UNIVERSITÉ DE MONTRÉAL

LEVERAGING SOFTWARE CLONES FOR SOFTWARE COMPREHENSION: TECHNIQUES AND PRACTICE

THIERRY M. LAVOIE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

LEVERAGING SOFTWARE CLONES FOR SOFTWARE COMPREHENSION:
TECHNIQUES AND PRACTICE

présentée par: LAVOIE Thierry M.
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. FERNANDEZ José M., Ph. D., président
M. MERLO Ettore, Ph. D., membre et directeur de recherche
M. ADAMS Bram, Doctorat, membre
M. DEAN Thomas R., Ph. D., membre

Tout ceci pour un chapeau de cow-boy...

ACKNOWLEDGEMENTS

Avoir le privilège d'accomplir un parcours universitaire de dix ans avec comme point culminant l'écriture d'une thèse est un honneur qui se doit d'être partagé avec tous les gens qui ont vu, participé, critiqué, soutenu et cru en moi. Il me fait un immense de plaisir de leur accorder une place particulière dans le préambule de mes travaux.

Tout d'abord, pour sept années réparties sur trois cycles et un nombre incalculable de projets et d'opportunités, je remercie le Professeur Ettore Merlo, directeur et mentor, de m'avoir accordé autant de liberté, de confiance et de défis et de m'avoir autant épaulé et encouragé. Ce fût, et je l'espère sera, un immense plaisir de vous avoir connu et d'avoir travailler pour et avec vous.

Je remercie mes parents, Murielle et Daniel, de m'avoir soutenu à travers le parcours le plus difficile qui soit. Je vous le promets, cette fois-ci je quitte définitivement l'université.

À ma soeur et à mon beau-frère, Maryse et Jean-Philippe, merci de m'avoir offert une vie comprise de 1000km de course et de poros. Votre accompagnement a été apprécié. Merci aussi à ces amitiés que vous avez partagées: Dave, Mr. Phil et Antoine.

Merci à mes collègues de laboratoire, Mathieu et Marc-André. Je vous souhaite à votre tour d'atteindre cette étape.

Un merci spécial à Pascal Potvin et son équipe, Mario, Marco, Gordon et Renaud de chez Ericsson pour m'avoir offert une chance unique d'avoir une expérience industrielle lors de mon passage à l'université. Votre influence m'a donné le goût de cette aventure.

Et finalement, à cinq grandes amitiés.

François, tu as été ce collègue qui fait de l'université ce qu'elle doit être: un lieu de haut échange intellectuel. J'ai été honnoré de travailler avec toi et de partager autant de jeux de société.

Elyse, merci d'avoir été là, merci d'avoir marché avec moi, merci pour les gâteaux, merci pour les projets un peu fou, de 20kg de pommes à la neige de l'Estrie jusqu'aux hanches, merci pour tout, tout simplement.

Sag, merci de m'avoir rappelé qu'il faut décrocher et garder les pieds sur terre.

Théotime, merci d'avoir été aussi joyeux et de m'avoir ouvert l'esprit un peu plus sur le monde.

Amélie, merci d'être cette constante dans ma vie qui me rappelle que les choses reviennent toujours à l'équilibre.

À vous tous, je vous serai toujours reconnaissant.

Additional Acknowledgements

This research was funded by NSERC, FQRNT, and Ericsson Canada.

Chapters 1, 2, 3, 5, 11, 12, and 13 were proofread by Ivanna Richardson.

Chapter 11 contains results on which the following people have contributed: Ettore Merlo, Pascal Potvin, Pierre Busnel, Gordon Bailey, François Gauthier, and Jean-François Li. They all gave their explicit consent to the use of some figures and text presented in that chapter. Ettore Merlo, Pascal Potvin, and François Gauthier also gave many comments that helped shape the discourse of that chapter.

RÉSUMÉ

Le corps de cette thèse est centré sur deux aspects de la détection de clones logiciels: la détection et l'application.

En détection, la contribution principale de cette thèse est un nouveau détecteur de clones conçu avec la librairie mtreelib, elle-même développée expressément pour ce travail. Cette librairie implémente un arbre de métrique général, une structure de donnée spécialisée dans la division des espaces de métriques dans le but d'accélérer certaines requêtes communes, comme les requêtes par intervalles ou les requêtes de plus proche voisin. Cette structure est utilisée pour construire un détecteur de clones qui approxime la distance de Levenshtein avec une forte précision. Une brève évaluation est présentée pour soutenir cette précision. D'autres résultats pertinents sur les métriques et la détection incrémentale de clones sont également présentés.

Plusieurs applications du nouveau détecteur de clones sont présentés. Tout d'abord, un algorithme original pour la reconstruction d'informations perdus dans les systèmes de versionnement est proposé et testé sur plusieurs grands systèmes. Puis, une évaluation qualitative et quantitative de Firefox est faite sur la base d'une analyse du plus proche voisin; les courbes obtenues sont utilisées pour mettre en lumière les difficultés d'effectuer une transition entre un cycle de développement lent et rapide. Ensuite, deux expériences industrielles d'utilisation et de déploiement d'une technologie de détection de clonage sont présentés. Ces deux expériences concernent les langages C/C++, Java et TTCN-3. La grande différence de population de clones entre C/C++ et Java et TTCN-3 est présentée. Finalement, un résultat obtenu grâce au croisement d'une analyse de clones et d'une analyse de flux de sécurité met en lumière l'utilité des clones dans l'identification des failles de sécurité.

Le travail se termine par une conclusion et quelques perspectives futures.

ABSTRACT

This thesis explores two topics in clone analysis: detection and application.

The main contribution in clone detection is a new clone detector based on a library called mtreelib. This library is a package developed for clone detection that implements the metric data structure. This structure is used to build a clone detector that approximates the Levenshtein distance with high accuracy. A small benchmark is produced to assess the accuracy. Other results from these regarding metrics and incremental clone detection are also presented.

Many applications of the clone detector are introduced. An original algorithm to reconstruct missing information in the structure of software repositories is described and tested with data sourced from large existing software. An insight into Firefox is exposed showing the quantity of change between versions and the link between different release cycle types and the number of bugs. Also, an analysis crossing the results from pattern traversal, flow analysis and clone detection is presented. Two industrial experiments using a different clone detector, *CLAN*, are also presented with some developers' perspectives. One of the experiments is done on a language never explored in clone detection, TTCN-3, and the results show that the clone population in that language differs greatly from other well-known languages, like C/C++ and Java.

The thesis concludes with a summary of the findings and some perspectives for future research.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS ET ABREVIATIONS

3PP	3rd Party Product
LCS	Longest Common Subsequence
LOC	Lines of Code
PDG	Program Dependency Graph
LDA	Latent Dirichlet Allocation
δ	Distance function
ϕ	Similarity function

CHAPTER 1

INTRODUCTION

Clone detection, as fascinating as controversial, is one of the most active fields in software engineering. Originating now more than 20 years ago, it focuses on the detection of similar code in software applications. Whether for plagiarism detection, bug detection, software evolution, software comprehension, or even simple curiosity, this discipline has produced many interesting ideas on how to improve software quality and maintainability.

However interesting the topic is, many problems are still considered unresolved. In the Dagstuhl seminar of 2012 [71], many unsolved issues were identified, but two major problems emerged as being of crucial interest: improving type-3 clone detection and finding useful industrial applications. A further problem has plagued the community for the past decade: the lack of an extensive methodology to evaluate clone detectors.

This thesis aims to provide solutions to the first two problems mentioned above. The main goal of this research is the design of an accurate type-3 clone detector by the means of estimating the Levenshtein distance. To assess the usefulness of the new method, some applications requiring extensive type-3 clone detection are proposed.

This thesis is divided into topics: the rest of the introduction will cover basic definitions and concepts followed by an overview of the research ideas; the second chapter presents a literature review of clone detection and string matching; the third chapter presents an overview of the technique; and the following chapters present published papers on the topics mentioned before. Further research opportunities are presented in the conclusion.

1.1 Basic Concept Definitions

Prior to any further discussion, we define some important concepts relevant to this thesis.

1.1.1 Definition of Clone Types

Providing a clear definition of all the clone types has proven to be a near-impossible task. Simply by analyzing the references provided in this thesis, many discrepancies are easily identified in the many definitions formulated by different authors. Disagreeing does not however argue against common ground shared among many clone definitions: typical clone

definitions are classified in 4 clone type categories[1]. We provide our own definition of those 4 categories inspired by the literature and by our own experience:

- Type-1: Identical clones. Those clones that have identical textual representation.
- Type-2: Parametric clones. Those clones that have one or more features, like parameters, variable types, or common operations, that can be factorized, making the clones parametrized on those features. If the parameter is not consistent within the clones, we say it is a type-2 inconsistent.
- Type-3: Gapped clones. Those clones part of whose structure is not syntactically identical. This may result from statement modifications, insertions, and deletions. Type-3 clones do not need to be semantically consistent, meaning they may exhibit different functionalities.
- Type-4: Semantic clones. Those clones who share part of their semantic, but may not share any common structural feature.

This thesis focuses on type-3 clone detection.

1.1.2 Definition of Similarity and Distance

First considering code fragments, clones always have at least an indirect reference to the notions of similarity and the dual concept of distance. Most of the clone detectors available today establish cloning relations based on the notion of similarity. Despite the possibility of computing clones with a similarity measure being a very attractive notion, laying strong mathematical foundations to similarity computation using a distance, instead of a direct similarity measure, is often the more elegant and better choice. To understand how these concepts will be used within this work, we will first define them:

- Similarity: A measure aimed at quantifying how many two objects share common structural features. It can be normalized in the interval $[0.0, 1.0]$, 1.0 meaning identical, 0.0 meaning sharing no features at all. In this work, a similarity function is noted ϕ
- Distance: A measure aimed at quantifying how much two objects differ in their structural features. A distance of 0.0 between two objects means identical objects. Distances are in general not normalized. In this work, a distance function is noted δ.

[1]Part of the literature now talks about type-4 and higher orders, instead of only type-4. The boundary surrounding the actual space of type-4 clones being elusive, it is hazardous to define clone types of higher order. Since this thesis focuses on type-3 clones, we do not feel it is appropriate to go further into the debates regarding type-4 clones and will thus only give a general definition of this type

In the case of normalized similarity, it might be tempting to define a distance based on the simple transformation $\delta = 1 - \phi$. Even though the dual meaning remains, in many cases this transformation results in a distance that does not have strong properties, like satisfying the axioms of a metric: hence the need to define the function as a distance first.

A remarkable group of distance functions called metrics are well-behaved functions defining a topology over a space and give rise to structures prone to optimization. Most of them by nature are not normalized and defining their exact similarity function counterpart is sometimes not trivial. Nevertheless, it is not necessary to define a dual function to interpret the results of a distance function in the context of similarity measures. It is sufficient to define the similarity criteria in terms of distances. For example, a large amount of similarity between objects translates to those objects being separated by a small distance. Thus, even if the literature usually defines clones in term of similarity, we will use the concept of distance instead to define clones. However, as we clearly showed the two concepts to be intertwined, we will say that code fragments are similar whenever we intend to say that they are separated by a (arbitrary) small distance if the context is not ambiguous.

1.2 Problem Definitions and Research Objectives

1.2.1 Accurate Clone Detection using Metric Tree, n-grams, and Non-Euclidean Metrics

Predating the other objective, the design of the new clone detector will be oriented towards an accurate approximation of the Levenshtein distance. There are two reasons to approximate the Levenshtein distance. First, the distance gives clone results of good quality. Second, while the results are good, the computation time is prohibitive. Thus, designing an approximation that preserves as much quality of the original distance as possible while reducing the computational cost is an interesting goal.

The key features of the tool are: the use of n-grams to interpret code fragments as frequency vectors, the use of different metrics to induce a topology on the vector space instead of an arbitrary similarity measure, and space partition and pruning by means of the metric tree to speed up the actual search for clones. Comparison of an extensive collection of clones against the oracle by the Levenshtein distance will be used to quantify and validate the accuracy. Curves showing the behavior of the error with respect to the distance are also provided. This is the first work to actually show the error curves. While the error cannot be lower than $O(logn)$ in general, it tends to be smaller for the approximation of smaller distances, and grows as the approximated distance grows. The curves thus provide further empirical results to enhance the theoritical fundation behind the approximation of

the Levenshtein distance.

This topic will be presented in detail in chapter 4, chapter 6, and chapter 7.

1.2.2 Exploring New Applications of Clone Detection Technology

New applications using type-3 clones applying to software evolution are presented. Using a nearest-neighbor search, a entirely new approach for clone detection, we aim at retrieving information in variations between software versions. We also discuss clones in test scripts compared to clones in general software. Additional results from clones in large industrial software and results obtained from cross-analysis of clones and security patterns are also presented.

This topic will be presented in detail in chapter 9, chapter 11, chapter 12, and chapter 13.

1.2.3 Contributions

This thesis is centered on work that has generated many other contributions, not limited to published papers. Here is the exhaustive list of those contributions:

- A total of 12 papers, cited 52 times according to Google Scholar.
- A robust implementation of the metric tree data structure, written in C++ and distributed under the BSD-4 clauses license.
- A robust parser for the PHP language.
- An accurate approximation of the Levenshtein distance, along with curves showing the behaviour of the error with respect to the distance.
- Seminal work on two languages never considered for clones: PHP and TTCN.
- The technique presented in chapter 4 was used as a mean of identifying intellectual property violations in a trial. This work was done at the request of Smart&Biggar, and under the supervision of Professor Ettore Merlo, who acted as the expert witness.

