermore, the function creates a similarity value out of the normalized difference.

As each function is now represented by the CVs of its variables, a matching needs to be done. Each variable from one function needs to be matched with one from the other function. If one function has fewer CVs than the other, those are filled up with zeroed CVs. A matrix is created which holds the difference for all possible variable pairs in each cell to do this matching.

With the cost matrix created, the Kuhn–Munkres (KM) algorithm[2] can be used to determine the perfect matching (a maximum matching with the highest possible similarity, i.e. the lowest cost) in polynomial time [5]. The sum of the matches is normalized by a heuristic function, taking the method and variable sizes into account. This value is compared to a threshold to sort function pairs into clone and non-clone pairs.

This procedure can also be applied to classes and packages to retrieve higher level duplicates which is useful for plagiarism detection and not a subject of this thesis.

2.3 CMCD Variant

A variant of the CMCD algorithm exists, which was written in order to validate the results of the initial proposal further [1]. In this work several changes are proposed:

This algorithm builds upon using difference instead of similarity values, which simplifies the model.

It parses Java directly and does not use an intermediate representation like Jimple. The authors argue that minor changes in the original source code might alter the intermediate representation considerably because of optimizations.

A set of new counting conditions, which are listed in table 2.2, were introduced to enhance the clone detection results. Especially important is the addition of counting conditions that count function calls (as those are ignored in the original implementation), the handling of the `switch-case` statement as well as the distinction of `for` and `while` loops. The loop distinction was not done in the original implementation because Jimple does not have the distinction of `for` and `while` loops.

[2] More information and a list of implementations for the KM algorithm can be retrieved here: https://en.wikipedia.org/wiki/Hungarian_algorithm.

Table 2.2: Additional Counting Conditions

CC ID	Short Description
11	In a first-level `while` loop
12	In a second level `while` loop
13	In a third-level `while` loop (or deeper)
14	In a first-level `for` loop
15	In a second-level `for` loop
16	In a third-level `for` loop (or deeper)
17	In a `switch-case` statement
18	Method invoked
19	Method used in `if` statement
20	Variable invoked on

This algorithm uses its own heuristic that excludes function duplicates where the textual difference (excluding comments and whitespaces) is over 50%. In addition, Chen et al. propose to exclude constructor methods because they are usually false positives.

2.4 Evaluation

The authors of the original technique evaluated their algorithm against the scenarios mentioned in Scenario Based Evaluation. They show that their algorithm can successfully detect all of them. This result was confirmed with the variant explained in the paragraphs above.

As the task of a clone detection algorithm is to sort code fragment pairs into clones and non-clones, the false positive rate cannot be ignored for a proper evaluation.

The original proposal states that the false positive rate, when searching for cloned classes to check for plagiarism, was zero. A false positive rate for the detection of cloned functions is not given. The authors of the derivative implementation state a false positive rate of 15%.

All in all, the CMCD technique is a very promising approach to find code clones. We see improvement possibilities especially with respect to the false positive rate and the configurability of the algorithm.

Chapter 3

Design

This chapter outlines the design of a new algorithm on the basis of CMCD as well as differences in comparison to the original algorithm.

Although there are other applications for code clone detection, searching refactoring opportunities to improve code quality is the focus of this thesis. This algorithm aims at detecting function pairs that provide such an opportunity and the sample pairs are classified accordingly. The false positive rates for plagiarism check and detecting function pairs are not comparable because the aim of a plagiarism check is not to detect all type 4 clones by definition.

The false positive rate is of high importance for productive usage; if many results yielded by such an algorithm are false positives, it can hardly be useful for productive work. Furthermore, the understanding of a useful result (recognized code clone) might differ from one user to another. Making the algorithm adaptive to the user's needs is another major goal. To meet those goals, this algorithm features some novel additions that make it more configurable. Also, a set of algorithms is proposed that help to evaluate and improve a code clone detection algorithm on the basis of a test dataset.

The main algorithm follows the same process as the CMCD does. It can be roughly partitioned into three steps: first, a dictionary is created that associates all functions and their location with their count matrix (Count Matrix Creation). Second, a difference value is generated for each pair of functions (Difference Calculation). Lastly, this difference value is compared with a threshold and the user is informed about the result (Clone Pair Determination). The following sections describe each of those steps.

3.1 Count Matrix Creation

This algorithm uses the same counting principle as the original technique. The choice of the CCs is vital for the precision of the algorithm. Note that the algorithm is designed such that it can easily use only a subset of the given CCs.

The CCs are designed with the C programming language in mind. Important additions to preceding CCs are the conditions filtering usages as a condition (`Is condition`), which is distinct from a usage inside a body of a condition (`In condition`). Where possible, generalizations are done to enhance the detection of type 4 clones. For example, `Is condition` matches not only variables used as a condition for an `if` statement but also ones that are used for conditions in loops; `for` and `while` loop bodies are treated the same as they can be easily transformed into the other form. As `switch-case` statements are semantically nothing but a sequence of `if` statements, they can be treated as such and have no own CC. Such distinctions are consciously removed at the CV creation step.

Table 3.1: New Counting Conditions

CC ID	Short Description
1	Used
2	Returned
3	Is condition
4	In first level condition
5	In second level condition
6	In third level condition (or deeper)
7	Is assignee
8	Is assigner
9	In first level loop
10	In second level loop
11	In third level loop (or deeper)
12	Is parameter
13	Is called
14	Is call parameter
15	Is added or subtracted
16	Is multiplied or divided or moduloed
17	Is used in binary operation
18	Member accessed

Contrary to previous implementations, all identifiers as well as constants are treated the same. Thus, all occurrences of variables, functions, and constants will be evaluated by the CCs given above. This allows function pointers to be represented appropriately as they may be used as a variable and as a function in the same scope. It also implies that if the constant `1` is used two times the appropriate CV entry will hold 2.

Empty and very small CMs can be excluded without loss of precision. Functions consisting only of very few or no lines of code (the latter being e.g. declarations) rarely represent a refactoring opportunity. In addition, functions consisting only out of many constants but doing few nonstatic calculations are also unlikely to be useful results. Because of this, only CVs for variables and function invocations are considered for function exclusion. Only CMs with more than two CVs and at least one CV $\mathbf{a}$ while $||\mathbf{a}||_1 \geq b$ with $b \in \mathbb{N}^+$ are considered for further processing. Furthermore, all CMs where the sum of the counts of all CVs is smaller than c are ignored. b and c can simply be set to the highest possible numbers that does not exclude any known clone in a given test dataset.

As some CCs might be more important than others for the comparison, weightings can be used to raise or lower the influence of single CCs over others. All counts in all CVs are multiplied by the weighting for the associated CC, raising or lowering its influence on the end result. Note that this weighting is applied after exclusion of small functions: it is only useful for tweaking the function comparison and would make the function exclusion unnecessarily complex while providing no advantages[1].

3.2 Difference Calculation

As the metric is a vector structure, a trivial way to calculate a difference is using the euclidean distance for those vectors. To create a difference value of two functions, some normalization needs to happen. This can happen either for variable pairs or for function pairs. Both of those approaches are outlined below in detail.

With the difference values in place, a cost matrix can be created holding the difference for each pair of CVs in each cell. The KM algorithm is used to determine the best possible combinations of CVs in order to minimize the cost (i.e. the sum of the differences). The resulting difference value can optionally be postprocessed as described further in Postprocessing Function Differences.

3.2.1 Normalized Variable Comparison

The difference value for each variable comparison can be normalized using an approach that is similar to what was used in previous work [1,5]. This eases the further steps for the function comparison, as it results in a relative CV_diff value within the interval $[0, 1]$.

[1] A big advantage of applying the exclusion before applying weightings to the CV is that the exclusion behaves stable and consistent even when the weightings change. Few manual experiments were not able to reveal any advantages of heuristics using weighted values for function exclusion.

Normalizing with the maximum over the norm of both vectors obviously may yield a result bigger than 1 and is thus not a valid option. This difference function instead normalizes the euclidean distance using the 2 norm of a vector that holds the maximum value of both vectors in each row:

$$CV\_diff(\mathbf{a}, \mathbf{b}) = \frac{||\mathbf{a} - \mathbf{b}||}{\sqrt{\sum_{i=1}^{dim(\mathbf{a})} max(\mathbf{a}_i, \mathbf{b}_i)^2}}$$

After applying the KM matching algorithm a simple average calculation returns a value in the target interval $[0, 1]$.

The following formula depicts the whole function comparison, while **matches** allows iterating over all pairs of matching variables determined by the KM algorithm:

$$CM\_diff_1(\mathbf{matches}) = \sum_{\mathbf{a},\mathbf{b}=\mathbf{matches}} \frac{||\mathbf{a} - \mathbf{b}||}{\sqrt{\sum_{i=1}^{dim(\mathbf{a})} max(\mathbf{a}_i, \mathbf{b}_i)^2} \cdot dim(\mathbf{matches})}$$

3.2.2 Normalized Function Comparison

The approach outlined above has the disadvantage of weighting all variables the same, independent of the size of their count vectors. For example, the difference value of the CMs $((0), (50))$ and $((1), (50))$ is 0.5. The variable normalization strips off the size information for each variable pair during the comparison. The novel approach presented below tries to conquer this symptom by first summing up all matching differences and then normalizing the whole function difference by the sum of the maximum vector norms. Note that to get the matches, the normalized differences (CV_diff) are calculated to model the impact of each variable best before performing the matching.

The whole function comparison can be mirrored to the following formula, while **matches** allows iterating over all pairs of matched variables:

$$CM\_diff_2(\mathbf{matches}) = \frac{\sum_{\mathbf{a},\mathbf{b}=\mathbf{matches}} ||\mathbf{a} - \mathbf{b}||}{\sum_{\mathbf{a},\mathbf{b}=\mathbf{matches}} \sqrt{\sum_{i=1}^{dim(\mathbf{a})} max(\mathbf{a}_i, \mathbf{b}_i)^2}}$$

3.2.3 Postprocessing Function Differences

Arguably, if two function pairs have the same difference value while one is a pair of two "big" functions and the other of two "small" functions, the former pair is more likely a refactoring opportunity for the user than the latter. To reflect this property of the reality in the algorithm, the function difference value can be multiplied by a factor $w \in [0, 1]$ that depends on the size of the functions.

The following term can be used to represent the size of a function pair; it is the normalization value used in the Normalized Function Comparison and takes the numbers of matches as well as the sizes of the variables in both fragments into account.

$$s(\mathbf{matches}) = \sum_{\mathbf{a},\mathbf{b}=\mathbf{matches}} \sqrt{\sum_{i=1}^{dim(\mathbf{a})} max(\mathbf{a}_i, \mathbf{b}_i)^2}$$

A function $w(s)$ that yields such a factor for a given function size s should have certain properties:

1. The function maps a value in $[1, \infty)$ to a value in $[0, 1]$.
2. $f(1) = 1$ and f should be monotonically decreasing. Differences of smaller functions are higher than differences of bigger functions with the same raw differences.
3. $\lim_{s\to\infty} f(s) = b$ with $b \in (0, 1)$. For example with $b = 0.75$ big functions are allowed to have an approximately 33% higher difference value than small ones to be detected as clones ($1.33 * 0.75 \approx 1$).

It is straightforward to model such a function. The functions given here represent two such approaches for $b = 0.75$:

$$w_{exp}(s) = \frac{e^{-(s-1)}}{4} + 0.75$$

$$w_{pol}(s) = \frac{3s + 1}{4s}$$

These approaches are evaluated and compared in the Results and Discussion chapter.

3.3 Threshold Determination

The calculated difference value needs to be compared to a threshold value to finally separate clones and non-clones. This value is dependent on various choices:

- The weightings of the CCs.
- Used heuristics for function exclusion.
- The choice of the normalization method.
- Any postprocessing of the difference values.

The threshold can easily be determined for a test dataset using an algorithm, assuming it is known for each function pair in the test dataset if it is a clone or not:

1. Parse all functions and create their CMs.
2. For each function pair that is known to be a clone: put difference in **clone_differences**.
3. For each function pair that is known to not be a clone: put difference in **non_clone_differences**.

The following must now hold for the threshold t:

$$max(\textbf{clones\_differences}) < t < min(\textbf{non\_clone\_differences})$$

If no such t exists, the algorithm does not fit the given test dataset with the configuration used, i.e. false positives or unrecognised true positives will occur. However, if it is possible to match this condition, the algorithm is able to evaluate the test dataset correctly. The existence of such a t that matches the given condition does not necessarily imply that the algorithm has a low false positive rate when running on real life code although this is a desired property.

3.4 CC Weighting Determination

With a decent number of CCs, there are many possibilities to weight them. Those cannot be evaluated easily. However, it is practical to determine the best suitable weightings automatically dependent on a test dataset.

To determine the "best suitable weightings", a fitness function f is needed that determines how good a configuration of the algorithm is. Such a function maps a vector **weightings** to a real number and can be easily derived from the algorithm stated in the previous section:

$$f(\textbf{weightings}) = min(\textbf{non\_clone\_differences}) - max(\textbf{clones\_differences})$$

Indeed, other approaches for this fitness function, taking not only the minimum non-clone difference and the maximum clone difference into account, are viable

and probably superior, especially for a negative fitness. However, the function depicted here has proven to be sufficient to generate a positive fitness value for the test dataset provided to it.

This ideal maximum of the fitness function can not be calculated easily. However, the function can be probed at arbitrary points. Probing an n-dimensional grid is not practical as the number of possibilities grows too fast.

Instead, it is feasible to write an algorithm that approaches a local maximum iteratively. Such an algorithm could start at an arbitrary point, probe its neighbours and repeat this procedure with the fittest neighbour point.

Such an approach is likely to only find a local maximum. To avoid this, a new algorithm was developed, which is depicted below.

```
old_weightings = [0 for cc in counting_conditions]
weightings = [1 for cc in counting_conditions]
while old_weightings == weightings:
    old_weightings = weightings.copy()
    for i in range(weightings):
        best_weighting = weightings[i]
        best_fitness = f(weightings)
        for weighting in frange(0, 2, step=0.2):
            fitness = f(weightings[:i]+[weighting]+weightings[i+1:])
            if fitness > best_fitness:
                best_weighting = weighting
                best_fitness = fitness

        weightings[i] = best_weighting
```

This algorithm is inspired by the idea of evolutionary algorithms. For each dimension of the **weightings** vector it creates a population spreading across a predefined interval. As the difference values and the weightings are relative, the actual interval does not matter; the step size can be used to trade off precision and performance. The algorithm selects the fittest member of the population and continues the search. This procedure is applied repeatedly until the **weightings** vector does not change anymore, i.e. a local maximum is reached.

Using this approach is less likely to only find a local maximum because the whole interval gets probed for each dimension.

3.5 Summary of Changes

All in all a number of changes were proposed to the original algorithm: A new set of CCs is used and all identifiers (variables, called functions) and constants are counted in the Count Matrix Creation step. Furthermore, a weighting

mechanism was proposed to be able to fine-tune the CCs. For the Difference Calculation step, two function normalization approaches were proposed. The new normalization takes the sizes of the matched variables into account. Additionally, a postprocessing step was introduced that allows artificially lowering the difference of big functions.

In addition, two algorithms were proposed that are able to determine good CC weightings and a suitable threshold for the detection algorithm based on a test dataset.

Chapter 4

Implementation

This chapter outlines the specifics concerning the implementation of the algorithms designed above. Each of the steps described in the previous chapter (Count Matrix Creation, Difference Calculation, Clone Pair Determination) is implemented in a module on its own. This modularization makes it easy to write evaluation tools like the described threshold determination algorithm. It also ensures that the language-independent parts of the algorithm can be reused without any Clang dependencies.

The tool chain and code were chosen and written for maximum precision of the results, usability, maintainability, and readability. Performance was only a secondary aspect.

Due to time limitations the `switch-case` is unhandled in the implementation. See https://github.com/coala-analyzer/coala/issues/652 for the current status of this task. All other aspects depicted in the Design Chapter are fully implemented.

4.1 Tool chain

The tools described in table 4.1 have been used to simplify the implementation of the CMCD algorithm.

Table 4.1: Tools Used for Implementation

Name	Short Description
coala[1]	A framework for simplifying code analysis
Clang[2]	A C language family frontend for LLVM
Python 3[3]	A programming language
Munkres3[4]	Kuhn-Munkres algorithm in Python 3

The coala framework is used to provide user interaction and simplify modularization of the problem. With the coala framework in place, this clone detection algorithm can not only be used for experiments but also easily be integrated and used for production code. It allows writing any analysis routine (so called Bear) as a simple function, makes the Bear easily configurable and presents results to the user in his preferred form. coala also takes care of parallelization and dependency resolving so that tasks can easily be split up in multiple steps and raw results can be reused by multiple other Bears. The CMCD implementation was integrated into the coala codebase; this made it easy to reuse coala's testing and build automation infrastructure. It also implies that the maintenance and further development is continued under the umbrella of the coala community.

As coala is implemented in Python 3 and provides a very good integration, it was the language of choice for this implementation.

Clang was used to parse C code into an Abstract Syntax Tree (AST). This simplifies the creation of CMs and CCs as it reduces the CV creation to only inspecting and iterating over the AST. It is worth to note that the Clang python bindings are only provided for Python 2 and needed adaption. Furthermore, Clang does not represent macros in the AST but resolves them. This may lead to unexpected behaviour when dealing with complex macros[5]. However, as CCs still behave consistently this is not considered to be a big issue with this implementation. This lies in agreement with the argumentations of previous work [4].

The Munkres3 PyPI module provides a readily implemented KM algorithm to solve the matching problem in polynomial time.

[1] https://github.com/coala-analyzer/coala
[2] http://clang.llvm.org/
[3] https://www.python.org/
[4] https://pypi.python.org/pypi/munkres3/1.0.5.5
[5] Clang does resolve the macros when creating the AST. However, as the python bindings are far from complete, some CCs rely on tokens to understand the code. This rather "hacky" way could not be avoided due to time limitations. In the tokenized representation, macros are still unresolved and thus may lead to unexpected behaviour.

4.2 Data Structures and Function Clusters

This section outlines the most important data structures and functions implemented during the work on this thesis.

The `CountVector` object is completely independent of the target language and consists of the weighted and unweighted count values as well as the CCs. Each CC is a function object with an arbitrary signature. The `CountVector` object provides means to count a variable occurrence. All provided parameters will be forwarded to each CC. This way, the caller can provide different information to the CCs dependent on the target language. Furthermore, the `CountVector` provides routines to calculate difference and normalization values.

The `ClangCountVectorCreator` uses the Clang-specific counting conditions to create CMs which are simply a dictionary associating each identifier name[6] with `CountVector`s. It creates CMs for all functions in a given file.

The `CloneDetectionRoutines` module holds several utilities to simplify the CMCD - all of them are target-language independent. It holds methods to simplify creation of count matrices for a set of files (using a language specific `CountVectorCreator` object), takes care of small function exclusion, and performs the function comparison including postprocessing.

4.3 Modularization

This section outlines the top-level structure of the implementation. Objects described here use the function and data structures described in the previous paragraphs.

The `ClangFunctionDifferenceBear` parses all files and calculates the difference values. Dependent on those values, the `ClangCloneDetectionBear` shows the results to the user, determined by a manually specified threshold. The `ClangCloneDetectionBenchmarkBear` also uses the results of the `ClangFunctionDifferenceBear` to generate a suitable threshold range.

The `ClangCCOptimizeBear` optimizes the CC weightings automatically as described in CC Weighting Determination. Because it overrides the user's weighting settings and runs the difference calculation multiple times, it uses the `ClangCountVectorCreator` and `CloneDetectionRoutines` directly.

The diagram below depicts the described top-level structure.

The code[7] as well as the tests[8] are publicly available in the official coala repository.

[6] Identifier names are variable names and function names. Constants are mangled with a preceding "#".