CHAPTER 2

LITERATURE REVIEW

The software engineering literature about clone detection, management, application, and evaluation is both well established and extensive. For close to 25 years researchers have contributed to this field and yet many problems are still open. This section presents a broad coverage of the literature. It is divided according to four main topics: clone detection, clone management, clone application, and clone evaluation. Some open problems are also highlighted. Additional references are provided in each paper for the relevant related work.

2.1 Clone Detection

Clone detectors are classified according to their paradigm. We summarize each of these paradigms with key related publications.

Token based clone detectors are among the most primitive clone detectors. Authors such as Kamiya [60], Jarzabek [20], Juergens [58], and Murakami [91] have proposed different variations of this paradigm. Most of the token based approaches use different flavors of hashing to identify clones. They also are agnostic of the syntactic structures because they only use a lexer and no parsing to identify structural boundaries. These clone detectors are relatively easy to implement, but they offer poor results when dealing with type-3 clones.

An exotic variant of token based clone detection based on suffix trees instead of hash functions was developed by Göde and Koschke [45]. Using many heuristics, they incorporate some syntactic features without the use of a parser.

Metrics based clone detection like the one presented in [85] compute metrics on syntactic units. The units are then represented by a vector and clustered to identify the clones. A final filtering step with the Levenshtein distance is usually done to improve the precision of the results. Contrary to token based detectors, metric based require a parser which increases the technical difficulty to implement the method. This method is usually slowed down by the quadratic component in the final filtering.

Matching Program Dependency Graphs (PDG) has also been successfully applied to clone detection by Li [83] and Krinke [74]. Clone detection based on program slices, which are computed from PDGs, was introduced by Komondoor [67]. Those techniques are known to be prohibitively slow and cannot scale to systems above a few hundred kLOC.

Clone detection using memory footprint behavior analysis was first presented by Kim [64].

Prohibitive running time, dependency on the knowledge of the memory behavior, semantic of external libraries, and the complexity of the matching function has confined this approach to a limited application. However, on limited examples, it claimed to perform better than most of the token based and graph matching clone detectors. Comparison with this tool is limited since it depends on proprietary software.

NiCad is a hybrid paradigm created by Cordy [30]. It uses transformational grammar operations with the TXL language code standardization and normalization, and a filtering step with the length of the LCS. Scalable, fast, and offering good performance on type-3 clones are key features of NiCAD.

For an exhaustive coverage of clone detection technology, the reader should refer to [98].

Although varied, these approaches are always either limited by their execution time or in the quality of their results. Also, even though clone detectors are by nature search tools, none of the existing approaches uses data structures to optimize the search space or even to explicitly define the clone detection as a search query. Our proposed tool has improved performance as compared to existing tools by making use of a sophisticated data structure and defining two search primitives. It also perform better by being the first to work on n-grams, gaped n-grams, and exploring metrics outside of set similarity measurements.

2.2 Approximate String Matching

The Levenshtein distance [82] has been extensively used in post-filtering of the results and evaluation purpose. This distance computes the optimal number of editing operations (insertion, deletion, changing) to morph one string into another. The classic implementation using dynamic programming (DP) by Wagner and Fisher [114] is still the tool of choice; a good textbook reference on that algorithm is [31]. The same algorithm was first discovered in the context of DNA sequence matching by Needleman and Wunsch [93] a few years before Wagner and Fisher. A common variation with emphasis on local alignment is the Smith-Waterman algorithm [107]. All these algorithms have worst-case running time in $O(nm)$, where n and m are the length of the two aligned strings. This is in practice reported to be in $O(n^2)$ by making the assumption of equally lengthened strings. The DP algorithm may be enhanced using an unusual optimization dubbed the "four Russians trick" [12], leading to a $O(\frac{n^2}{log^2\ n})$. For any practical purposes, this speedup is not likely to provide any measurable gain especially considering the hidden constants of the "four Russians trick".

A more convenient algorithm is the one proposed by Ukkonen [113] and later improved by Berghel and Roach [24]. Although still a DP algorithm, both approaches propose a transformation of the DP matrix to exploit the monotonicity of the Levenshtein distance to

exclude cells from the computation. The complexity of the algorithm is $O(n+m+s^2)$, where s is the edit distance between the two strings, and n and m the length of the strings. In particular, this leads to a $O(n+m)$ running time for the two strings with an edit distance $s = 0$. In practice, for strings with a relatively small edit distance, there is a remarkable gain in time notably for very large strings. For clone detection, this algorithm provides a very interesting opportunity since by the nature of clones, we expect them to be somehow similar. Another alternative for speeding up the computation of the Levenshtein distance is the use of a GPU. The seminal work with GPU in clone detection is [77].

Approximate string-matching[1] has also been studied in the search for fast, albeit sub-optimal, matching. An interesting work using thr Number Theoretic Transform (a generalization of the Fourier transform over a ring) and prefix indexes has been published by Percival [96]. This work is optimized towards binary patch compression and is thus not completely relevant to this work.

Other works by Andoni [10, 11] have highlighted some theoretical bounds on the approximation of edit-distances, stating the impossibility of achieving an approximation with constant distortion by using any embedding in $\mathbb{R}^n$ with any of the standard metrics l_i (l_1, Manhattan distance, and l_2, Euclidean distance, are the most common) with a sub quadratic running time. However, these theoretical results do not disprove the possibility of having a good practical approximation using embedding in $\mathbb{R}^n$: they simply state the impossibility of having a well-structured distortion in general. They also do not state anything about the possible existence of a well-behaved distortion localized on small edit distances: this work exhibits an experiment showing evidence that approximating an edit distance for a small value can be easier than approximating larger ones.

2.3 Clone Management and Applications

Numerous applications have been published. We will limit the discourse here to a few selections regarding software evolution and bug fixing.

A good example of a recent study of applications of similarity between common code base is in [51]. That work identified many replica of common parts from a mobile operating system. In the case of large scale development, these duplications, which never merged together, generate more maintenance and co-maintenance tasks. Another good example of an origin study is [47].

[1] This standard terminology is somehow confusing. Colloquially, only the term string-matching is used, thus leading one to believe approximate string-matching to be an approximate solution. On the contrary, it does not refer to approximate results on the string-matching problem, but to the problem of matching strings that are approximately the same. For example, the Levenshtein distance computes the exact edit distance between two strings and thus matches strings that are approximate to one another.

Using clones to detect errors and bugs is also a widespread idea. Recent work by [83] and [56] extensively used clones to detect inconsistencies in programs and to suggest potential bugs. In both cases, the clone detector leverages information from a defective piece of code to detect latent bugs. The main limitation of these approaches lies in the fact that a defective chunk of code is required *a priori*.

2.4 Clone Evaluation

The seminal and most commonly cited work in clone evaluation is the study by Bellon [23]. This study was designed by manually inspecting a sample of clones in some systems and then extrapolating the results on the whole corpus by statistical means. Precision and recall were the main measures of the study. Conclusions of the study stated that clone detectors usually have higher precision than recall, with a small number having higher recall than precision, but none achieved a good balance between precision and recall, hence the need for improvement in the quality of the results. The benchmark from Bellon has been repeated notably by Higo [91] with his tool FRISC and NiCAD, but the overall distribution of the results were not significantly dissimilar from the results obtained by Bellon.

A more recent idea for clone detector comparison is the use of injection frameworks by Cordy and Roy [102]. Although injection is not new and was predated by Bellon, automating it by means of transformational grammar operations as proposed by Cordy and Roy would allow a larger scale deployment of such a benchmark. A major drawback with that approach is its intrinsic bias towards precision measurement and not recall (one has to know the universal definition of a clone to perfectly measure recall, and nobody agrees on a universal definition).

Automated oracles based on mathematical criteria are also present in the literature. Oracling type-3 clones with the Levenshtein distance has been applied by [109], and then extended to a larger scale by [78]. Those techniques are of great interest, measuring the distortion of the technique presented in this proposal, although they may be criticized in the context of general precision and recall evaluation.

CHAPTER 3

TECHNICAL IMPLEMENTATION, PRELIMINARY CONSIDERATIONS, AND RESEARCH OVERVIEW

This chapter aims to briefly introduce all the papers that make up the body of this thesis. It also introduces most of the software used as the common basis for the experiments in all the papers as well as an overview of the matching technique. The matching technique is only summarized in this chapter as the key components relevant to each experiment are detailed in the individual papers.

3.1 Implementation Of mtreelib

While implementation details of the core libraries developed as part of the new clone detection tools are not themselves experimental and scientific results, the experience and the knowledge acquired while writing this code is highly relevant in a software engineering context. For this reason, this section is dedicated to an exhaustive description of the main library used for clone detection in this thesis, the mtreelib.

The mtreelib library was jointly developed by my advisor, Ettore Merlo, and myself over the course of this thesis, from 2011 to 2014. The library and the source code is distributed upon request under the BSD-4 clauses license. The package includes the source code, a pre-configured makefile, a test suite to validate the library after compilation, and a bundle of example binaries helpful to understand how to use the library. This package is itself an important contribution of this work, as mtreelib was created to help other researchers do work and experiments with a metric tree in a more convenient fashion.

The design of mtreelib is centered around extensivity, flexibility and resilience, because the mtree data structure can naturally come in many flavors. The core concept of the mtree, a space partitioning structure[1], is to lessen the very strict hypotheses of many spaces, like the $\mathbb{R}^n$ family, in order to have a data structure that applies to more general spaces. One could argue that since the $\mathbb{R}^n$ family is of interest, structures like k-d trees, box trees, quad-trees, and all their generalizations should be enough, but they limit the objects of interest to tuples of reals indexed according to the l_i metric family. Using a metric tree allows for two things:

[1]While space partitioning is the correct term, it is misleading to many people as it is often associated with the $\mathbb{R}^n$ family of space or other spaces relevant to geometry or physical space. In this context, space should be interpreted in the more general topological sense of space, and even more precisely as any general metric space.

- Extending objects of interest to strings, as long as we can form a space with strings structured with a metric
- Extending the metrics of interest to vectors with non-linear metrics, like sine, variants of cosine, and some normalizations, without any complications related to the more specialized data structure like the k-d tree

Since exploring different metrics and different representation of the code, such as strings or tuples of metrics[2], are a key part of this research, the natural flexibility provided by the metric tree is very convenient. This allows us to build software around a standardized space partitioning data structure for fast indexing.

Like the STL, the current implementation is a completely templatized framework. Instead of using inheritance for different implementation of the metric tree, we chose to rely on static templates and static instantiations to prevent against involuntary type-casting between incompatible metric trees. The metric tree instantiation is done on two template parameters: the metric and the element type. While metrics are often compatible with only one element type, element types usually have a lot of legal metrics. To prevent against mismatch between the metrics and the element types, explicit template specializations are done in the header files. Also, the implementation of the metric trees is not done within the header files like most templates, but is instead done in a standard source file with every specialization provided in concrete implementation. That way, the compiler instantiates every legal implementatio when the library is compiled and not when the client program is compiled. While the error messages provided by the compiler when compiling the client code may be cryptic, it will at least always refuse to compile an improper instantiation, one with incompatible parameters, of the metric tree. While tedious, these precautions lead to a resilient as well as flexible design, allowing us to add new objects and new metrics in the library without too much effort by using template code instantiation, and preventing developers from making type incompatibility mistakes at compile time instead of runtime.

3.2 Explanations on the order of the presented papers

The first three papers presented in chapters 4, 6, and 7 are the core presentation of the idea behind the new clone detection technique developed in this thesis. While section 3.1 introduces the technical details of the software implementing the technique, these three chapters present experimental results as well as a general quality assessment of the technique. The first paper describes the quality of the proposed approximation, the second paper discusses

[2]This use of the word metric might be a little confusing. It refers to metrics gathered from the code, and not a distance satisfying the metric axioms.

different metric configurations as well as some metric equivalencies, and the last one describes a modification of the technique to support incremental clone detection.

The second of the three papers presented in chapters 9, 11, and 12 are the main applications and experiments conducted using the aforementioned clone detection technique. The three applications that are explored are distinct. The first application measures the average change in all functions in a system and links it to two different development cycles to assess the real amount of work between different system versions. The second application explores clones in test suites. The final application reconstructs file repository structures. Additional and complementary results are presented in chapter 13, for a total of five different applications investigated in this thesis. A general discussion providing a global perspective on the work is then presented before the conclusion.

CHAPTER 4

PAPER 1: AN ACCURATE ESTIMATION OF THE LEVENSHTEIN DISTANCE USING METRIC TREES AND MANHATTAN DISTANCE

4.1 Abstract

This paper presents an original clone detection technique which is an accurate approximation of the Levenshtein distance. It uses groups of tokens extracted from source code called windowed-tokens. From these, frequency vectors are then constructed and compared with the Manhattan distance in a metric tree. The goal of this new technique is to provide a very high precision clone detection technique while keeping a high recall. Precision and recall measurement is done with respect to the Levenshtein distance. The testbench is a large scale open source software. The collected results proved the technique to be fast, simple, and accurate. Finally, this article presents further research opportunities.

4.2 Introduction

The Levenshtein distance is a well known similarity measure between strings. It computes the number of insertions, deletions and character swaps to transform a string s_1 into a string s_2. It is conceptually very close to how humans edit text. All its characteristics make the Levenshtein distance a good candidate for clone detection. However, it runs in worst-case $O(n^2)$ which is prohibitive in most applications. However, to overcome the worst-case scenario, one can try to rely on an embedding of the distance in a more common space, such as $\mathbb{R}^n$, and hope to have a low distortion on the new space metric. Unfortunately, it has been proved [11, 73] that the Levenshtein distance cannot be embedded with a constant distortion in $\mathbb{R}^n$ without spending at least $O(n^2)$ time to compute the embedding. Even if one wants to rely on an $O(log\ n)$ distortion, the required time to compute the embedding is superlinear.