[7] https://github.com/coala-analyzer/coala/tree/master/bears/codeclone_detection

[8] https://github.com/coala-analyzer/coala/tree/master/bears/tests/codeclone_detection

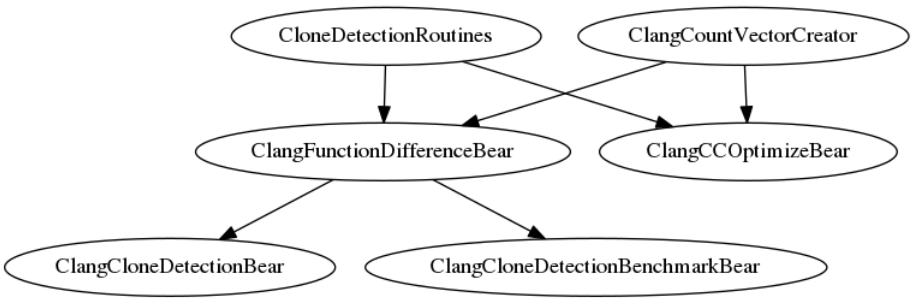

Figure 4.1: Top-Level Implementation Structure

Code for evaluation and tweaking resides in an extra repository[9].

4.4 Include Path Guessing

Clang creates a full AST out of the given code. Knowing possible include paths is thus highly relevant for the precision of the CVs, as Clang simply ignores e.g. variables with unresolvable types[10]. As explained before, one of the requirements for this algorithm is to deal with insufficient information, that means it shall be able to detect suitable include paths by itself.

To deal with this problem, all subdirectories of the project directory can be used as possible include paths. The location of the configuration file can be used as the "project directory". This approach obviously is able to resolve all project internal dependencies. Manual testing has revealed that the performance penalty is neglectable compared to even simpler include path guessing methods.

[9] https://github.com/coala-analyzer/clone-evaluation-helpers
[10] Only the declaration of a variable that has an unknown type will appear in the AST.

Chapter 5

Results

This chapter lists results and evaluation data gathered with the algorithms described above. The results consist of the test dataset, an evaluation of each modification done to the original CMCD algorithm and determination of the optimal configuration. For those evaluations, the test dataset is used in conjunction with the fitness function presented earlier. In the end, the optimal configuration is evaluated against real software projects, evaluating each result manually.

Because all presented algorithms are deterministic, all results can be reproduced using revision `6bf58925` of the coala project and `b0e12834` of the clone-evaluation-helpers project. The table given at Tested Projects in the appendix allows the exact identification of the code the detection was tested on and provides commit/revision IDs to keep the identification stable over time.

5.1 Test Data Set

On the basis of the sample clones [4] a test dataset was created. Some of those samples are copied from freely available sources, the exact sources are mentioned in the source code. The test dataset is contained in the coala project in `bears/tests/codeclone_detection/clone_detection_samples/`.

The clone dataset contains the sample clones by Roy et al. and the following other samples:

- Two factorial implementations, one using a `while`, the other using a `for` loop.
- Two different implementations of the bubblesort algorithm from distinct sources on the internet.

- Other real life functions that were already discovered by the clone detection algorithm.

The non-clone samples were created from scratch. Essentially they cover six different sort algorithms from various sources and other functions that were assumed to be problematic or recognized as false positives during explorative testing.

The test dataset contains 22 clone samples and numerous non-clone samples. It is not possible to compare the number of clone and non-clone samples because non-clone samples are not strictly organized in pairs: e.g. for sorting algorithms it makes sense to combine all six functions into one file and request that no combination is recognized as clone. The non-clone samples consist out of 27 unique functions.

5.2 Algorithm Modifications

The table given below depicts the fitness of the algorithm using the different modifications proposed below. For each combination of modifications, it lists the "Unoptimized" as well as the "Optimized" fitness: the unoptimized value is the achieved fitness when using all counting conditions with the same weight. The optimized value is the achieved fitness after applying the CC weighting determination algorithm in its default configuration (weightings from 0 to 2, step width 0.2) without a time limitation. All CC weightings were set to 1 as start value and CCs are ordered as indicated by their IDs in this document.

Optimizing the CC weightings for one configuration took between 10 and about 50 minutes on an average consumer processor[1]. The actual time depends much on the number of iterations needed to get a stable fitness.

Table 5.1: Fitness of Different Modifications

Normalization Type	Postprocessing	Unoptimized	Optimized
Average	Off	−0.3628	−0.198
Average	Polynomial	−0.2712	−0.173
Average	Exponential	−0.2721	−0.149
Average	Both	−0.2034	−0.130
Function Wise	Off	−0.1512	0.069
Function Wise	Polynomial	−0.1127	0.087
Function Wise	Exponential	−0.1134	0.052
Function Wise	Both	−0.0845	0.065

[1] A paralellelized version of the algorithm was used on an Intel(R) Core(TM) i5-4570 CPU with a base frequency of 3.2 GHz and a maximum frequency of 3.6 GHz.

Because one test case was specifically designed to exploit the weakness of the average calculation normalization, the evaluation of the normalization type was repeated with no postprocessing. The following table denotes the unoptimized fitness only because this test is not used for further processing:

Table 5.2: Fitness of Different Normalization Types (Normalization Exploiting Test Excluded)

Normalization Type	Fitness
Average	−0.2541
Function Wise	−0.1512

The configuration with the highest fitness (0.087) is depicted below.

```
average_calculation = false
poly_postprocessing = true
exp_postprocessing = false
```

Table 5.3: Fittest Counting Condition Weightings

CC ID	Short Description	Weighting
1	Used	0.0
2	Returned	1.4
3	Is condition	0.0
4	In first level condition	1.4
5	In second level condition	1.4
6	In third level condition (or deeper)	1.0
7	Is assignee	0.0
8	Is assigner	0.6
9	In first level loop	0.0
10	In second level loop	1.0
11	In third level loop (or deeper)	1.0
12	Is parameter	2.0
13	Is called	1.4
14	Is call parameter	0.0
15	Is added or subtracted	2.0
16	Is multiplied or divided or moduloed	0.0
17	Is used in binary operation	1.0
18	Member accessed	1.0

A lower boundary for the threshold is 0.308. This is the lowest lower boundary within the precision of three decimal places, determined using the algorithm

described in Threshold Determination. The clone with the highest difference value consists of `clearScreen()` and `scrollUp()`. The non clone with the lowest difference value consists of `array_editorconfig_name_value_clear( array_editorconfig_name_value *)` and `reset_special_property_name_ value_pointers(array_editorconfig_name_value *)`.

5.3 Real Code Tests

The code clone detection was evaluated against different projects. For the evaluation, the configuration with the highest fitness given in the previous section was used. coala was run from the project root, even if only a subdirectory was evaluated[2]. The `coala-json` binary was used to export the results to JSON and ease the evaluation of the large amounts of results.

The tables below associate the project name with the number of false and true positives. In some columns, two values separated by a forward slash are given. In those cases, the first value relates to the number of function pairs, the second to the number of unique functions. E.g. if 95 legitimate clone pairs consisting of 57 unique functions were found, the "True" value is "95/57".

The "False Positive Rate" is calculated by dividing the number of false positives through the sum of true and false positives.

Any function which was detected as a duplicate which was judged legitimate, was counted as a "True" Positive. Any function that was detected as an illegitimate duplicate was counted as a "False" Positive. A pair of functions was considered a legitimate code clone if it hints clearly to an opportunity to modularize code in a more generic or efficient way or the fact that it was detected as a clone is directly related to one of the functions having a bug. The former case implies that at least parts of both functions exist that can be written as an own function. Signatures of all evaluated function pairs are given in the Tested Projects.

The "NCLOC" column lists the number of non-comment non-blank lines of C code. Measured using the cloc utility.

Table 5.4: False Positive Rates With Real Code: Threshold 0.308

Project	Directory	False	True	False Positive Rate	NCLOC
Yafos	.	0/0	3/6	0%/0%	1649
TE3D	.	1/2	95/57	1%/3%	3114
CPython	`Parser`	8/13	11/16	42%/45%	4169
Linux	`fs/ext2`	4/8	3/6	57%/57%	6546
Builder	.	35/28	6/12	85%/70%	91848

[2]This is an important fact because the location of the project file is used for include path guessing.

The figure below depicts how true and false positive pairs as well as the false positive pair rate relate to threshold changes.

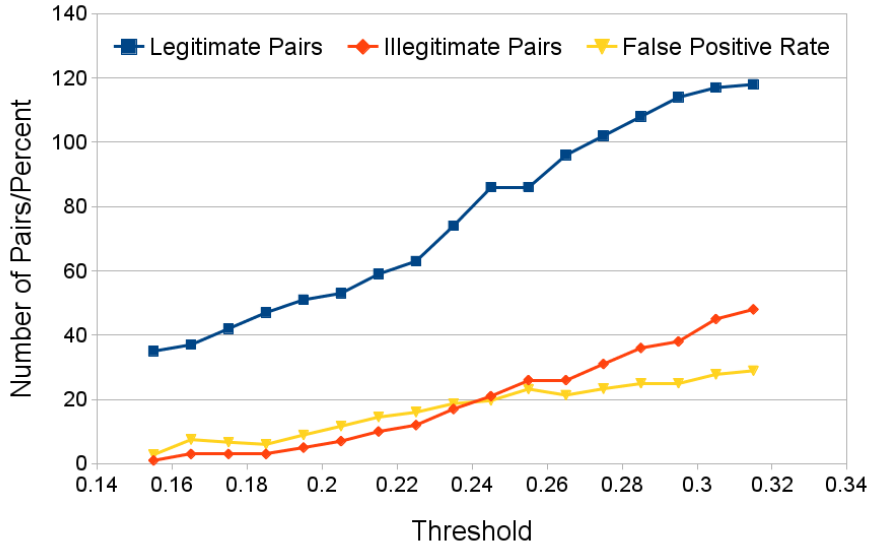

Figure 5.1: Influence of the Threshold to the False Positive Rate

The following table shows the result when lowering the threshold to 0.185. The "Missed" column shows how many clone pairs and unique clone functions are undetected compared to the original threshold of 0.308.

Table 5.5: False Positive Rates With Real Code: Threshold 0.185

Project	Directory	False	True	False Positive Rate	Missed	NCLOC
Yafos	.	0/0	1/2	0%/0%	2/4	1649
TE3D	.	0/0	34/47	0%/0%	60/10	3114
CPython	`Parser`	0/0	4/8	0%/0%	7/8	4169
Linux	`fs/ext2`	0/0	3/6	0%/0%	0/0	6546
Builder	.	3/6	5/10	38%/38%	1/2	91848

Of the samples given by Roy et al. only the samples s3c and s3e were not recognized using a threshold value of 0.185.

Chapter 6

Discussion

In the previous chapter it was discovered that the CC weighting determination algorithm is able to raise the fitness significantly. Albeit, it is not able to raise the fitness of any configuration of the algorithm into the positive range. The configuration which achieves the best fitness is still not suitable to achieve a low false positive rate on many real life projects. Plotting true and false positives against different possible threshold values reveals that a lower threshold value can be used in order to get a lower false positive rate.

6.1 Algorithm Modifications

As shown in Table 5.1, the modifications done to this algorithm have a major influence on its quality measured by the fitness function. It is evident that the function wise normalization performs "fitter" than the traditional average normalization. The postprocessing functions do not show a similarly clear characteristic. However, a trend is visible, indicating that postprocessing has a limited positive effect on the fitness.

There are several possible hypotheses to explain the observed trends. They are not exclusive:

1. The test conditions were unfair.
    a) Either the test dataset or the modifications were developed dependent on each other. Using the same model for developing an algorithm and testing it inherently yields positive results that are not necessarily mirroring real conditions. This is called an inverse crime.
    b) The fitness function does not mirror the quality properly.
2. The proposed modifications are beneficial for the precision of the algorithm.

The test dataset and the two algorithm modifications were developed independent from each other except one clone sample designed to exploit the described weakness of the average calculation. When executing some tests without this test case (see table 5.2), results look similar: the function wise normalization still acquires a significantly better fitness. Thus, although hypothesis 1.a) applies partly, it cannot be considered a hint that hypothesis 2. is invalid.

Hypothesis 1.b) cannot be answered with a simple boolean answer. The test dataset is built on the data of Roy et al. as well as real true and false positives and thus closely related to the real application. Although it is clear that the fitness function is a very simple model that can be improved, it is closely related to the test dataset and may serve as a valid indicator for the algorithm quality if no inverse crime is applied.

All in all it can be stated that the fitness function does mirror the quality of the clone detection and that the function wise normalization as well as the general idea of postprocessing are beneficial for the quality of the code clone detection.

The main difference of the approaches is the convergence speed. The polynomial approach converges slower towards 0.75 than the exponential one and seems to be slightly superior according to the measurements.

The fact that applying both postprocessing methods at the same time yields sometimes better results can be attributed to either the convergence value of each postprocessing method (0.75) being too high or the ideal convergence speed lying in between the tested ones. It is likely that both reasons apply partly. Because the difference is very low, it can be stated that one likely can get good results with using only the polynomial postprocessing with a lowered threshold while avoiding the complexity of needing to use multiple postprocessing methods.

6.2 CC Weightings

The automated process of determining CC weightings is an inverse crime by definition and thus depends heavily on how close the reality relates to the test dataset. This implies that the fitness function can not be used to evaluate the quality of the weighting determination algorithm. To judge the usefulness of the determination algorithm, a better understanding of it and its results is needed.

Table 5.3 shows the values determined by the algorithm. Especially interesting are zeroes because those CCs are excluded from the difference calculation completely. We observe a weighting of zero for the following counting conditions:

- Used
- Is condition
- Is assignee
- In first level loop

- Is call parameter
- Is multiplied or divided or moduloed

The fact that 6 out of 18 CCs are not taken into account after applying the algorithm is suspicious. To understand, why these values were generated we need to take a further look at the fitness function. This function is designed such that a positive fitness means that the tested algorithm can satisfy a test dataset. The fitness function relies heavily on two sample pairs which are defining the actual fitness value - the clone with the highest difference and the non-clone with the lowest difference.

The defining clone pair for this configuration is `clearScreen()` and `scrollUp()`. This sample is a very interesting one because `scrollUp()` and `clearScreen` can be joined into a more generic function `scrollUp(int lines)` easily.

```
// Clone 1
void clearScreen() {
    uint16_t i;
    volatile unsigned char *videoram = (unsigned char *)FB_MEM_LOCATION;
    for(i=0; i<25*80*2; i++)
        if(i % 2)
            videoram[i]=COLCODE(STDFG, STDBG);
        else
            videoram[i]=BLANK;
    setCursor(GET_POS(0,0));
}

// Clone 2, for the general form, simply replace 1 by an int lines parameter
static void scrollUp() {
    uint16_t i;
    volatile unsigned char *videoram = (unsigned char *)FB_MEM_LOCATION;
    for(i=0; i<(25-1)*80*2; i++)
        videoram[i]=videoram[i + (1 * FB_COLUMNS * 2)];

    for(; i<25*80*2; i++)
        if(i % 2)
            videoram[i]=COLCODE(STDFG, STDBG);
        else
            videoram[i]=BLANK;
}
```

As we see, `scrollUp()` does contain a loop that is not executed in the special case `clearScreen()` because `lines` equals `25` - this is a major syntactic modification. Essentially this pair is syntactically a clone at sub function level which makes this sample very hard to detect with this algorithm: however the algorithm

obviously did find a way to separate this clone from the non-clone samples. The additional loop contains several constants used as a condition, an assignment in the first level loop body and two multiplications with several constants involved. This relates closely with the excluded counting conditions.

When looking at the most relevant non-clone pair analogue observations can be done: Both samples use exactly the same condition and the usage count of identifiers within the loops is similar. Lowering the importance of those counting conditions is a simple way to achieve a higher fitness.

Of course the fitness is not only defined by those two relevant samples: no other sample must remain between the relevant samples by definition. Still, the exact distribution of the other clone samples does not matter for the fitness.

The fact that the algorithm could easily optimize away those counting conditions can have mainly two possible reasons:

1. They do not matter.
2. The test dataset is incomplete and the CC determination algorithm exploited weaknesses in the test dataset.

From the high false positive rates shown in the evaluation against real life code bases it can be inferred that reason 1 does not apply: many illegitimate clone pairs had major differences with respect to loops. If code samples were present that exploited this weakness of the configuration, the algorithm would not be able to raise the fitness by simply turning off CCs.

Taking all this into account it can safely be stated that the test dataset is not sufficient for this kind of automated optimization. This also implies that the determined weightings are not a good basis for future research. However, the process of getting the weightings can lead to better results when applying a larger test dataset to it.

It is worth to note that the fitness function of the proposed algorithm is very simple. Improvements to this function are not able to solve the problem stated above but especially the negative values of the fitness function could be improved. If the algorithm is not able to satisfy a test dataset, the simple distance of the relevant samples is not a good metric to determine its quality. Instead, the number of false positives and false negatives could be used to indicate the negative quality of a clone detection more accurately.

6.3 Real Code Tests

In the Real Code Tests section, we saw false positive rates for various software projects. Furthermore, the total number of true and false positives over all of

those projects was plotted against a number of thresholds. Finally, we saw the detailed results for a lowered threshold of 0.185.

To properly interpret the data, we first have to gain an understanding of each code base and important properties affecting the code clone detection. Through those different properties the investigated code bases pose different (overlapping) challenges to the clone detection algorithm. The false positive rates of personal projects (Yafos and TE3D) are expected to be lower than on community maintained production software. In active open source projects, usually many people read code until it is committed on the main branch which leads to an extra protection layer against obvious code clones.

Yafos is a very small kernel written by the author of this thesis as a fun project. It does not rely on any outside libraries and does not contain any known clones[1].

TE3D is a terminal 3D engine written by a small group of students in the first semester. Because it uses vector and matrix structures with hardcoded dimensions and provides different routines to work with them it was suspected to contain many clones. TE3D does not rely on external libraries other than the C standard libraries.

CPython is the implementation of the interpreter which also runs the code clone detection. CPython is developed by an international community, mainly lead by an idealist, Guido van Rossum. In 2012 Coverity did large scale analysis on open source software projects: CPython surpassed all expectations with a "defect density" of 0.005 and was lauded to be a "model citizen of good code quality practices" [2]. Because of this, the clone detection was not expected to find many clones in this code base. The core CPython implementation depends only on the C standard libraries.

The Linux Kernel is a huge kernel developed by a large international community. Linux is well known for its unique review process: every commit needs to get approved by its creator, Linus Torvalds, in order to get into the mainline kernel. This sets the quality bar for new code rather high. We only evaluated the ext2 filesystem driver code: ext2 is still used and thus its code can be assumed to be actively maintained. On the other hand it is long superseded by ext3 and ext4 and thus not actively developed. Because of this, we expect the code clone detection to be able to detect a small portion of cloned code. Although it does not rely on potentially unresolvable dependencies, the project makes heavy use of macros which may influence the code clone detection.

Builder is a rather new IDE that was started by Christian Hergert in 2014. Christian has many years of experience with the C programming language

[1] The known clone present in the test dataset was actually identified by the stated code clone detection algorithm in the early stages.

> and the GLib which is used in Builder consistently. Because of the experience of the main developer and the novelty of the project, we expect to see few legitimate code clones in Builder. Builder uses types defined in the GLib almost exclusively (following an object oriented style in C). We expect the code clone detection algorithm to ignore large portions of code since the GLib include cannot be resolved automatically. The clone detection algorithm will have to deal with incomplete information for almost all functions.

The results of the Yafos project are positive while not astonishing and meet the expectations. The clone detection was able to detect three small but viable refactoring opportunities while yielding no false positives.

The analysis of TE3D yields many results, as expected. Mainly it uncovers the major design issue of this project: there are several hardcoded vector and matrix types which all have their own transformations reimplemented again and again. Furthermore, the detection of a vector addition function as a clone of various vector multiplication functions lead to the detection of a bug that resulted out of wrongly copied code. The illegitimate clone pair already hints at one important weakness of the core algorithm: constructor- and destructor-like methods. Methods that only initialize (or deinitialize) structs are inherently type 4 clones. Still, they cannot be generalized in most programming languages. This weakness was already stated to be problematic by Chen et al. who ignore constructors as well as Yang and Yao who ignore getters and setters and can be confirmed. Furthermore, it can be generalized that any function that only gets or sets member variables is probably not a useful clone. This is neither limited to small methods nor to constructor/destructor methods. Obviously a more sophisticated method is needed for ignoring such methods.