However, clone detection do not always need to know the exact distance between clones as long as the reported clones are significant. Furthermore, clones are required to be either identical or to share similar traits; there's no interest in finding pairs that do not share most of their features. This is equivalent to saying that the fragments in a clone pair are separated by an arbitrary small distance. Thus, even though it is impossible to compute a practical embedding of the Levenshtein distance in $\mathbb{R}^n$, it could be sufficient to have one that gives a precise answer to a binary question: is the distance between two fragments less than a fixed value ? This leads to the following research questions:

- Is it possible to use the Manhattan distance on windowed-tokens frequency vectors to obtain low-distortion results equivalent to binary Levenshtein queries ?
- What is the expected limit on which the Manhattan distance stops emulating Levenshtein properly ?
- Does the window size favorably influence precision and recall of the results ?

This paper introduces an original technique that gives encouraging practical answers to these questions.

The rest of the paper is divided as follows: section 4.3 gives a rationale to the technique using clone detection state-of-the-art, section 4.4 details the algorithm, section 4.5 presents an evaluation of the technique, section 4.6 discusses the obtained results, and section 4.7 finally summarizes the work and suggests further research opportunities.

4.3 Literature Review

Clone detection state of the art includes different techniques. For type-1 and type-2, AST-based detection has been introduced by [21]. Other methods for type-1 and type-2 include metrics-based clone detection as in [86], suffix tree-based clone detection as in [45], and string matching clone detection as in [38]. For a detailed survey of clone detection techniques, a good portrait is provided in [98].

The introduced technique in this paper refers to a widely spread idea in the literature: the Levenshtein distance is effective at finding meaningful clones. Cordy and Roy have introduced NiCAD in [101], a clone detection tool based on many prefiltering algorithms with a final filtering step using the length of the longest common subsequence (LCS). The LCS is dual to the Levenshtein distance and both can be viewed as roughly equivalent since the Levenshtein distance naturally provides a lower bound on the LCS length. The CLAN tool by Merlo in [86] also suggests to use LCS to filter results for better precision. The difference between NiCAD and CLAN is the number and the complexity of each step before the LCS filtering: NiCAD uses many lexical and syntactic preprocessing steps whereas CLAN uses a single metric projection step. The Levenshtein distance has also been suggested as a reference to benchmark clone detection. Tiarks et al. conceived a small reference set inspired by the Levenshtein distance in [109]. An exhaustive set of all pairs satisfying a chosen Levenshtein distance threshold was proposed by this paper's authors in [78]. Without claiming the Levenshtein distance produces the absolute clone reference, it is still reasonable to assume it produces good results. Consequently, there is a motivation to create a tool which produces results similar to those computed with the Levenshtein distance.

In the next section, algorithmic details are provided.

4.4 Algorithm

The proposed algorithm combines different known ideas for clone detection along with new ones. Token-based clone detection was first proposed by [60]. Other authors [20, 101] have since used tokens with different algorithms and token-based clone detection is now a family of clone detection tools. Code metric based clone detection was introduced by [86] and developed in [88, 89]. The idea of code metric clone detection is to chose software syntactic features, such as statements, branching instructions, function calls, etc., and to build a vector for which each dimension is a selected feature. The value of each vector component is the frequency of the corresponding feature. Syntactic analysis is first done to compute the frequencies to then build the vectors. These are then compared using a similarity criterion, such as the euclidean distance, cosine distance, etc. The original technique of [86] used space discretization for clone clustering. The new algorithm presented in this paper combines token analysis and code metric to create a vector: it builds vectors of token frequencies. It is not limited to the use of single tokens, but can also be extented to n-grams which we called windowed-tokens. With the new vectors, similarity is computed according to the Manhattan distance, also known as l_1 metric. Since finding all pairs of similar vectors requires $O(n^2)$ time, the algorithm relies on a metric tree to accelerate this process. Each step of the algorithm is now detailed.

The first step of the algorithm is to extract the tokens from the software source code. This is done with a lexical analyzer on a file basis. Using lexical analysis instead of syntactic analysis has some advantages. It is faster, first, since most of the times it relies only on regular expression matching instead of context-free grammar matching, and second, it is also usually easier to write a lexer than a parser.

The second step uses extracted tokens from the files to build the frequency vectors. The base case is to use single tokens or windowed-tokens of length 1. A unique id number corresponding to its represented dimension is assigned to every different token type. The tokens id are generated dynamically. Each time a new token type is encountered, the next available id, starting with id 0, is assigned to the newly discovered type. After the token type has been identified, the corresponding vector component is incremented by one. Every component in the vector has an initial value of 0. For better memory storage, the frequency vectors are not stored in a vector-array data structure but rather are in a hash table. Since code fragments have a frequency of 0 for most token types, the hash table will reduce memory usage. This may not be trivially apparent, but if we allow windowed-tokens of length 2 or more, storing

data in vector-arrays is not an option because of storage capacity. For example, in a language with 200 token types, the required array length to store every components explicitly is 200 for window length 1, 19 900 for length 2, and 1 313 400 for length 3. In general, the vector length is described as:

$$\frac{t!}{(t-l)!} \tag{4.1}$$

where t is the number of token types in the language and l the window length. This equation is of course growing exponentially with respect to the length of the window and thus compels for a better memory storage. Since to any fragment there cannot be more token types associated to it than its number of tokens, the hash table will use a storage linear in the number of tokens.

To extend the base case to a window length above 1, the same procedure is used but token type identifiers are assigned on an n-gram basis. For example, let the tokens of a language be {A,C,G,T}, and let s_0 be:

$$s_0 = ATGCGTCGGGTCCCAG \tag{4.2}$$

a random string. With a window of length 1, id assignment would be:

$$(A, 0)\ (T, 1)\ (G, 2)\ (C, 3) \tag{4.3}$$

and s_0 frequency vector v_{s_0}:

$$v_{s_0} = (2, 3, 6, 5) \tag{4.4}$$

With a length 2 window, id assignment would be:

$$(AT, 0)\ (TG, 1)\ (GC, 2)\ (CG, 3)\ (GT, 4) \tag{4.5}$$

$$(TC, 5)\ (GG, 6)\ (CC, 7)\ (CA, 8)\ (AG, 9) \tag{4.6}$$

and s_0 frequency vector v_{s_0}:

$$v_{s_0} = (1, 1, 1, 2, 2, 2, 2, 2, 1, 1) \tag{4.7}$$

and so forth with higher window lengths. The reader should note that the total number of token types in the second example, 10, is less than the theoretic maximum, 12. This is almost always the case with higher window lengths and thus it supports the idea that a hash

table will consume much less memory than a vector-array.

The third step, after extracting the tokens from software and building their corresponding frequency vectors, is to build a metric tree [28, 78] with all the resulting vectors under a chosen distance. The l_i metric family is a natural choice. The goal of the algorithm is to be as precise as possible and to have a fine grain adjustable threshold on the similarity criterion. The l_1 metric, or Manhattan distance, is the best choice in the l_i family according to our criterion. The motivation behind this choice is a simple geometric observation. Using a sketch proof, it will be shown that at each step, the l_1-sphere is always smaller than the l_2-sphere. An analogous reasoning can then be applied to show that an l_i-sphere is always smaller than an l_j-sphere if $i < j$. Lets define an l_i-sphere $\mathbf{S}_{i,\delta}$ as the set $x \in \mathbb{R}^n | l_i(x - y) \leq \delta$ for a fixed n. In $\mathbb{R}^2$, the l_1-sphere $\mathbf{S}_{1,\delta}$ is thus a $\frac{\pi}{4}$ radians rotated square of side-length $\sqrt{2\delta}$ and the $l_2 - sphere$ $\mathbf{S}_{2,\delta}$ is a circle of radius δ. In $\mathbb{R}^n$, every point x in $\mathbf{S}_{1,\delta}$ have a distance to the origin equal to:

$$l_1(x, 0) = \sum_{i=0}^{d} |x_i| \tag{4.8}$$

and the distance of every point in $\mathbf{S}_{2,\delta}$ to the origin is:

$$l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{4.9}$$

Now we have:

$$l_1(x, 0) = \sum_{i=0}^{d} |x_i| > l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{4.10}$$

$$l_1(x, 0) = \sum_{i=0}^{d} \sqrt{x_i^2} > l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{4.11}$$

which holds $\forall d \geq 2$ and $x_i \neq 0$. Thus, an l_1-sphere covers less space than an l_2-sphere since there are some points in the l_2-sphere that do not belong to the l_1-sphere, but the l_1-sphere is entirely comprised in the l_2-sphere. It should be obvious that this argument holds for every metric in the l_i family. It follows that the best choice to have a fine control over the sensibility of the algorithm is l_1.

The Manhattan distance, l_1 metric, between two vectors u and v is defined as:

$$l_1(u,v) = \sum_{i=0}^{d} |u_i - v_i| \tag{4.12}$$

One can recall that higher metrics require root extraction which is an expansive operation. Clearly, l_1 is the fastest to compute in the l_i family. Being the best for precision control and the fastest to compute, it is a natural choice. It can be found in every undergraduate textbook in geometry that l_1 satisfies the metric axioms (non-negativity, symmetry, identity and triangle inequality). Fulfilling such axioms allows us to use it in a metric tree.

The fourth step is to build a metric tree with all the vectors. A metric tree is a data structure that separates a search space according to a metric in order to increase the speed of range-queries and nearest-neighbours searches. The structure was first presented in [28], and first used in clone detection in [78] to produce an automated Levenshtein oracle. It supports 3 primitives: $insert(v), rangeQuery(v,\delta), nearestNeighbour(v)$. The primitive $insert(v)$ inserts vector v in the tree, $rangeQuery(v,\delta)$ finds every vector v' in the metric tree for which $l_i(v,v') < \delta$, and $nearestNeighbour(v)$ finds the vector v' in the tree that satisfies $l_i(v,v') \leq l_i(v,v_j) \forall j$. For clone detection, both query primitives may be useful, but the range-query is more consistent with the idea of finding every fragment within a specific distance. It is worth noting that once built, the tree may be dumped on disk to execute many analyzes on it. However, as it will be shown in section 4.5, combining both the building and the querying time does not add a significant amount of overhead.

The fifth and last step is the tree querying step. The overall routine is the same as the one in [78] and is presented in figures 4.1 and 4.2, for absolute and relative radii respectively. For the relative radius version of the algorithm, the radius is defined with respect to the overall length of the fragments. Let a, b be two vectors associated to code fragments. Lets define $len(a), len(b)$ as the sum of their components. Then the threshold ϵ for pair (a,b) is defined as:

$$\epsilon = distance * max(len(a), len(b))) \tag{4.13}$$

where $distance \in (0,1)$ is the coefficient of desired maximum distance. Another meaningful interpretation of *distance* is that the desired similarity between fragments is $(1-distance)$. The cloning criterion may be phrased as: the pair (a,b) is in a clone relation *iff* the Manhattan distance between a and b is smaller or equal to threshold ϵ, where ϵ is the maximum length of a and b times the distance coefficient or relative distance (or times one minus the similarity coefficient). Thus, the queries' *radius* is proportional to the length of the fragments and the varying parameter is the distance coefficient and not directly the Levenshtein

distance. Metric trees do not prohibit the use of fixed thresholds instead of proportional ones, but to recover more significant clones for larger fragments, it is more natural to specify the thresholds as a function of size.

As noted in [78], it is easy to build fragments for which fixed thresholds would result in false positive or false negative clones. For example, let fragment f be of size 10 000 and fragment f' be of size 13 000. Assume f and f' have identical first 10 000 tokens, but at the end of f' there is an appendage of 3 000 tokens. If threshold ϵ was below 3 000, the algorithm would miss such a clone candidate. In practice, this case could be represented by f being a class and f' being the same class with new methods added at its end. However, $\epsilon = 3000$ would report pairs that are not clones. For example, let f and f' be of size 200 and $\delta_l(f, f') = 200$. Now, $\delta_l(f, f') \leq \epsilon$, f and f' would be reported as clones even though they probably share no similarities. Hence, the query's *radius* should be size-sensitive. This assertion will be verified in the experiments section.

```
1  manhattanCloneDetectionAlgorithm(N, distance)
2      tree = new MetricTree
3      forall f ∈ System :
4          tree.insert(f)
5      clones = ∅
6      forall f ∈ N
7          clones[f] = tree.rangeQuery(tree.root, f, distance)
8      return clones
```

Figure 4.1 Manhattan clone detection algorithm with absolute radius

```
1  manhattanCloneDetectionAlgorithm(N, distance)
2      tree = new MetricTree
3      forall f ∈ System :
4          tree.insert(f)
5      clones = ∅
6      forall f ∈ N
7          radius = distance * len(f)
8          clones[f] = tree.rangeQuery(tree.root, f, radius)
9      return clones
```

Figure 4.2 Manhattan clone detection algorithm with relative radius

4.5 Experiments and Results

The experimental setup uses a 2.16GHz Core2 Duo Intel processor with 4GB of RAM under Linux Fedora 13 and softwares were compiled with g++ 4.4.1. The analyzed system is presented in table 4.1. Tomcat is a web-service host produced by the Apache Foundation. Table 4.1 shows relevant system characteristics. System tokens were extracted using a homemade Java lexer. A significance cutoff of 100 tokens was applied to filter out small fragments. Prior to the experiments, the oracles which are composed of all pairs with Levenshtein distance less than a desired thresholds were computed and stored. The selected thresholds were 0.1, 0.2 and 0.3, being relative to the fragments length.