The GNOME Builder project raises the highest false positive rate. This is certainly mainly due to the fact that the include paths are intentionally unknown. The reason for the bad results are certainly to find in an implementation decision: the usage of the AST generated by Clang can be considered a bad decision. To create an algorithm that is stable against Macros and missing include paths, a lexical approach is recommended that does not rely on semantic relations of the source code.

In the CPython parser, the clone detection is able to detect eleven clone pairs and uncover one refactoring opportunity that applies to parts in many functions[2]. Most of those detected clones are on a sub function level. However, a large amount of the false positives yielded were unrelated functions which provide no refactoring opportunities. The false positive rate of almost 50% is unacceptable for production use.

[2] Many functions allocate memory and check immediately afterwards if the memory allocation was successful. If this check does not succeed, they exit the parser. This memory allocation and checking can easily be unified into one function/macro avoiding manual errors and lowering the amount of cloned code within the source code.

The clone detection yields more false positives than legitimate clones for the ext2 filesystem code. We can observe structurally similar false positive pairs again like a `get` and a `set` method which only gets or sets the given parameters out of or into a struct.

For the creation of the test dataset (as well as for the evaluation of the real code) a rather broad perception of a "code clone" was used. We have already seen that the algorithm is able to detect code clones at a sub function level to a degree and there exist at least one sample that is syntactically a clone of a smaller fragment than a function. This can be considered a problem because the algorithm is only designed to detect clones of full functions. To gather further facts, Figure 5.1 was created depicting the false positive rate against different threshold values. A clear trend is observable here: the clone pair false positive rate clearly relates to the threshold value - by lowering the threshold and our expectations to the detected clones the previous mistake can be corrected.

By lowering the threshold value to 0.185 we were able to exclude almost all false positives, while keeping many clones recognized. The algorithm can by design still recognize all type 1, 2, many type 3 and most syntactic type 4 clones with this lower threshold value and is thus better suited than most comparable software that is currently used while providing a very low false positive rate. Only the code of Builder still yields false positives with this configuration which can be attributed to the missing include paths.

6.4 Summary

The results we have seen lead to several conclusions: the modifications proposed to the original CMCD algorithm are beneficial for its precision. The CC determination algorithm is able to adapt the configuration to detect many clones of a given test dataset. However, a new approach for the fitness function would be beneficial, especially to improve the negative range of the fitness value. Furthermore, we determined that the test dataset was not sufficiently large to be used successfully with the CC determination algorithm without lowering the threshold manually. We identified a clone sample which is syntactically only a clone at sub function level to be problematic because CMCD is not designed to detect clones at sub function level.

The evaluation on the GNOME Builder project shows that the algorithm yields a high number of false positives when posed with incomplete information. The usage of the Clang AST was identified as a major problem with respect to this.

When studying other code bases, we saw also high false positive rates when applying the algorithm with the generated configuration. We also saw that the algorithm is able to detect even clones on a sub function level with this configuration. From several false positives, we were able to infer that not only getters, setters and constructors are unuseful type 4 clones, but that this group

can be generalized to all constructor-like methods that simply initialize a number of values. This confirms and extends the result of previous research on CMCD. We were able to see that the false positive rate can be controlled through a simple threshold adjustment, making it possible to trade off the number of missed clones for a low false positive rate. We noted that through lowering the threshold, no type 1, 2 or syntactic type 4 clones are excluded from the detection by the design of the algorithm. This renders the algorithm still superior to many alternatives and makes it easily adaptive to the user's needs. With a low threshold it can be used for production purposes as long as include paths are given manually where needed.

Chapter 7

Recommendations

This chapter summarizes possible improvements to the algorithm in order to reduce the effect of the identified weaknesses. Furthermore, it summarizes possible ideas for future work that can be built on the results stated here.

7.1 Algorithm Improvements

There are a lot of possible modifications that can still be evaluated to improve the performance as well as the precision of the algorithm described in this thesis.

The most important modification is obviously using a lexical approach instead of traversing an abstract syntax tree as this inherently makes the clone detection stable against missing information and can even work on semantically invalid code.

We have seen that the detection of clones at sub function level is possible to a degree, although the algorithm is not designed for this purpose. Adapting the algorithm to be able to find such clones by design could be a promising approach to overcome this major design limitation. A main problem when trying to identify clones at sub function level is probably performance because many more possible combinations need to be checked.

There are several approaches imaginable that can improve the performance of a CMCD variant while keeping or even raising its precision. Because the bipartite graph matching is a rather costly operation, it is reasonable to try to avoid it in trivial cases:

- CMs could be sorted by their size. Only similarly sized CMs should be chosen for the comparison.

- CMs can be compared by a fast bipartite matching algorithm that provides a result and has known error limits. This should be able to exclude many pairs, others can exactly be matched with the more expensive matching algorithm.
- Yuan and Guo propose idealizing similarities to a digital scale to speed up the actual matching [5]. They note that this approach does yield a performance gain while loosing some precision.

7.2 Future Research

We consider the lack of a sufficiently large and accepted test dataset containing clones as well as non-clones to be a fundamental problem in the field of code clone detection. Roy et al. have taken a first step towards making clone detection algorithms comparable and better evaluable with creating a small number of clone samples. However, a more diverse set of samples as well as non-clone samples are needed for a proper evaluation of a code clone detection.

For the special case of the CMCD algorithm and its variants a system could be built allowing users to submit true and especially false positives automatically and collaboratively. Any samples could simply be submitted in their form as a CM with the unweighted values for the counting conditions. Because the identification of code clones is subjective, this process would result in a test dataset that represents several opinions. Redesigning the CC weighting algorithm to find not the optimal solution but the best compromise would result in a code clone detection that suits the need of most participating users without any further configuration.

When creating such a test dataset and applying it to the CC determination algorithm, a negative fitness is inevitable. We have seen that the semantic meaning of negative fitness values is not very strong and needs an improvement. It might be beneficial to trade off a low difference value of a non-clone sample for high difference values of two or more other non clone samples to get more clones and non-clones correctly realized in the reality. A better fitness value in this case would be the negative number of wrongly recognized clones and non-clones while choosing the optimal threshold value.

The CC weighting determination algorithm itself is also improvable. The risk of it finding only a local maximum that is lower than another maximum is not neglectable and has to be evaluated. Extending it to a real genetic algorithm might be a possible approach to lower this risk with acceptable performance penalties.

Last but not least, it is desirable to create and implement a variant of CMCD that is as language independent as possible. Because the main parts of the algorithm are already designed and implemented in a language agnostic way, only the creation of CVs would need to be generalized. It is imaginable to

simplify CCs to regular expressions thus significantly easing the adaption needed to add a language to the code clone detection. Furthermore, the algorithm needs to handle different styles of programming properly. E.g. it is likely beneficial to create own CMs for member variables of structs in C and classes in languages that support object oriented style natively.

Ultimately, a major goal of the coala community is to explore scientific work on abstractions for language agnostic program transformations and analysis [3] further and use these and similar approaches to create the basis for performing arbitrary language-independent code analysis and transformations while only having to write one parser in order to get a universal set of analysis routines.

Appendix A

CMCD Usage

This section contains instructions needed to reproduce the results presented in this thesis or to apply the CMCD algorithm to an own code base.

To install coala, please follow the official coala installation instructions[1] or simply install coala via `pip3 install coala`. For using the CMCD, libclang needs to be installed. This can be done with `sudo apt-get install -y libclang1-3.4` on debian based systems.

The following commands can be used to run the CMCD code clone detection on arbitrary C code:

```
cd your_project
coala --bears=ClangCloneDetectionBear --files=**/*.c
```

Note: The `-s` flag can be appended to save this configuration persistently in a `.coafile`. `coala` can be invoked with no arguments next time to get the same result. Note that the `*` character is expanded by most shells: to save the glob correctly it needs to be escaped with a backslash in those cases.

The following command reveals all configuration options with respect to the CMCD algorithm (e.g. weighting CCs):

```
coala --bears=ClangCloneDetectionBear,ClangFunctionDifferenceBear \
--show-bears
```

The coala source code contains the test dataset and can be cloned with the following command:

[1] http://coala.rtfd.org/en/latest/Install/

```
git clone https://github.com/coala-analyzer/coala/
cd coala
```

The test dataset is located in `bears/tests/codeclone_detection/clone_detection_samples/`. The test suite for the CMCD algorithm and the related components can be executed with the following set of commands from the root of the coala source:

```
./run_tests.py -t ClangCountVectorCreatorTest CountVectorTest \
ClangCloneDetectionBearTest ClangCountingConditionsTest \
CloneDetectionRoutinesTest
```

Note: The `-c` flag can be appended, to get test coverage information. Note that it is possible that you do not get a full coverage unless you merge the coverage for all supported platforms and python versions.

For more information, please read the documentation given at http://coala-analyzer.org/.

Appendix B

Abbreviations

Table B.1: Abbreviations

Abbreviation	Long Form
CMCD	Count Matrix based Clone Detection
CM	Count Matrix
CV	Count Vector
CC	Counting Condition
AST	Abstract Syntax Tree
KM Algorithm	Kuhn–Munkres Algorithm

Appendix C

Tested Projects

The table below lists a repository URL and the commit SHA checksum or the mercurial changeset ID to allow identifying each project in a certain state uniquely.

Table C.1: Identity of Tested Sources

Project	Commit SHA/Changeset ID	Repository URL
Yafos	196909e0f9fa10afd7db2322fbeeb963a97ec0ed	https://github.com/sils1297/yafos
TE3D	9ac90fc2f0d101607598d7646efcec9d9b1c6739	https://github.com/Makman2/TE3D
CPython	0035fcd9b9243ae52c2e830204fd9c1f7d528534	https://hg.python.org/cpython
Linux	b787f68c36d49bb1d9236f403813641efa74a031	https://github.com/torvalds/linux
Builder	2b81117e9035bcc7efeb05766ed797470b35544b	git://git.gnome.org/gnome-builder

C.1 False/True Positives

This chapter lists signatures of pairs detected as duplicate. For each project it lists the number of true and false positive pairs first and then the number of unique functions involved in a true/false positive separated by a slash.