Table 4.1 System features

System	Tomcat
Version	5.5
LOC	200k
Av. Length of fragments in tokens	341.29
Max. Length of fragments in tokens	19999
Num. of methods with more than 100 tokens	2374

Quality assessment of the results was done using precision and recall. The former is measured with respect to the Levenshtein distance. A clone pair is accepted if the distance between its two fragments is less than the selected Levenshtein distance. The precision then becomes the ratio between all the accepted pairs over all the reported pairs. Results recall is also measured with respect to the Levenshtein distance. Precision being measured first, recall then becomes the ratio of all accepted pairs over the cardinality of the corresponding oracle set. The F-value is then computed with the resulting precision and recall values.

Figures 4.3 and 4.4 show the precision with respect to the recall curves for different Levenshtein threshold. The blue curves are for a Levenshtein threshold of 0.3, the green ones for 0.2 and the red ones for 0.1 . All curves for both window lengths exhibit stable precision for low recall, then a slight decrease for higher recall, then a steady decline in precision for recall value with precision lower than 0.8. This suggests that in need of a high-precision tool, it is still possible to achieve good recall. For example, requiring a precision of 0.9 for a Levenshtein query of 0.1 can still give recall higher than 0.8 using window length 2. These

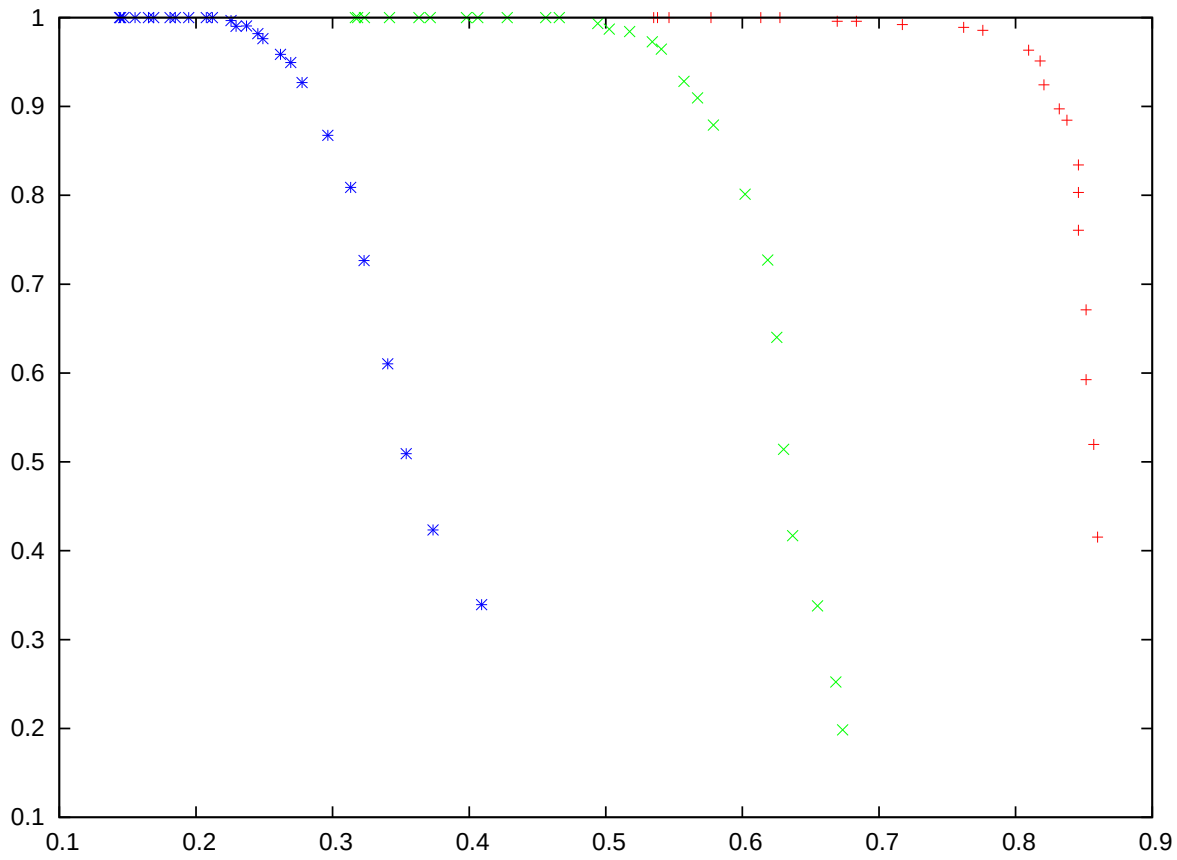


Figure 4.3 Precision with respect to recall for Tomcat and a window length of 1. Radii value on y-axis is relative. Blue curve is for Levenshtein 0.3, green curve, 0.2, and red curve 0.1

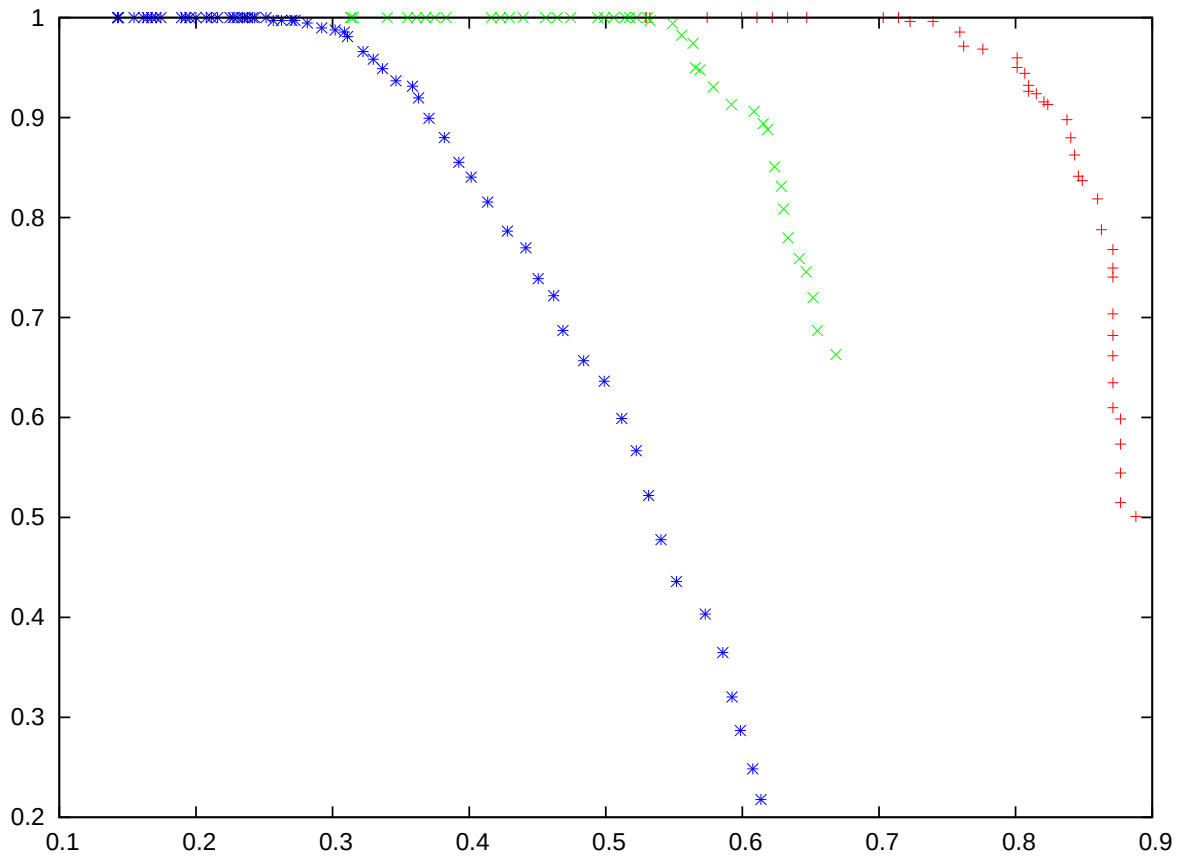


Figure 4.4 Precision with respect to recall for Tomcat and a window length of 2. Radii value on y-axis is relative. Blue curve is for Levenshtein 0.3, green curve, 0.2, and red curve 0.1

figures also indicate that higher window lengths increase recall for same precision values.

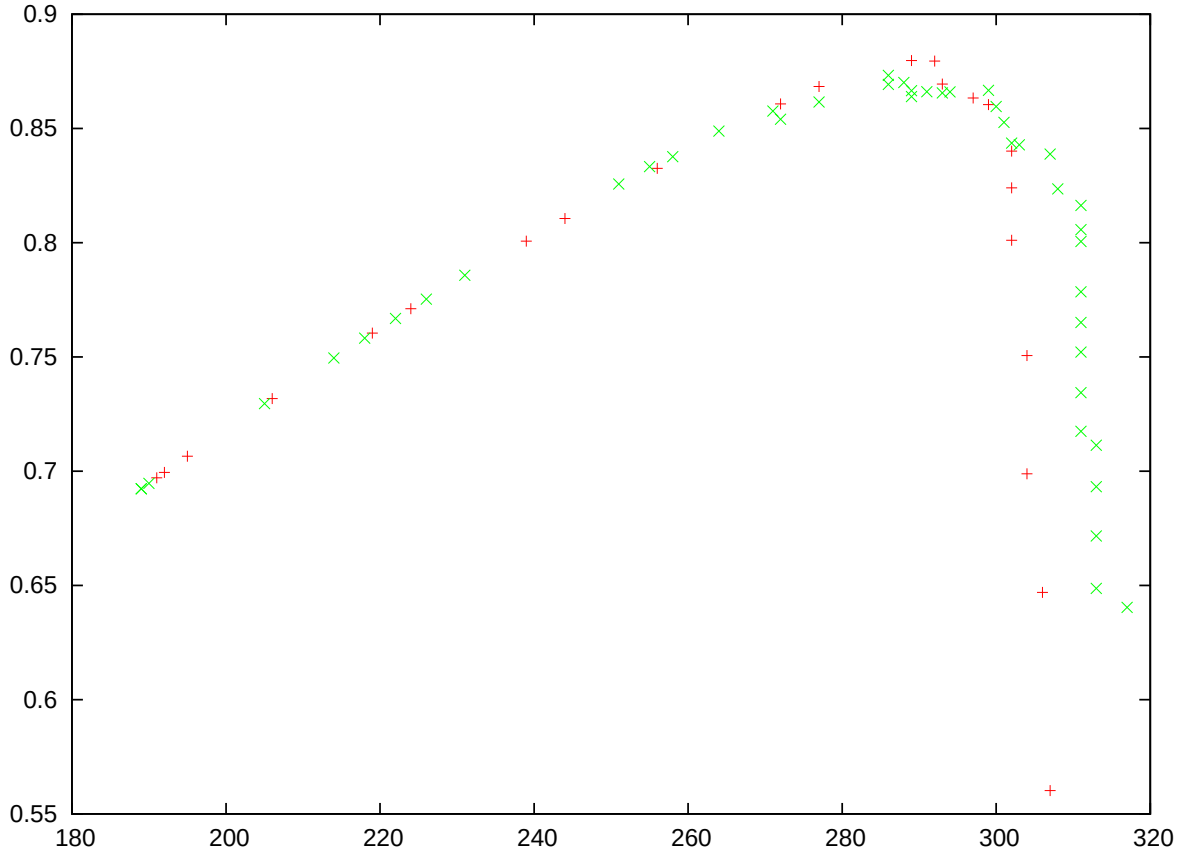


Figure 4.5 F-Value with respect to the total number of accepted clones for Tomcat with a Levenshtein threshold of 0.1. Red curve is for window length 1, green curve is for window length 2

The overall accuracy of the tool is depicted in figures 4.5, 4.6, and 4.7. These figures show the number of reported accepted pairs with respect to their corresponding F-Value. The red curve in each figure is the curve for a window length of 1 and the green one is for a window length of 2. Each figure shows the behavior for a different Levenshtein threshold. All curves exhibit a maximum before they all steadily decline. Figure 4.5 has the green curve decline a little bit farther after the peak value than the others. This might be a side effect of the window length at small Levenshtein threshold.