C.1.1 Yafos

C.1.1.1 False Positives

None.

C.1.1.2 True Positives

Threshold: 0.308: 3/6

Threshold: 0.185: 1/2

```
// Difference: 0.24
err_t loadModule(const void *, const void *);
err_t mapRegion(uintptr_t, uintptr_t, uintptr_t);
// Difference: 0.28
void scrollUp();
void clearScreen();
// Difference: 0.0
void genMemSet(genmem_t *, const uintptr_t, uint32_t);
void genMemFreeAdv(genmem_t *, const uintptr_t, uint32_t);
```

C.1.2 TE3D

C.1.2.1 False Positives

Threshold: 0.308: 1/2

Threshold: 0.185: 0/0

```
// Difference: 0.28
// Functions initialize similar members of different structs
void TE3D_ReleaseSurface(TE3D_Surface *)
void List_Clear(List *)
```

C.1.2.2 True Positives

Threshold: 0.308: 95/57

Threshold: 0.185: 34/47

```
// Difference: 0.27
void TE3D_Vector4f_normalize(TE3D_Vector4f *);
void TE3D_Vector2f_normalize(TE3D_Vector2f *);
// Difference: 0.27
TE3D_Matrix3x3f TE3D_Transformation3x3f_Rotate(TE3D_Vector3f,
    double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
// Difference: 0.26
TE3D_Matrix2x2f TE3D_Transformation2x2f_RotateOrigin(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
```

```
// Difference: 0.22
TE3D_Vector3f TE3D_Matrix3x3f_mul3(TE3D_Matrix3x3f,
    TE3D_Vector3f);
TE3D_Vector3f TE3D_Vector3f_cross(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.10
TE3D_Vector3f TE3D_Matrix3x3f_mul3(TE3D_Matrix3x3f,
    TE3D_Vector3f);
TE3D_Matrix2x2f TE3D_Matrix2x2f_mul(TE3D_Matrix2x2f,
    TE3D_Matrix2x2f);
// Difference: 0.18
TE3D_Vector3f TE3D_Vector3f_cross(TE3D_Vector3f, TE3D_Vector3f);
TE3D_Matrix2x2f TE3D_Matrix2x2f_mul(TE3D_Matrix2x2f,
    TE3D_Matrix2x2f);
// Difference: 0.24
TE3D_Vector2f TE3D_Vector2f_sub(TE3D_Vector2f, TE3D_Vector2f);
TE3D_Vector3f TE3D_Vector3f_add(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.0
TE3D_Vector2f TE3D_Vector2f_sub(TE3D_Vector2f, TE3D_Vector2f);
TE3D_Vector2f TE3D_Vector2f_add(TE3D_Vector2f, TE3D_Vector2f);
// Difference: 0.24
TE3D_Vector2f TE3D_Vector2f_sub(TE3D_Vector2f, TE3D_Vector2f);
TE3D_Vector3f TE3D_Vector3f_sub(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.24
TE3D_Vector3f TE3D_Vector3f_add(TE3D_Vector3f, TE3D_Vector3f);
TE3D_Vector2f TE3D_Vector2f_add(TE3D_Vector2f, TE3D_Vector2f);
// Difference: 0.0
TE3D_Vector3f TE3D_Vector3f_add(TE3D_Vector3f, TE3D_Vector3f);
TE3D_Vector3f TE3D_Vector3f_sub(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.18
TE3D_Vector3f TE3D_Vector3f_add(TE3D_Vector3f, TE3D_Vector3f);
TE3D_Vector4f TE3D_Vector4f_sub(TE3D_Vector4f, TE3D_Vector4f);
// Difference: 0.23, bug in TE3D_Vector4f_add
TE3D_Vector4f TE3D_Vector4f_add(TE3D_Vector4f, TE3D_Vector4f);
TE3D_Vector3f TE3D_Vector3f_cross(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.21, bug in TE3D_Vector4f_add
TE3D_Vector4f TE3D_Vector4f_add(TE3D_Vector4f, TE3D_Vector4f);
TE3D_Vector4f TE3D_Vector4f_div(TE3D_Vector4f, float);
// Difference: 0.21, bug in TE3D_Vector4f_add
TE3D_Vector4f TE3D_Vector4f_add(TE3D_Vector4f, TE3D_Vector4f);
TE3D_Vector4f TE3D_Vector4f_muls(TE3D_Vector4f, float);
// Difference: 0.11, bug in TE3D_Vector4f_add
TE3D_Vector4f TE3D_Vector4f_add(TE3D_Vector4f, TE3D_Vector4f);
TE3D_Vector2f TE3D_Matrix2x2f_mul2(TE3D_Matrix2x2f,
    TE3D_Vector2f);
// Difference: 0.30, bug in TE3D_Vector4f_add
TE3D_Vector4f TE3D_Vector4f_add(TE3D_Vector4f, TE3D_Vector4f);
```

```
TE3D_Matrix2x2f TE3D_Matrix2x2f_mul(TE3D_Matrix2x2f,
    TE3D_Matrix2x2f);
// Difference: 0.0
TE3D_Vector4f TE3D_Vector4f_N(float, float, float, float);
TE3D_Matrix2x2f TE3D_Matrix2x2f_N(float, float, float, float);
// Difference: 0.19
TE3D_Vector4f TE3D_Vector4f_N(float, float, float, float);
TE3D_VectorIndex3 TE3D_VectorIndex3_N(int, int, int);
// Difference: 0.19
TE3D_Vector4f TE3D_Vector4f_N(float, float, float, float);
TE3D_Vector3f TE3D_Vector3f_N(float, float, float);
// Difference: 0.21
TE3D_Vector2f TE3D_Vector2f_muls(TE3D_Vector2f, float);
TE3D_Vector3f TE3D_Vector3f_div(TE3D_Vector3f, float);
// Difference: 0.21
TE3D_Vector2f TE3D_Vector2f_muls(TE3D_Vector2f, float);
TE3D_Vector3f TE3D_Vector3f_muls(TE3D_Vector3f, float);
// Difference: 0.0
TE3D_Vector2f TE3D_Vector2f_muls(TE3D_Vector2f, float);
TE3D_Vector2f TE3D_Vector2f_div(TE3D_Vector2f, float);
// Difference: 0.24
TE3D_Vector2f TE3D_Vector2f_add(TE3D_Vector2f, TE3D_Vector2f);
TE3D_Vector3f TE3D_Vector3f_sub(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.27
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_Rotate(TE3D_Vector3f,
    double);
// Difference: 0.29
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
TE3D_Matrix2x2f TE3D_Transformation2x2f_RotateOrigin(double);
// Difference: 0.29
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
TE3D_Matrix2x2f TE3D_Transformation2x2f_RotateOrigin(double);
// Difference: 0.27
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_Rotate(TE3D_Vector3f,
    double);
// Difference: 0.14
TE3D_Matrix3x3f TE3D_Transformation3x3f_Scale(float, float,
    float);
TE3D_Matrix3x3f TE3D_Transformation3x3f_Translation(float,
    float);
// Difference: 0.0
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
// Difference: 0.0
```

```
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
// Difference: 0.20
TE3D_Matrix3x3f TE3D_Transformation3x3f_OrthogonalProjection(
    TE3D_Vector3f, TE3D_Vector3f);
TE3D_Matrix4x4f TE3D_Transformation4x4f_OrthogonalProjection(
    TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.0
// /home/lasse/prog/TE3D/test/linemodel/main.c
int main();
// /home/lasse/prog/TE3D/test/modelloader/main.c
int main();
// Difference: 0.19
TE3D_Matrix2x2f TE3D_Matrix2x2f_N(float, float, float, float);
TE3D_VectorIndex3 TE3D_VectorIndex3_N(int, int, int);
// Difference: 0.19
TE3D_Matrix2x2f TE3D_Matrix2x2f_N(float, float, float, float);
TE3D_Vector3f TE3D_Vector3f_N(float, float, float);
// Difference: 0.17
TE3D_Vector3f TE3D_Vector3f_div(TE3D_Vector3f, float);
TE3D_Vector4f TE3D_Vector4f_div(TE3D_Vector4f, float);
// Difference: 0.17
TE3D_Vector3f TE3D_Vector3f_div(TE3D_Vector3f, float);
TE3D_Vector4f TE3D_Vector4f_muls(TE3D_Vector4f, float);
// Difference: 0.0
TE3D_Vector3f TE3D_Vector3f_div(TE3D_Vector3f, float);
TE3D_Vector3f TE3D_Vector3f_muls(TE3D_Vector3f, float);
// Difference: 0.21
TE3D_Vector3f TE3D_Vector3f_div(TE3D_Vector3f, float);
TE3D_Vector2f TE3D_Vector2f_div(TE3D_Vector2f, float);
// Difference: 0.24
TE3D_Vector3f TE3D_Vector3f_div(TE3D_Vector3f, float);
TE3D_Vector4f TE3D_Vector3f_ExpandTo4(TE3D_Vector3f);
// Difference: 0.13
void TE3D_Vector4f_normalize(TE3D_Vector4f *);
void TE3D_Vector3f_normalize(TE3D_Vector3f *);
// Difference: 0.27
TE3D_Matrix4x4f TE3D_Transformation4x4f_OrthogonalProjection(
    TE3D_Vector3f, TE3D_Vector3f);
TE3D_Matrix4x4f
TE3D_Transformation4x4f_OrthogonalProjectionWithOffset(
    TE3D_Vector3f, TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.18
TE3D_Vector3f TE3D_Vector3f_sub(TE3D_Vector3f, TE3D_Vector3f);
TE3D_Vector4f TE3D_Vector4f_sub(TE3D_Vector4f, TE3D_Vector4f);
// Difference: 0.04
```

```
// /home/lasse/prog/TE3D/test/perspectivecube/main.c
int main();
// /home/lasse/prog/TE3D/test/cube/main.c
int main();
// Difference: 0.24
TE3D_Vector3f TE3D_Vector3f_cross(TE3D_Vector3f, TE3D_Vector3f);
TE3D_Vector2f TE3D_Matrix2x2f_mul2(TE3D_Matrix2x2f,
    TE3D_Vector2f);
// Difference: 0.0
TE3D_Vector4f TE3D_Vector4f_div(TE3D_Vector4f, float);
TE3D_Vector4f TE3D_Vector4f_muls(TE3D_Vector4f, float);
// Difference: 0.17
TE3D_Vector4f TE3D_Vector4f_div(TE3D_Vector4f, float);
TE3D_Vector3f TE3D_Vector3f_muls(TE3D_Vector3f, float);
// Difference: 0.24
TE3D_Vector4f TE3D_Vector4f_div(TE3D_Vector4f, float);
TE3D_Vector4f TE3D_Vector3f_ExpandTo4(TE3D_Vector3f);
// Difference: 0.0
TE3D_VectorIndex3 TE3D_VectorIndex3_N(int, int, int);
TE3D_Vector3f TE3D_Vector3f_N(float, float, float);
// Difference: 0.24
float TE3D_Vector2f_mul(TE3D_Vector2f, TE3D_Vector2f);
float TE3D_Vector3f_mul(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.16
void TE3D_Vector3f_normalize(TE3D_Vector3f *);
void TE3D_Vector2f_normalize(TE3D_Vector2f *);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
    float, float);
// Difference: 0.28
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
    float, float);
TE3D_Matrix3x3f TE3D_Transformation3x3f_Scale(float, float,
    float);
// Difference: 0.28
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_PerspectiveProjectionZ(
    double, float, float, float);
// Difference: 0.26
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
```

```
// Difference: 0.28
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
TE3D_Matrix3x3f TE3D_Transformation3x3f_Scale(float, float,
    float);
// Difference: 0.0
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
// Difference: 0.0
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.26
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
// Difference: 0.26
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
// Difference: 0.26
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
// Difference: 0.18
float TE3D_Vector4f_mul(TE3D_Vector4f, TE3D_Vector4f);
float TE3D_Vector3f_mul(TE3D_Vector3f, TE3D_Vector3f);
// Difference: 0.17
TE3D_Vector4f TE3D_Vector4f_muls(TE3D_Vector4f, float);
TE3D_Vector3f TE3D_Vector3f_muls(TE3D_Vector3f, float);
// Difference: 0.24
TE3D_Vector4f TE3D_Vector4f_muls(TE3D_Vector4f, float);
TE3D_Vector4f TE3D_Vector3f_ExpandTo4(TE3D_Vector3f);
// Difference: 0.0
TE3D_Vector2f TE3D_Vector2f_project(TE3D_Vector2f,
    TE3D_Vector2f);
TE3D_Vector4f TE3D_Vector4f_project(TE3D_Vector4f,
    TE3D_Vector4f);
// Difference: 0.0
TE3D_Vector2f TE3D_Vector2f_project(TE3D_Vector2f,
    TE3D_Vector2f);
TE3D_Vector3f TE3D_Vector3f_project(TE3D_Vector3f,
    TE3D_Vector3f);
// Difference: 0.0
TE3D_Vector4f TE3D_Vector4f_project(TE3D_Vector4f,
    TE3D_Vector4f);
TE3D_Vector3f TE3D_Vector3f_project(TE3D_Vector3f,
    TE3D_Vector3f);
// Difference: 0.29
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
```

```
    float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_PerspectiveProjectionZ(
    double, float, float, float);
// Difference: 0.09
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
    float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
    float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
// Difference: 0.26
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
    float, float);
TE3D_Matrix4x4f TE3D_Matrix2x2f_ExpandTo4x4(TE3D_Matrix2x2f);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_Translation(float,
    float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.20
TE3D_Matrix4x4f TE3D_Matrix3x3f_ExpandTo4x4(TE3D_Matrix3x3f);
TE3D_Matrix4x4f TE3D_Matrix2x2f_ExpandTo4x4(TE3D_Matrix2x2f);
// Difference: 0.27
TE3D_Matrix4x4f TE3D_Transformation4x4f_PerspectiveProjectionZ(
    double, float, float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
// Difference: 0.28
TE3D_Matrix4x4f TE3D_Transformation4x4f_PerspectiveProjectionZ(
    double, float, float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
// Difference: 0.28
TE3D_Matrix4x4f TE3D_Transformation4x4f_PerspectiveProjectionZ(
    double, float, float, float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.0
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateY(double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
// Difference: 0.26
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.26
```

```
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
// Difference: 0.26
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateZ(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
// Difference: 0.26
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateX(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
TE3D_Matrix4x4f TE3D_Matrix2x2f_ExpandTo4x4(TE3D_Matrix2x2f);
// Difference: 0.23
TE3D_Matrix4x4f TE3D_Transformation4x4f_Scale(float float,
    float);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.22
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
TE3D_Matrix4x4f TE3D_Matrix2x2f_ExpandTo4x4(TE3D_Matrix2x2f);
// Difference: 0.0
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateZ(double);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.21
TE3D_Vector3f TE3D_Vector3f_muls(TE3D_Vector3f, float);
TE3D_Vector2f TE3D_Vector2f_div(TE3D_Vector2f, float);
// Difference: 0.24
TE3D_Vector3f TE3D_Vector3f_muls(TE3D_Vector3f, float);
TE3D_Vector4f TE3D_Vector3f_ExpandTo4(TE3D_Vector3f);
// Difference: 0.15
TE3D_Vector4f TE3D_Vector2f_ExpandTo4(TE3D_Vector2f);
TE3D_Vector4f TE3D_Vector3f_ExpandTo4(TE3D_Vector3f);
// Difference: 0.07
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateOrigin(
    TE3D_Vector3f, double);
TE3D_Matrix3x3f TE3D_Transformation3x3f_RotateOrigin(
    TE3D_Vector3f, double);
// Difference: 0.22
TE3D_Matrix4x4f TE3D_Matrix2x2f_ExpandTo4x4(TE3D_Matrix2x2f);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateX(double);
// Difference: 0.22
TE3D_Matrix4x4f TE3D_Matrix2x2f_ExpandTo4x4(TE3D_Matrix2x2f);
TE3D_Matrix4x4f TE3D_Transformation4x4f_RotateY(double);
```

```
// Difference: 0.30
TE3D_Matrix4x4f TE3D_Transformation4x4f_PerspectiveProjection(
    double, float, float, float, TE3D_Vector3f, TE3D_Vector3f);
TE3D_Matrix4x4f
TE3D_Transformation4x4f_PerspectiveProjectionWithOffset(double,
    float, float, float, TE3D_Vector3f, TE3D_Vector3f,
    TE3D_Vector3f);
```