Programs were executed until a computed F-value was suspected to be maximum. This is why curves do not have points for every radii and some curves have more points than others. It is worth noting that on both absolute and relative radii, windows of length 2 induce a slower growth rate on the F-value. This leads to longer curves for this length. Tables 4.2 and

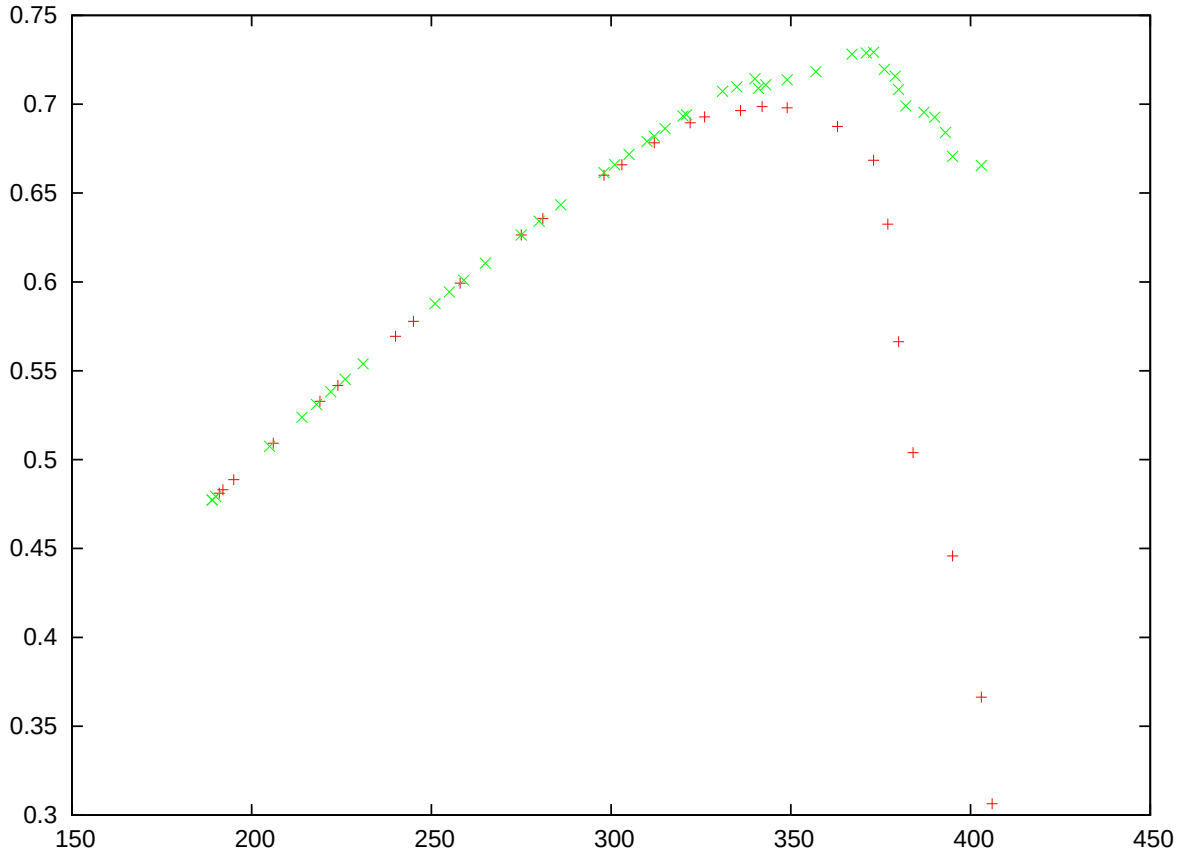


Figure 4.6 F-Value with respect to the total number of accepted clones for Tomcat with a Levenshtein threshold of 0.2. Red curve is for window length 1, green curve is for window length 2

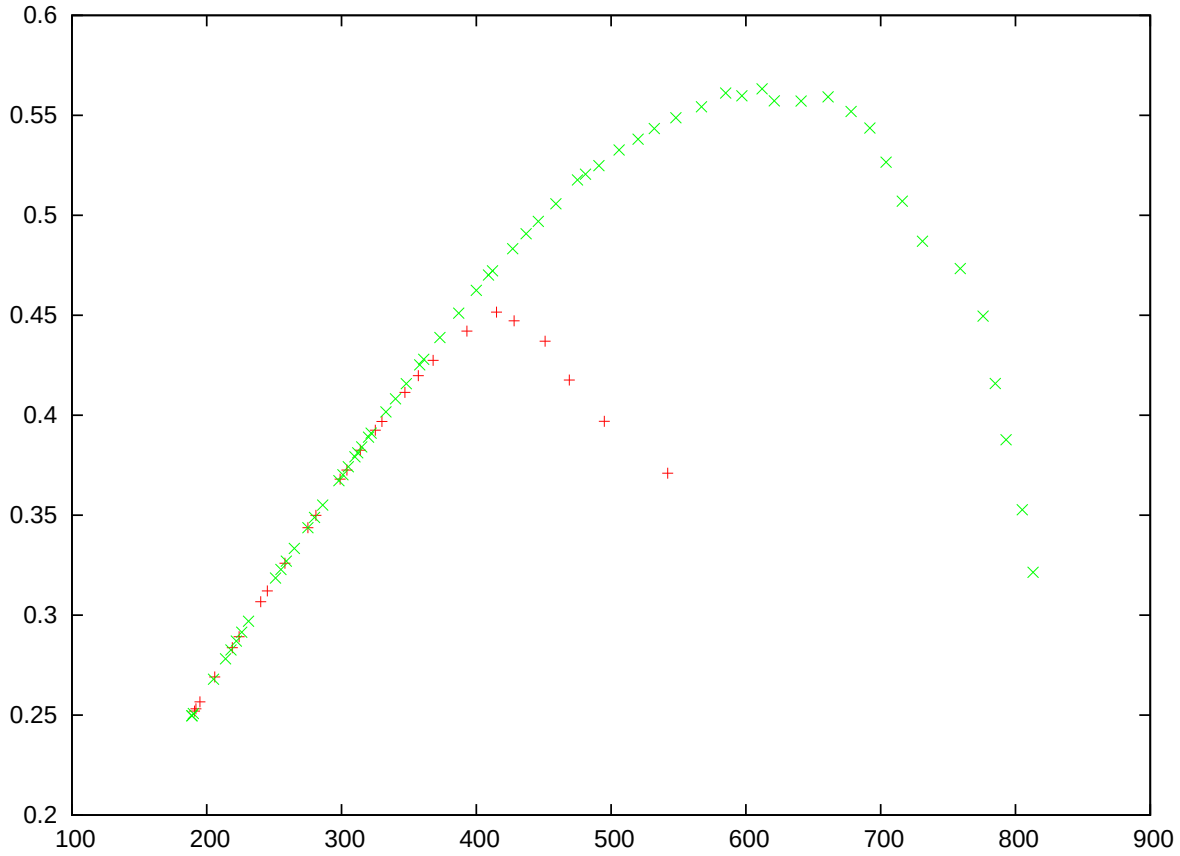


Figure 4.7 F-Value with respect to the total number of accepted clones for Tomcat with a Levenshtein threshold of 0.3. Red curve is for window length 1, green curve is for window length 2

4.3 show the triplet of optimal values under different configurations. Each row represents a different Levenshtein threshold and each column has values for different window lengths. The triplets' values are in the form (radii,f-value,number of clones). From these tables, numerous observations can be drawn. First, for Levenshtein distance greater than 0.0, the number of clones for the optimal triplet is always higher using the relative radius instead of the absolute one. Second, except for the relative radius with Levenshtein threshold 0.1, a window length of 2 gives better results. Third and last, the total number of found clones increases with the Levenshtein threshold indicating the method effectively finds clones with higher Levenshtein distance. Table 4.4 summarizes the best achievable F-Value for the different parameters. Explicit value of the precision and recall for each case is given.

Figure 4.8 displays the variation of execution time with respect to the Manhattan distance range-query radius. The red curve is associated with a window length of 1 and the green one a window length of 2. The execution time increases monotonically as the radius increases. Execution time is also greater for a greater window length.

4.6 Discussion

Results support the possibility of achieving a good approximation of the Levenshtein distance using tokens frequency vectors and the Manhattan distance. Indeed, for a Levenshtein threshold of 0.1 an F-value close to 0.88 is achieved with a very high precision. It is worth noting that at equal precision, the algorithm running does find more clones for threshold 0.2 than for threshold 0.1. Thus, if one is ready to accept a lower recall, the technique still finds clones within the region between 0.1 and 0.2 with high precision and hence could gives interesting candidates to analyze. The same observation can be applied to the 0.3 threshold. Overall, the technique has a higher precision than recall. This can be acceptable for a clone detection tool, but not as an oracle producer. However, for quick oracle production at low Levenshtein thresholds, the technique provides a reliable approximation at a fraction of the

Table 4.2 Tomcat optimal F-value for fixed Levenshtein distance and window length. Values in table are formatted as (radius,F-value,clones number). Radii are absolute values.

	Window Length	
Levenshtein	1	2
0.0	(0.0,0.992,188)	(0.0,0.997,188)
0.1	(8.0,0.844,267)	(14.0,0.851,276)
0.2	(11.0,0.655,302)	(25.0,0.685,345)
0.3	(15,0.397,399)	(37.0, 0.524, 573)

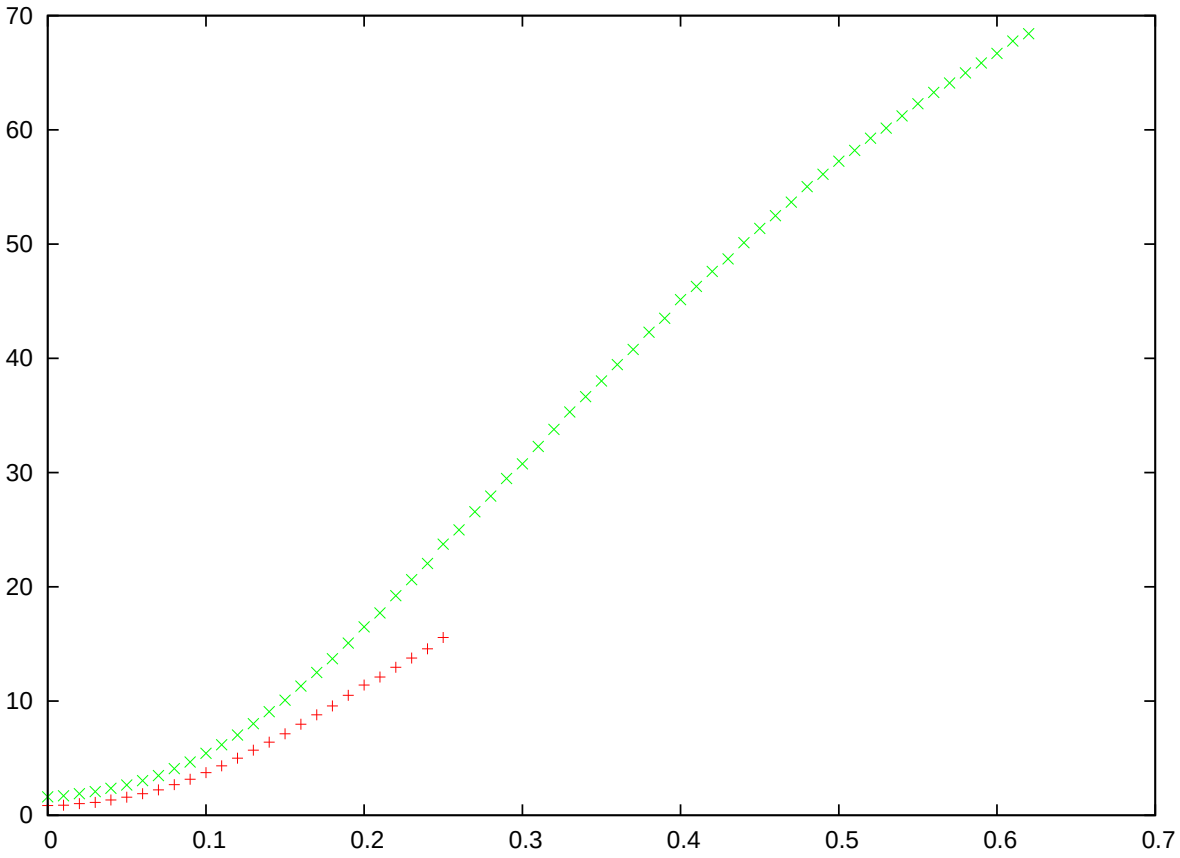


Figure 4.8 Total execution time with respect to query radius in Tomcat. Red curve is for window length 1, green curve is for window length 2

Table 4.3 Tomcat optimal F-value for fixed Levenshtein distance and window length. Values in table are formatted as (radius,F-value,clones number). Radii are relative values.

	Window Length	
0.0	(0.0,0.992,188)	(0.0,0.997,188)
0.1	(0.11,0.879,289)	(0.16,0.873,286)
0.2	(0.17,0.699,342)	(0.33,0.729,373)
0.3	(0.2,0.452,415)	(0.48,0.563, 612)

Table 4.4 Tomcat precision and recall values for best achievable F-value

Levenshtein	F-Value	Precision	Recall
0.0	0.997	0.997	1.0
0.1	0.879	0.963	0.810
0.2	0.729	0.888	0.619
0.3	0.563	0.722	0.462

time cost.

Regarding execution time, in every configuration it increases along the increase of the absolute or relative radii. This is explained by the nature of the metric tree: as the radius increases, less of the tree can be pruned at each step resulting in more comparisons executed for each range-query, the worst-case being the all-pairs comparison. However, even with high radii, the execution time is still under a minute which is more than 50 times faster than computing the exact Levenshtein set (also using a metric tree) as shown in table 4.5. Results accuracy is at worst half the target set. This suggests that the accuracy trade-off with the enhanced speed is appealing even in more extreme cases.

Of course, the Levenshtein significance thresholds are not universal. The choice of 0.3 as

Table 4.5 Execution time of the Levenshtein oracle compared to the configuration for best results with Manhattan

Levenshtein	Levenshtein (s.)	Manhattan (s.)
0.0	202	1.00
0.1	1662	4.35
0.2	2984	35.30
0.3	4080	55.03

an upper limit is motivated by the authors prior experience and the literature. The calibration parameters presented in section 4.5 are valid only for the presented thresholds. However, repeating these experiments with different thresholds would provide the configuration required to have a binary answer to the approximate Levenshtein query. The authors suspect the behavior to be similar albeit fall of the precision should occur earlier or later depending on the magnitude of the threshold.

Since it uses a metric tree, the technique is additively incremental. New fragments can be inserted in the tree as well as additional range-queries be performed without the need to rebuild it as can additional range-queries. Deletion requires rebuilding the entire sub-tree rooted at the deletion site. However, as shown by the execution time, building the tree is computationally inexpensive and deletion of some fragments shouldn't hinder performances. Being incremental makes the new algorithm well-suited for clone evolution analysis.

There are some threats to validity. Results are only preliminary since they come from one system only. Therefore, no definitive statistical properties may be inferred. However, the exposed perspective opens up a whole new way to develop clone detection tools.

4.7 Conclusion

A new clone detection technique based on metric trees and the Manhattan distance has been presented. Experimental evidences show the technique to be closely related to the Levenshtein distance for small differences. Regarding the three presented research questions, the results gave interesting positive answer. It is indeed possible to give low-distortion answers to Levenshtein binary queries and the window length can improve the results overall quality. The expected limit where the technique stops emulating Levenshtein accurately seem to be between 0.1 and 0.2 .

Only preliminary results were presented, but they encourage to pursue further experiments using this technique. Some of these could include an assessment of window lengths higher than 2 as well as significance cutoff value higher than 100. Experimental evidences of the rationale supporting Manhattan distance being better than the other metrics of the l_i family could also be gathered. Software clone evolution analysis as well as quick oracling are also some of the tool applications that could be investigated.

Acknowledgments

This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program and the Fonds Quebecois Nature et Technologie (FQRNT).

CHAPTER 5

ERRATA AND COMMENTS ON PAPER 1

At the time the paper in chapter 4 was published, a small mistake in an equation was missed by the author and the reviewers. Incidently, the same mistake was repeated in chapter 12, 4 months later and wasn't caught again at that time. In fact, it took nearly four years before the small incoherence in the equation was finally found. Although the correction in the model is not expected to affect any of the experimental results (because it does not affect the approximation at all), it is still in order to present the errata in this thesis.

The equation:

$$\frac{t!}{(t-l)!} \tag{5.1}$$

where t is the total number of token types and l the size of the window was written under the hypothesis that every token type is unique in a tuple of length l, or l-gram. While this is true for $l = 1$, and mostly true for $l = 2$, it is not strictly correct and there exists some infrequent sequences of tokens of small lengths that have duplicated token types. This becomes more visible even more true as l grows. The proper number of all different l-gram is:

$$t^l \tag{5.2}$$

which, for $t = 200$ is 200, 40,000, and 8,000,000 for $l = 1, 2, 3$ respectively. Since no part of the experiment relied on this result, it does not affect the general conclusions. However, it would be correct to note that the proportion between the observed number of different l-grams and the maximum possible number is actually smaller than what is reported and observed in the paper. Thus, in practice, the result is slightly better than what is interpreted in the original paper. Of course, a clever reader has probably already noticed that in all cases, it is impossible for the number of different l-gram to be larger than the size of the analyzed system, as it would imply that the system has more l-grams than the total number of tokens it contains, which is a paradox.

Incidently, the figures showing the examples of n-grams actually contain a contradiction with the original equation, because the figure contains an l-gram with a duplicated token. The figure can stay as it is, as the mistake was indeed in the equation. Also note that in the text, the equation is referred to as "exponential", while the actual equation shown is of

a combinatoric nature and not of a true exponential nature. The new equation is, on the contrary, exponential, and so the text did report the relation correctly.

While the curves of the F-values clearly show a local maximum, we could not prove this maximum to be the global maximum even though the shape of the curves suggested they were the maximum. However, since the number of identified clones always has a monotonic growth with respect to the radius, the subset containing false-positives will keep increasing while the subset containing true-positives will reach saturatation. Hence, the increasing ratio of false-positives will strongly affect the future F-values, making it highly unlikely that, after the sharp decline following the observed maximum, the F-values will be greater than our suspected maximum F-values.

The paper uses some basic properties of square roots to prove equation 4.10. Precisely, the inequality

$$\sqrt{x_1} + \sqrt{x_2} \geq \sqrt{x_1 + x_2}$$

holds because elevating both members to their square gives

$$x_1 + 2\sqrt{x_1}\sqrt{x_2} + x_2 \geq x_1 + x_2$$

and the left member is clearly greater than the right member, with equality holding if x_1 and x_2 are equal to 0. By induction, it can be proven that this holds for an arbitrary number of x_i and the truth of equation 4.10 follows.

CHAPTER 6

PAPER 2: ABOUT METRICS FOR CLONE DETECTION

6.1 Introduction

Clone detectors rely on the concept of similarity and distance measures to identify cloned fragments. The choice of a specific distance function in a clone detector is arbitrary up to some extent. However, with a deeper knowledge of similarity measures, we can condition this choice to have some properties that can help improve scalability and quality of tools. This paper presents some interesting results, insights and questions about similarity and distance measures, including a somehow counter-intuitive result on the cosine distance.

For a comprehensive survey of many distances and metrics, the reader is invited to read [106].

This paper covers the following topics:

- A link between the Jaccard measure on sets and the Manhattan distance in euclidean space
- Limitations of the cosine distance on normalized vectors and approaches to overcome them
- Unanswered questions on some similarity detection techniques

As a convention in this paper, we set $\mathbb{R}_+^n = [0, \infty)^n$.

6.2 An Equivalence of the Jaccard Metric and the Manhattan Distance

Sometimes it is useful to link two known metrics to get a desired behavior or a better understanding of one of the two. In our case, we link the Jaccard metric on sets with the Manhattan distance on the space $\mathbb{R}_+^n$. Our equivalence holds for arbitrary sets, but different representations in the space $\mathbb{R}_+^n$ may be possible, which leads to different applications. We first prove the result, then we will explain how the result may be used.

Recall the Jaccard measure on two finite sets U, V is defined as:

$$\mathcal{J}(U,V) = \frac{|U \bigcap V|}{|U \bigcup V|}$$

and from this we define the Jaccard metric as:

$$\delta(U,V) = 1 - \frac{|U \bigcap V|}{|U \bigcup V|} = \frac{|U \bigcup V| - |U \bigcap V|}{|U \bigcup V|}$$

which is known to satisfy all the metric axioms (see section 3 for a brief recall). The Manhattan distance, noted l_1, on two vectors[1] u, v from $\mathbb{R}_+^n$ is:

$$l_1(u,v) = \sum_{i=1}^{n} |u_i - v_i|$$

Now, lets associate a unique integer $1 \leq i \leq |U \bigcup V|$ to every element μ in U and V. Now, choose $n = |U \bigcup V|$ and take $u, v \in \mathbb{R}_+^n$ such as $u_i = 1 \leftrightarrow \mu_i \in U$ otherwise $u_i = 0$, and $v_i = 1 \leftrightarrow \mu_i \in V$ otherwise $v_i = 0$. From this, we draw:

$$|U \bigcup V| = \sum_{i=1}^{n} max(u_i, v_i)$$
$$|U \bigcap V| = \sum_{i=1}^{n} min(u_i, v_i)$$

The min and max terms in the preceding equations are only equal if $u_i = v_i$, otherwise one of the two is u_i and the other is v_i and because $|u_i - v_i| = |v_i - u_i|$ we must have:

$$\begin{aligned} |U \bigcup V| - |U \bigcap V| &= |\sum_{i=1}^{n} max(u_i, v_i) - \sum_{i=1}^{n} min(u_i, v_i)| \\ &= \sum_{i=1}^{n} |u_i - v_i| \end{aligned}$$

[1] Actually, to define the Manhattan distance , we do not need the full power of a vector space but only requires the n-tuples in $\mathbb{R}_+^n$. However, it is now a custom to name everything in $\mathbb{R}_+^n$ a vector and we shall follow the custom.

Replacing the last two equalities in the original Jaccard metric leads to:

$$\delta(U,V) = \frac{|U \bigcup V| - |U \bigcap V|}{|U \bigcup V|} = \frac{\sum_{i=1}^{n} |u_i - v_i|}{\sum_{i=1}^{n} max(u_i, v_i)}$$

with the numerator of the last term being the Manhattan distance between u and v. This proves the existence of an equivalence between a normalization of the Manhattan distance between certain vectors and the Jaccard similarity between sets. It is easy to generalize this result to allow any positive integer for u_i and v_i instead of $0, 1$. We simply need to project multiple elements of the sets U and V onto a single coordinate i. The proof is then almost identical to the one presented here.

Why is this result interesting ? In practice, clone detectors use a lot of similarity and distance measures on sets. Most of them are not metrics (like the Dice coefficient, the Tanimoto distance, etc.) and thus have a behavior less understood. Moreover, metrics lead to known opportunity of optimizations in search spaces that arbitrary distances do not offer [28]. Thus, even if the choice is ultimately arbitrary, there exist some arguments that favor metrics over arbitrary distances and knowing the link between some of them can help making a better choice. In this case, a simple distance between vectors gives us a useful interpretation as a distance between sets.

6.3 Properties of Some Angular Distances

We start by proving a result on the sine function.

Theorem 6.3.1. *Let X be a subset of $\mathbb{R}_+^n | x \in X \rightarrow ||x|| = 1$. Let $\theta_{x,y}$ be the angle between any two x and $y \in X$. Finally, let $\delta : \mathbb{R}_+^n \times \mathbb{R}_+^n \longrightarrow \mathbb{R}$ be defined as $\delta(x,y) = \sin \theta_{x,y}$. Then, δ is a metric on X.*

Proof. We need to prove that δ satisfies the four properties of a metric.

(Non-negativity) $\delta(x,y) \geq 0$. This is true, since the all x, y have positive coordinates and the angle between such vectors must be in $[0, \frac{\pi}{2}]$.

(Nullity) $\delta(x,y) = 0 \leftrightarrow x = y$. This is true, since the sine of an angle restricted to $[0, \frac{\pi}{2}]$ is 0 if and only if that angle is 0, and the angle between x and y is 0 if and only if $x = y$.

(Symmetry) $\delta(x,y) = \delta(y,x)$. This is true since the angle between x and y equals the angle between y and x.

(Triangle inequality) $\delta(x,y) + \delta(y,z) \geq \delta(x,z)$. The property holds, but it is tricky to prove. First, observe that if the angle between x and y or the angle between y and z is greater

than the angle between x and z, then the property must hold since the sine is monotonically increasing in $[0, \frac{\pi}{2}]$. It remains to prove that it holds if the bigger angle is between x and z.

Now, clearly the sum of the angles $\theta_{x,y}$ and $\theta_{y,z}$ is greater or equal than $\theta_{x,z}$. Because sine is monotonically increasing in the considered interval, we now have:

$$\sin(\theta_{x,y}) + \sin(\theta_{y,z}) \geq \sin(\theta_{x,y} + \theta_{y,z}) \geq \sin(\theta_{x,z})$$

We develop the first two members of this inequality:

$$\sin(\theta_{x,y}) + \sin(\theta_{y,z}) \geq \sin(\theta_{x,y} + \theta_{y,z}) = \sin(\theta_{x,y})\cos(\theta_{y,z}) + \sin(\theta_{y,z})\cos(\theta_{x,y})$$

Subtracting the right member to the left leaves:

$$\sin(\theta_{x,y}) - \sin(\theta_{x,y})\cos(\theta_{y,z}) + \sin(\theta_{y,z}) - \sin(\theta_{y,z})\cos(\theta_{x,y}) \geq 0$$

Because the cosine of these angles lies in $[0, 1]$, we have

$$\sin(\theta_{x,y}) - \sin(\theta_{x,y})\cos(\theta_{y,z}) \geq 0$$
$$\sin(\theta_{y,z}) - \sin(\theta_{y,z})\cos(\theta_{x,y}) \geq 0$$

and we conclude that the sum of the two must be greater than or equal to 0. The inequality between the first two members of equation 1 holds, and because of our definition of the second member, the inequality between the last two members already held. Thus, we conclude that:

$$\sin(\theta_{x,y}) + \sin(\theta_{y,z}) \geq \sin(\theta_{x,z})$$

and the triangle inequality is satisfied.

All four properties hold and we have secured the theorem. □

Why is this result interesting ? First, even if the cosine distance is a very popular similarity measure on vectors, it is not a metric. To preserve the non-linearity of the cosine function, this results actually states that you need its dual function, the sine, to get a metric, under

certain restriction. The sine is not as straightforward to compute as the cosine on vectors, but it has the advantage of being a metric on the first quadrant of $\mathbb{R}^n$.

It is also worth questioning whether or not the non-linearity of the sine and cosine is a desirable property. The angle distance between two vectors is a metric (this fact is used in the above proof) and is linear on the arc distance between the vectors on a unit circle. It would be interesting to verify whether or not the cosine is actually better than the sine and the angular distance considering our arguments. In general, it would be safe to compare a measure with closely related one to assess which is better even if it means a small additional computational cost.

6.4 Further Questions and Research: Implicit Distances, Why Should we Recover them ?

The previous sections dealt with special cases of distances that can be easily converted to a metric in order to use all the properties and knowledge we can draw from the vast literature on the subject. All these observations are interesting to ponder in the context of clone detectors explicitly based on a similarity or a distance measure. However, many tools only use implicit distances that are not properly defined as a mapping between a pair or a cluster of objects onto the real line $\mathbb{R}$. What should be the course of action for these tools ?