C.1.3 CPython (Parser)

C.1.3.1 False Positives

Threshold: 0.308: 8/13

Threshold: 0.185: 0/0

```
// Difference: 0.28
int s_push(stack *, dfa *, node *);
dfa * adddfa(grammar *, int, const char *);
// Difference: 0.31
int PyTokenizer_Get(struct tok_state *, char **, char **);
int findlabel(labellist *, int, const char *);
// Difference: 0.31
void dumpnfa(labellist *, nfa *);
void fixdfa(grammar *, dfa *);
// Difference: 0.30
// Functions initialize similar members of different structs
nfa * newnfa(char *);
dfa * adddfa(grammar *, int, const char *);
// Difference: 0.22
// Functions initialize similar members of different structs
grammar * newgrammar(int);
int addstate(dfa *);
// Difference: 0.24
node * PyParser_ParseFileObject(FILE *, PyObject *,
    const char *, grammar *, int, const char *, const char *,
    perrdetail *, int *);
int push(stack *, int, dfa *, int, int, int);
// Difference: 0.27
void fixdfa(grammar *, dfa *);
void PyGrammar_AddAccelerators(grammar *g);
// Difference: 0.24
void translatelabels(grammar *);
void PyGrammar_AddAccelerators(grammar *g);
```

C.1.3.2 True Positives

Threshold: 0.308: 11/16

Threshold: 0.185: 4/8

```
// Difference: 0.29
void compile_alt(labellist *, nfa *, node *, int *, int *);
void compile_rhs(labellist *, nfa *, node *, int *, int *);
// Difference: 0.29
void printdfas(grammar *, FILE *);
void printstates(grammar *, FILE *);
// Difference: 0.31
nfa * newnfa(char *);
nfagrammar * newnfagrammar(void);
// Difference: 0.16
nfa * newnfa(char *);
grammar * newgrammar(int);
// Difference: 0.29
node * PyNode_New(int);
grammar * newgrammar(int);
// Difference: 0.23
grammar * newgrammar(int);
nfagrammar * newnfagrammar(void);
// Difference: 0.14
node * PyParser_ParseStringFlagsFilenameEx(const char *,
    const char *, grammar *, int, perrdetail *, int *);
node * PyParser_ParseFileFlagsEx(FILE *, const char *,
    const char *, grammar *, int, const char *, const char *,
    perrdetail *, int *);
// Difference: 0.23
int push(stack *, int, dfa *, int, int, int);
int shift(stack *, int, char *, int, int, int);
// Difference: 0.17
void addnfaarc(nfa *, int, int, int);
void addarc(dfa *, int, int, int);
// Difference: 0.30
void addarc(dfa *, int, int, int);
dfa * adddfa(grammar *, int, const char *);
// Difference: 0.18
void printarcs(int, dfa *, FILE *);
void printstates(grammar *, FILE *);
```

C.1.4 Linux (ext2)

C.1.4.1 False Positives

Threshold: 0.308: 4/8

Threshold: 0.185: 0/0

```
// Difference: 0.28
int ext2_mknod(struct inode *, struct dentry *, umode_t, dev_t);
int ext2_link(struct dentry *, struct inode *, struct dentry *);
// Difference: 0.21
void rsv_window_remove(struct super_block *,
    struct ext2_reserve_window_node *);
void ext2_update_dynamic_rev(struct super_block *);
// Difference: 0.23
void ext2_set_inode_flags(struct inode *);
void ext2_get_inode_flags(struct ext2_inode_info *);
// Difference: 0.23
struct ext2_dir_entry_2 * ext2_dotdot(struct inode *,
    struct page **);
int ext2_mknod(struct inode *, struct dentry *, umode_t, dev_t);
```