As hard as it can be, it should be possible in practice to recover the distance function. The mapping might be hard to express, or too many parameters may interact together to produce a long and complex formula, but these are only additional reasons to support the need to formalize implicit or hidden distance functions. Complete understanding of clone detection technology is intertwined with our ability to encapsulate their behavior in mathematical formula: if this task proves to be tedious, what can we say of understanding their behavior or how can we even completely compare them ?

The following questions are a starting point to help investigate the mathematical foundation of clone detection tools:

- What distance function does the clone detector uses ?
- How many parameters does the distance depends upon ? Are some redundant ? Are some useless ?
- If no mathematical formula seems possible to exist to encapsulate the distance used, why is it so ?
- What key features does the tool use ? Are there other tools using those features ? Do those tools have a well-formulated distance ?

- Is there an already existing distance that approximates the tool's implicit function ? How good is this approximation ?

Nevertheless, unanswered questions do not hinder performances and tools built around implicit distances can produce good results. To shed a deeper light on why they do have good results might however help the general understanding of clone detectors' behavior.

Acknowledgement

The authors wish to thank Theotime Menguy for his quick review of the proof in section 3. They would also like to thank Mathieu Merineau for providing the excellent reference [106].

CHAPTER 7

PAPER 3: PERFORMANCE IMPACT OF LAZY DELETION IN METRIC TREES FOR INCREMENTAL CLONE ANALYSIS

7.1 Abstract

Doing clone detection in multiple versions of a software can be expensive. Incremental clone detection is acknowledged to be a good method to make this process better. We extend existing ideas in incremental clone detection to metric trees using lazy deletion. We measured the execution time of the non-incremental and the incremental version of the clone detector and discovered that incremental clone detection can save a sizable amount of time even for versions separated by large variations. We discuss the results and propose some future research.

7.2 Introduction

Clone detection is usually applied on many consecutive versions, releases or revisions of a system because change over time is of great interest. While the analysis can always be done from scratch on all target versions, it is clear that we expect a lot of the work done to be redudant because we expect different consecutive versions to share some of their code. This observation is even more likely to be true on small revisions based on daily commit, but it should resaonnaly hold even for larger versions spanned across years. Thus, because multi-version clone analysis is expensive, it is natural to try to figure out a way to use the redundancy to accelerate the process. This process is known as incremental clone detection and different techniques already support it.

This paper will focus on showing how to make metric tree clone detection incremental using lazy deletion in the tree. The main reason to use lazy deletion is because true deletion is too expensive in a metric tree, with a linear worst case per operation. We will then compare the non-incremental and the incremental versions of the technique on a large corpus for different clone similarity thresholds in order to measure the expected speed-up. We chose to use different versions of a project instead of different revisions because we wanted to assess the performance on large deltas, suspected to give worst performance. Thus, if with large delta the acceleration is large, it will follow that with smaller delta the performance will even be better.

The rest of the paper is divided as follows: Section 7.3 explains the algorithmic modifications to metree, the experimental methodology as well as some underlying hypotheses of our work, Section 7.4 presents the execution time, non-incremental and incremental, for many versions of Firefox with different thresholds, Section 7.5 discusses and interprets the results, Section 7.6 introduces relevant related works, and Section 7.7 offers some conclusions of our work along with further research possibilities.

7.3 Methodology

The original metric tree clone detection was described in [79]. We will only briefly summarize the technique and then explain how it was modified to support lazy deletion.

Clone detection with metric trees first parses a system to extract some functions. Then, it computes a representation of the function that will be consistent with the desired metric (Manhattan, Jaccard, Levenshtein, etc.) to compute the similarity between functions. The set of the representation of all the functions is then use to build a metric tree, a space partitioning data structure, that indexes all fragments in order to speed-up space queries like range-queries and nearest-neighbour-queries. Once built, the metric tree is queried with all the elements we wish to know their clones. These elements may be the same as the ones used to build the tree, but they may be completely different.

Building the metric tree is worst-case quadratic, but in practice it is very fast even for hundred of thousands of fragments. The queries are much slower and depend on the desired similarity. They are very fast at 100% similarity, but are already much slower at 90%, and becomes untractable at around 40%. This is because the speed of the queries indirectly depends on the size of the returned set, and this set quickly increase in size with the increase of the number of tolerated differences between the elements. Thus, while building a new tree is very fast, repeating all the queries is inherently slow. Between two versions of a software, however, it is expected that a high number of queries are actually redundant and are not necessary to repeat.

The exact steps of what one should do between two versions are:

- Identify which functions from version N+1 are new
- Identify which functions from version N were deleted
- Mark all deleted functions from version N as such in the metric tree
- Do a query on all new functions of version N+1 in the marked tree

- Remove all deleted functions from version N in the clones of version N, because they no longer exist in versions N+1

Identifying with very high accuracy what methods were added and deleted between two versions is in itself a tedious task. However, the following simple heuristic can be used. For all fragments, associate it with the tuple composed of the path of its file along with its first and last line. If such a tuple exists in both versions and the fragment is identical in both, then we can assume its the same. This heuristic cannot erroneously report a fragment as deleted if it is still there, but it can identify persisting fragments as deleted; the heuristic has no false-negative (it cannot miss a deleted fragment), but it has false-positives (it can report more deleted fragments than necessary). Therefore, it may lead to unnecessary computations. However, it is expected to be reasonably precise and very fast. In the end, non-associated tuples will lead to a deletion in version N and a new fragment in version N+1.

The lazy deletion is straightforward to implement. The metric tree is augmented with a hash table and every time an element is deleted, it is added to the hash table. Then, when a query is processed, if an element would be added to the result, the tree checks if that element is in the hash table, in which case the element is not added to the result. Alternatively, the table could have been kept outside the tree, and the results could have been post-processed to remove the elements from the list of deleted functions. Because those operations are simple, we expect that the reduced number of queries will give a sizable acceleration in the incremental version. The main mitigation factor is the increasing size of the tree, because the elements are still nodes in the tree despite being deleted. Because the speed of the queries depend on the size of the tree, we expect that even though all queries will be slower than the one in the old smaller tree, the number of queries will be sufficiently small to improve the performance. In the experiment, we will try to assess the maximum tree growth on which the performance starts to decrease and would force a whole tree reconstruction.

Because the incremental version of the algorithm does not change the kernel of the matching procedure, the quality of the results are not affected. They will have the same precision and recall as before and all other relevant properties still hold.

Using this algorithm, we conducted the experiments in the next section.

7.4 Results

The experimental setup uses a 3.40GHz i7 Intel processor with 16GB of RAM under CentOS 6.2 and software were compiled with g++ 4.8.1. The corpus is composed of 17 versions of Firefox and around 80MLOC. The relevant details on the corpus are presented in Table 7.1. The last column tells the number of methods processed in the incremental version of the tool.

Table 7.1 Details of Firefox 17 versions. Methods of interest are only those of size greater than 100 tokens.

Version #	Release date	Elapsed time since last release (days)	MLOCs	Methods of interest	Δ Methods of interest
2.0	2006-10-23	682	4.13	27,254	0
3.0	2008-06-19	605	3.78	24,480	20,473
4.0	2011-03-22	1,006	4.75	31,574	29,558
5.0	2011-06-21	91	4.71	31,438	10,398
6.0	2011-08-31	71	4.67	31,239	10,878
7.0	2011-09-27	27	4.66	31,308	10,844
8.0	2011-12-18	82	4.64	31,319	9,452
9.0	2011-12-20	2	4.70	30,627	10,422
10.0	2012-01-31	42	4.94	32,613	7,364
11.0	2012-03-13	42	4.96	32,478	18,323
12.0	2012-04-24	42	5.00	32,628	11,291
13.0	2012-06-05	42	5.04	32,881	10,472
14.0	2012-07-17	42	5.13	33,605	10,629
15.0	2012-08-28	42	5.05	34,476	22,982
16.0	2012-10-09	42	5.71	39,247	15,980
17.0	2012-11-20	42	5.79	39,785	11,938
18.0	2013-01-08	42*	6.27	42,991	25,772

In Table 7.2, the execution time of the metric tree is reported for both the non-incremental and the incremental version for all versions and all relevant incremental versions. These results are valid for a similarity threshold of 100%. The acceleration factor varies from 1.00 to 5.05, with an average of 2.43. While it is noticeable, it may seem modest, but this is largely due to the fact that queries at this threshold are inherently fast, and thus the size of the tree, unless enormous, doesn't have a big impact on the total time of the queries. Still, for many versions the acceleration factor is above 3 and this represents an interesting gain. These results would suggest that the incremental version would fare even better with slower queries of higher thresholds.

Table 7.3, the execution time of the metric tree is reported for both the non-incremental and the incremental version for all versions and all relevant incremental versions. These results are valid for a similarity threshold of 90%. As predicted earlier, we observe many accelerations as high as the one observed at 100%. However, it is not always the case. It does seem, however, that the average of 1.97 is not significantly different that the one of 2.43, and thus it seems that the expected acceleration is relatively independent of the desired similarity of the results for high thresholds. This effect however is expected to increase drastically at lower thresholds like 70%.

From a glance at the last columns of both Table 7.2, 7.3 and Table 7.1, we can observe there is a slight correlation between the lowest deltas and the largest acceleration factors, although the correlation does not hold in some cases. This may be explained by the fact that the tree structure may vary greatly between versions and that some trees are more degenerated than others and so some versions just exhibit performances out of their inherent structure. Also, execution times account for the time of writing the results on disk, and in some cases these may induce unwanted fluctuations in the measured time.

While extrapolation might be slightly inaccurate with such limited data, it is safe to assume that incremental clone detection would still be faster at higher thresholds as long as the increment in the number of methods remains small. In particular, when the increment is almost as large as the original method set, we can see a decrease in performance that is marked at higher thresholds. This is shown for version 3.0 and 4.0 at threshold 90%.

Overall, the results should be appraised for their globality, and not their specificity on one version or on a specific size of their delta of functions.

7.5 Discussion

From the results in the previous section, incremental clone detection undoubtedly has a faster execution time than non-incremental one on consecutive version. It is also worth noting that

Table 7.2 Execution time of metric tree clone detection at distance 0.0 for non-incremental and incremental clone detection in Firefox versions

Version #	Non-Incremental Time (s.)	Incremental Time	Acceleration Factor (1.0x)
2.0	5.65	N/A	N/A
3.0	5.25	5.25	1.00
4.0	7.73	4.92	1.55
5.0	7.27	3.08	2.36
6.0	7.43	2.50	2.97
7.0	7.52	2.71	2.77
8.0	7.77	2.25	3.45
9.0	7.62	2.48	3.07
10.0	6.02	3.59	1.68
11.0	8.25	2.93	2.82
12.0	8.01	3.31	2.42
13.0	8.03	2.70	2.97
14.0	8.94	1.77	5.05
15.0	8.16	7.39	1.10
16.0	9.36	2.70	3.47
17.0	10.62	2.87	3.26
18.0	13.32	8.02	1.34

Table 7.3 Execution time of metric tree clone detection at distance 0.1 for non-incremental and incremental clone detection in Firefox versions

Version #	Non-Incremental Time (s.)	Incremental Time	Acceleration Factor (1.0x)
2.0	492.31	N/A	N/A
3.0	451.01	847.48	0.53
4.0	779.18	1078.56	0.72
5.0	830.33	417.12	2.36
6.0	680.95	649.12	1.05
7.0	704.42	551.93	1.28
8.0	711.59	519.78	1.37
9.0	763.89	537.34	1.42
10.0	550.24	202.20	2.72
11.0	912.0	649.07	1.41
12.0	901.62	359.75	2.51
13.0	944.52	226.80	4.16
14.0	928.52	340.22	2.73
15.0	1066.08	809.91	1.32
16.0	1315.68	560.49	2.35
17.0	1276.97	439.37	2.91
18.0	1628.96	1031.70	1.58

the difference between versions can be as high as 50% for the incremental version to be faster. While those occurrences are rare, they exhibit a very good limit on which we expect the incremental analysis to be faster. Specifically, with the results obtained on versions that had at least 20% of variations in their method set, it is safe to assume that on a smaller granularity, such as a daily commit, should produce orders of magnitude higher acceleration factors. Also, because the points to compare in time would be more dense, the observed delta would be smaller because the heuristic used to identify deleted methods is expected to be more precise on points closer in time. This would lead to less total work done on two distant points by using the intermediate point to identify the delta in the method sets.

The following remark provides some thoughts on the necessity of developing solutions such as incremental clone detection. While for intensive clone detection on very large set of versions the incremental technique provides a good acceleration, it is worth noting that implementing the solution, testing it and configuring the other parts of the tool to support it takes some development time. Precisely, in this case, about 28 hours of work were required to deploy a version of the software with the incremental clone detector. Assuming from the results of this experiment that on average 5 minutes is saved on every experiment, it would take a little bit less than 350 experiments to compensate for the development time, and about 700 experiments to have a net gain of 28 hours. This paper had 17 incremental experiments. While we could think of papers with much more large delta experiments (small deltas would require thousands more experiments), it is safe to assume that it would take between 20 and 25 papers before the authors have a net gain in time equal to the time needed to develop the software for this paper. While this remark is a little bit humoristic in nature, it does provide on interesting thought: many attempts to speed-up software are only worth doing if it is expected that the tuned-up part will be solicited enough. Also, despite giving a good acceleration, the tool would on average be able to analyze 3 times more versions than before in the same time. If however the rate of release of software versions keeps increasing, then even a factor 3 acceleration will start to be barely noticeable at some point.