C.1.4.2 True Positives

Threshold: 0.308: 3/6

Threshold: 0.185: 3/6

```
// Difference: 0.11
unsigned long ext2_count_free_inodes(struct super_block *);
unsigned long ext2_count_dirs(struct super_block *);
// Difference: 0.10
int ext2_create(struct inode *, struct dentry *, umode_t, bool);
int ext2_tmpfile(struct inode *, struct dentry *, umode_t);
// Difference: 0.0
unsigned long ext2_count_free_blocks(struct super_block *);
unsigned long ext2_count_free_inodes(struct super_block *);
```

C.1.5 Builder

C.1.5.1 False Positives

Threshold: 0.308: 35/28

Threshold: 0.185: 3/6

```
// Difference: 0.22
void ide_search_reducer_push(IdeSearchReducer *,
    IdeSearchResult *);
void ide_highlight_engine_set_buffer(IdeHighlightEngine *,
    IdeBuffer *);
// Difference: 0.25
void ide_makecache_new_worker(GTask *, gpointer, gpointer,
    GCancellable *);
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
// Difference: 0.16
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
void ide_context_unload_buffer_manager(gpointer, GCancellable *,
    GAsyncReadyCallback, gpointer);
// Difference: 0.30
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
void test_buffer_manager_basic_cb1(GObject *, GAsyncResult *,
    gpointer);
// Difference: 0.27
char * lskip(const char *);
char * ec_strlwr(char *);
// Difference: 0.20
void ide_highlight_engine_set_buffer(IdeHighlightEngine *,
    IdeBuffer *);
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
// Difference: 0.27
void ide_highlight_engine_set_buffer(IdeHighlightEngine *,
    IdeBuffer *);
void ide_source_snippet_after_insert_text(IdeSourceSnippet *,
     GtkTextBuffer *, GtkTextIter *, gchar *, gint);
// Difference: 0.0
// Functions initialize similar classes
void gb_preferences_window_class_init(
    GbPreferencesWindowClass *)
void gb_workbench_class_init(GbWorkbenchClass *);
// Difference: 0.30
void ide_makecache_new_worker(GTask *, gpointer, gpointer,
    GCancellable *);
void ide_source_snippet_after_insert_text(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, gchar *, gint);
// Difference: 0.29
void ide_makecache_new_worker(GTask *, gpointer, gpointer,
    GCancellable *);
```

```
void test_buffer_manager_basic()

// Difference: 0.30
// Functions essentially assign values to struct and reverse
void editorconfig_handle_get_version(const editorconfig_handle,
    int *, int *, int *);
void editorconfig_handle_set_version(editorconfig_handle, int,
    int, int);
// Difference: 0.29
void ide_makecache_new_worker(GTask *, gpointer, gpointer,
    GCancellable *);
void ide_project_add_file(IdeProject *, IdeProjectFile *);
// Difference: 0.30
void ide_project_add_file(IdeProject *, IdeProjectFile *);
void idedit__context_new_cb(GObject *, GAsyncResult *,
    gpointer);
// Difference: 0.24
void print_build_info(IdeContext *, IdeDevice  *);
void exercise1(test_state_t *, IdeBackForwardList *);
// Difference: 0.16
void ide_project_add_file(IdeProject *, IdeProjectFile *);
void exercise1(test_state_t *, IdeBackForwardList *);
// Difference: 0.28
void ide_project_add_file(IdeProject *, IdeProjectFile *);
void ide_device_manager_provider_removed(PeasExtensionSet *,
    PeasPluginInfo *, PeasExtension *, gpointer);
// Difference: 0.21
void ide_device_manager_provider_removed(PeasExtensionSet *,
    PeasPluginInfo *, PeasExtension *, gpointer);
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
// Difference: 0.23
void ide_device_manager_provider_removed(PeasExtensionSet *,
    PeasPluginInfo *, PeasExtension *, gpointer);
void ide_context_unload_buffer_manager(gpointer, GCancellable *,
    GAsyncReadyCallback, gpointer);
// Difference: 0.23
void ide_device_manager_provider_removed(PeasExtensionSet *,
    PeasPluginInfo *, PeasExtension *, gpointer);
void ide_source_snippet_after_insert_text(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, gchar *, gint);
// Difference: 0.25
void ide_project_add_file(IdeProject *, IdeProjectFile *);
void ide_buffer_manager_remove_buffer(IdeBufferManager *,
    IdeBuffer *);
// Difference: 0.25
```

```
void ide_project_add_file(IdeProject *, IdeProjectFile *);
void ide_highlight_engine_set_buffer(IdeHighlightEngine *,
    IdeBuffer *);
// Difference: 0.30
void ide_project_add_file(IdeProject *, IdeProjectFile *);
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
// Difference: 0.20
void exercise1(test_state_t *, IdeBackForwardList *);
void ide_device_manager_provider_removed(PeasExtensionSet *,
    PeasPluginInfo *, PeasExtension *, gpointer);
// Difference: 0.27
void exercise1(test_state_t *, IdeBackForwardList *);
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
// Difference: 0.21
void exercise1(test_state_t *, IdeBackForwardList *);
void ide_highlight_engine_set_buffer(IdeHighlightEngine *,
    IdeBuffer *);
// Difference: 0.25
IdeSourceSnippet * ide_source_snippet_copy(IdeSourceSnippet *);
IdeBackForwardList * ide_back_forward_list_branch(
    IdeBackForwardList *);
// Difference: 0.25
void ide_makecache_new_worker(GTask *, gpointer, gpointer,
    GCancellable *);
void ide_context_unload_buffer_manager(gpointer, GCancellable *,
    GAsyncReadyCallback, gpointer);
// Difference: 0.23
void ide_context_unload_buffer_manager(gpointer, GCancellable *,
    GAsyncReadyCallback, gpointer);
IdeDiagnostic * create_diagnostic(IdeClangTranslationUnit *,
    IdeProject *, const gchar *, GFile *, CXDiagnostic *);
// Difference: 0.28
void ide_source_snippet_after_insert_text(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, gchar *, gint);
IdeDiagnostic * create_diagnostic(IdeClangTranslationUnit *,
    IdeProject *, const gchar *, GFile *, CXDiagnostic *);
// Difference: 0.24
void ide_buffer_manager_load_file__load_cb(GObject *,
    GAsyncResult *, gpointer)
void idedit__context_new_cb(GObject *, GAsyncResult *,
    gpointer);
// Difference: 0.30
void gb_search_display_add_provider(GbSearchDisplay *,
    IdeSearchProvider *);
```

```
IdeProjectItem * ide_project_files_find_child(IdeProjectItem *,
    const gchar *);
// Difference: 0.27
void ide_buffer_manager_save_file__save_cb(GObject *,
    GAsyncResult *, gpointer)
void idedit__context_new_cb(GObject *, GAsyncResult *,
    gpointer);
// Difference: 0.31
void ide_buffer_manager_load_file__load_cb(GObject *,
    GAsyncResult *, gpointer)
void ide_buffer_manager_save_file__load_settings_cb(GObject *,
    GAsyncResult *, gpointer);
// Difference: 0.19
void ide_buffer_manager_save_file__load_settings_cb(GObject *,
    GAsyncResult *, gpointer);
void test_buffer_manager_basic_cb1(GObject *, GAsyncResult *,
    gpointer);
// Difference: 0.19
void ide_context_unload_buffer_manager(gpointer, GCancellable *,
    GAsyncReadyCallback, gpointer);
void ide_source_snippet_after_insert_text(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, gchar *, gint);
```

C.1.5.2 True Positives

Threshold: 0.308: 6/12

Threshold: 0.185: 5/10

```
// Difference: 0.0
// plugins/clang/ide-clang-symbol-node.c
enum CXChildVisitResult find_child_type(CXCursor, CXCursor,
    CXClientData);
// plugins/clang/ide-clang-translation-unit.c
enum CXChildVisitResult find_child_type(CXCursor, CXCursor,
    CXClientData);
// Difference: 0.24
void test_buffer_manager_basic_cb1(GObject *, GAsyncResult *,
    gpointer);
void test_buffer_basic_cb1(GObject *, GAsyncResult *, gpointer);
// Difference: 0.0
// test-egg-signal-group.c
gint main(gint, gchar **);
// test-egg-binding-group.c
gint main(gint, gchar **);
```

```
// Difference: 0.0
void save_state_free(gpointer);
void load_state_free(gpointer);
// Difference: 0.08
void ide_buffer_manager_load_file__load_cb(GObject *,
    GAsyncResult *, gpointer)
void ide_buffer_manager_save_file__save_cb(GObject *,
    GAsyncResult *, gpointer)

// Difference: 0.09
void ide_source_snippet_after_insert_text(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, gchar *, gint);
void ide_source_snippet_after_delete_range(IdeSourceSnippet *,
    GtkTextBuffer *, GtkTextIter *, GtkTextIter *);
```

Appendix D

Affidavit

I, Lasse Schuirmann, hereby confirm that my thesis entitled **Investigating the Influence of Different Counting Conditions in Count Matrix based Code Clone Detection** is the result of my own work. I did not receive any help or support from commercial consultants. All sources and/or materials applied are listed and specified in the thesis.

Furthermore, I confirm that this thesis has not yet been submitted as part of another examination process neither in identical nor in similar form.

References

[1] Chen, X., Wang, A.Y., and Tempero, E.D. "A Replication and Reproduction of Code Clone Detection Studies". *Thirty-Seventh Australasian Computer Science Conference, ACSC 2014, Auckland, New Zealand, January 2014*, Australian Computer Society (2014), 105–114.

[2] Coverity. "Coverity Finds Python Sets New Level of Quality for Open Source Software". 2013. http://www.coverity.com/press-releases/coverity-finds-python-sets-new-level-of-quality-for-open-source-software/.

[3] Kalleberg, K.T. "Abstractions for Language-Independent Program Transformations". 2007.

[4] Roy, C.K., Cordy, J.R., and Koschke, R. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". *Science of Computer Programming 74*, 7 (2009), 470–495.

[5] Yuan, Y. and Guo, Y. "CMCD: Count Matrix Based Code Clone Detection". *18th Asia Pacific Software Engineering Conference, APSEC 2011, Ho Chi Minh, Vietnam, December 5-8, 2011*, IEEE Computer Society (2011), 250–257.