In the end, benchmarking performance is scientifically important because it helps validate techniques that are suspected to good in practice, like lazy deletion. But from a software development point of view, sometimes the time invested in such development might not be worth your while even for acceleration factors of 5.

7.6 Related Works

Clone detection state of the art includes different techniques. For type-1 and type-2 clones, AST-based detection has been introduced in [21]. Other detection methods for type-1 and

type-2 clones include metrics-based clone detection as in [86], suffix tree-based clone detection as in [45], and string matching clone detection as in [38]. For a detailed survey of clone detection techniques, a good portrait is provided in [98].

It is worth noting that [45] is another well known paper on incremental clone detection and that most of the techniques mentioned above could support incremental clone detection with few development hours, like our technique. However, this paper is one the first to investigate direct performance impact of incrementality on large deltas. Many experiments usually focus on commit-based granularity, but analyzing software based on releases give more insight on the variations between production versions rather than work in progress. Dealing with large deltas is also harder than with smaller deltas.

7.7 Conclusion

In this paper, we presented a variation of the metric tree to allow incremental clone detection using lazy deletion. The results show acceleration factors around 2.5 on average independent of the similarity of the clones. The results also suggested that the incremental clone detection could even be faster on smaller delta versions. Further research includes investigating smaller deltas and trying to improve true deletion in metric trees.

Acknowledgment

This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 8

COMMENTS ON PAPER 3

This chapter mentions the running time of the metric tree based clone detection is influenced by the radius of the queries. While no results are presented in this thesis to quantify the exact effect, understanding that increasing the radius will naturally lead to a larger area of the tree being visited is enough to be convinced of the existence of the effect.

Nevertheless, some experience with the metric tree has shown that for a radius of 0.0, fewer than 10 nodes can be sufficient to find the correct results. With a normalized radius of 0.1, the number of nodes will sharply increase to a few hundred. This leads to a relative increase of the running time that is significant. However, in absolute time, this increase is less than a second, making it very fast in practice. With a radius greater or equal to 0.4, most of the tree must be explored and the running time starts to exhibit the quadratic behavior characteristic of this type of problem. In general, the metric tree is an interesting data structure for spatial queries if the radius is reasonably small.

CHAPTER 9

PAPER 4: HOW MUCH REALLY CHANGES? A CASE STUDY OF FIREFOX VERSION EVOLUTION USING A CLONE DETECTOR

9.1 Abstract

This paper focuses on the applicability of clone detectors for system evolution understanding. Specifically, it is a case study of Firefox for which the development release cycle changed from a slow release cycle to a fast release cycle two years ago. Since the transition of the release cycle, three times more versions of the software were deployed. To understand whether or not the changes between the newer versions are as significant as the changes in the older versions, we measured the similarity between consecutive versions. We analyzed 82MLOC of C/C++ code to compute the overall change distribution between all existing major versions of Firefox. The results indicate a significant decrease in the overall difference between many versions in the fast release cycle. We discuss the results and highlight how differently the versions have evolved in their respective release cycle. We also relate our results with other results assessing potential changes in the quality of Firefox. We conclude the paper by raising questions on the impact of a fast release cycle.

9.2 Introduction

Competition and incredibly quick changes in the software market encourage faster software releases, which leads to fast cycle development. Whether for needs of functionality, quality or marketing, the number of major version releases of software have increased dramatically in the last years. However, in many cases it is nebulous how and to what degree a software has changed since its last major release and, consequently, how these changes will have an impact on users and developers interaction.

To keep up with the competition, some software projects have chosen to adopt a fast development release cycle. This was the case of Firefox in April 2011 when Firefox 4.0, last release following the old development release cycle, was followed the same year by Firefox 5.0, the first release under the fast development release cycle. Since then, 13 more major versions were released as of February 2013, and at least one more will be released before the end of the first quarter of 2013. While a fast release cycle enables a faster appropriation of new technologies, it might also cause several issues, such as increased stress for third-

party developers to migrate quickly to the latest version or the risk of non-compatibility of extension modules.

While examining versions from two different release cycles, some questions seem legitimate. Are the changes between versions in a fast release cycle as significant as they were in a slow release cycle or are there, overall, fewer changes ? Are fast-cycle versions re-engineered and thoroughly extended in the same way than their older counterparts or are they collections of minor fixes paired with small technical improvements ? What are the possible consequences of fast development cycle on evolution patterns of a software ?

In order to answer some of these questions, we focus on studying the overall similarity distribution between versions of Firefox before and after the development cycle change. For this end, we use a clone detector to find the closest match of functions from one version in the preceding one. To the best of our knowledge, this is the first study concerning differences in version evolution between two different development release cycles using a similarity analysis.

The rest of the paper is divided as follows: section 9.3 explains the experimental methodology as well as some underlying hypotheses of our work, section 9.4 presents the similarity distribution results for all 18 versions of Firefox, section 9.5 discusses and interprets the results, section 9.6 introduces relevant related works, and section 9.7 offers some conclusions of our work along with further research possibilities.

9.3 Methodology

To understand the impact of the new development release cycle on Firefox, we compute the distribution of similarities between code fragments of one version in relation to its preceding one. In this section, we summarize the technique used for this purpose and provide all the relevant parameters.

The system chosen for this study is Firefox. All major versions from 1.0 to 18.0 were downloaded from the official Firefox source server [3]. The source code from Firefox is written in a mix of C and C++. As for a similarity study it is not required to compile the code, we use approximate parsing to extract the required information. The approximate parsing strategy is composed of two steps.

First, a home-made pre-processor simulator is run to interpret the C/C++ pre-processor directives as much as possible. No #include directives are resolved, but multiline definitions are normalized in single line definitions and some #if constructions are resolved. Of course, as it is an approximation, some of the resulting code could still be unparsable, for example if unmatching pairs of braces exist because of pre-processing directives.

Second, we use an island-driven C/C++ parser to parse the code and extract all the

methods and functions. The island-driven strategy eliminates the need of a complete grammar at the cost of finer syntactic details. Since methods and functions are high-level structures which do not require the analysis of expressions, the adopted strategy performs well in practice and extracts almost all functions and methods.

In practice, this strategy proves itself to be robust and fast and we achieve an average parsing success rate of 99.63% on the 82MLOC composing all Firefox versions.

The chosen clone detector is the same as the one presented in [79]. It is a syntactic clone detection computing a similarity distance using lexical features. Although some algorithmic details are omitted in the present paper, we introduce in the following the key concepts along with the major parameter choices to allow a proper experimental replication.

Using the above mentioned combination of pre-processor simulator and island-driven C/C++ parser, we extract the tokens of each method and function of interest. A method or a function is of interest if it contains at least 100 tokens as specified by the C/C++ languages. As both methods and functions can be of interest and the difference is mainly semantic and not syntactical, for the rest of this paper, we will refer to both of them as methods.

For each previously extracted method, we take its associated string of tokens and convert it to a string of integers where each integer corresponds to the type of the token at the same offset. Frequencies of that new string's 2-grams are then computed in a vector and associated to the initial method. Since the proposed experiment composes with different versions, an external file keeps track of the different 2-grams in all the versions in order to coherently associate the same 2-grams from different versions to the index in the frequency vector.

As a next step, we take two consecutive versions and build a metric from the frequency vector set of the older one. Thereafter, that metric tree is used to find the nearest-neighbor of all the frequency vectors from the latest version. The distance of all the vectors to their nearest-neighbor is reported back and constitutes the basis of our distribution of changes between the two versions. Albeit not being the ground truth of all the changes between versions, we believe this distribution to capture enough information to understand quantitatively how much of the software evolved and how significant were the applied changes.

The soundness of the presented distribution of changes relies on the following assumption: the nearest-neighbor, the closest resembling fragment, of a fragment is assumed to be its ancestor in the preceding version. In general, this might not be the case, but in most practical cases, this tends to be true as developers seldomly morph one function into another without the original still resembling the transformed one. Even if this might not be the case, it is likely that the closest matching function will still be very close to the original leading only to a small error in the reported distance. This distortion, however, is extremely hard to

measure, and we do not provide an empirical proof of this fact. We rely on our experience as software developers to state it.

Naturally, the used method only allows a quantitative measure of changes and not a qualitative one. Therefore, to augment our analysis, we will present other observations drawn from [63] in section 9.5. In that paper, the authors assess the possible change of quality in Firefox caused by the change in the development release cycle. We found two measures to be relevant for our analysis. First, the median time before failure before and after the change in the release cycle and, second, the percentage of code submissions that are bug fixes. Although code submissions are not strictly linked to a specific number of methods, we assume that the overall percentage of bug fixes in the submissions is closely related to the modifications in the existing code base and not related to new functionalities and thus, little increase of the total number of methods should be observed for that activity. We will also assume that in general, a bug fix induces much less change in a method than a re-engineering activity or the creation of a new functionality. As our goal is to measure how the fast release cycle influences the overall change patterns between versions and not to strictly classify the different changes, we will solely use this information to differentiate bug fixing from re-engineering activities.

9.4 Results

All the experimental results are summarized in two tables. Table 9.1 presents interesting information about the corpus to help interpret the experimental data whereas table 9.2 presents different change measures including the average difference between methods computed by our clone detector. Each table is separated into two sections by an horizontal line: above the line are the versions released in the old cycle and under the line are the versions released the fast release cycle.

We do not report the detailed execution times of the experiment. Roughly, on a per version basis, the parsing time is about 5 minutes and the similarity analysis varies between 30 and 45 minutes.

Table 9.1 Details of Firefox 18 versions. Methods of interest are only those of size greater than 100 tokens.

Version #	Release date	Elapsed time since last release (days)	MLOCs	Methods of interest
1.0	2004-12-10	N.D.	3.69	24,342
2.0	2006-10-23	682	4.13	27,254
3.0	2008-06-19	605	3.78	24,480
4.0	2011-03-22	1,006	4.75	31,574
5.0	2011-06-21	91	4.71	31,438
6.0	2011-08-31	71	4.67	31,239
7.0	2011-09-27	27	4.66	31,308
8.0	2011-12-18	82	4.64	31,319
9.0	2011-12-20	2	4.70	30,627
10.0	2012-01-31	42	4.94	32,613
11.0	2012-03-13	42	4.96	32,478
12.0	2012-04-24	42	5.00	32,628
13.0	2012-06-05	42	5.04	32,881
14.0*	2012-07-17	42	5.13	33,605
15.0	2012-08-28	42	5.05	34,476
16.0	2012-10-09	42	5.71	39,247
17.0	2012-11-20	42	5.79	39,785
18.0	2013-01-08	42*	6.27	42,991

The time that elapsed between each release as shown in table 9.1 clearly illustrates the change in the speed of the release cycle since version 5.0. The fast release cycle is based on a 6 weeks cycle (the same as its competitor Chrome). The release dates reported are not the official ones, but rather the last dates on which modifications occurred in the frozen source directory on the Firefox file server. Keeping this in mind, table 9.1 reveals that until version 10.0, the observed time span between versions is actually not a uniform 6 weeks (42 days) cycle. In our discussion, we take this into account to draw possible conclusions on how Firefox source code evolved during that time. From version 10.0 to 18.0, the release cycle length is exactly the one intended by the developers. It is worth noting that for version 18.0, the actual time elapsed is 49 days; however, since the dates include the Christmas holidays, we assume there was an intended inactivity of 7 days, which would lead to the consistent 42 days. We believe this assumption to be reasonable, although not officially verified.

Concerning version 14.0, the actual release was 14.0.1 as only beta versions of 14.0 were actually released. Consequently, we decided to take version 14.0.1, but we still tag it as 14.0 because in practice we suspect the difference between the last beta of 14.0 and the actual 14.0.1 release to be negligible.

In table 9.2, only the methods of interest as defined in section 9.3 are considered. Since the average volume change between versions is not consistent between versions under both release cycles, we report the average difference of the change distribution and the average difference of the change distribution without identical methods. Doing so allows us to capture the changes restricted only to the parts that actually changed in order to compare the different change patterns between versions. In the table, the acronym NZ stands for Non-Zero which means non-identical.

In the old development release cycle, a startling point of interest is the version 3.0. At that release, there was a significant decrease in the number of methods of interest and of the total number of lines of code. In fact, this is the strongest decrease observed during the entire life-cycle of Firefox. However, our similarity analysis revealed that the closest matching method in version 2.0 of every method in version 3.0 is on average 13.65% different. Only one version (4.0) has an average per method difference higher than this one. This strongly suggests a major re-engineering of the application at that specific moment because, as most of the functionalities of version 2.0 survived in version 3.0, most of the changes must have occurred in the design of the application. In the fast release cycle, no such obvious evidences of major re-engineering activities may be observed even though minor re-engineering activities may have occurred.

Since the time between major versions decreased after the release cycle change, we also measured the global change between version 18.0 and 5.0, which represents 651 days of

Table 9.2 Summary of changes between versions. Relative values are relative to the total number of methods in the preceding version.

Version #	ΔMLOCs	Δ Methods	Relative Δ Methods	Average difference	Number of NZ methods	Average difference of NZ methods
1.0	N.D.	N.D.	N.D.	N.D.	N.D.	N.D.
2.0	+0.44	+2,912	+11.96%	0.0956	11,835	0.2200
3.0	-0.35	-2,774	-10.18%	0.1365	13,076	0.2556
4.0	+0.97	+7,094	+28.98%	0.2174	21,279	0.3226
5.0	-0.04	-136	-0.40%	0.0090	1,971	0.1441
6.0	-0.04	-199	-0.60%	0.0112	2,386	0.1466
7.0	-0.01	+69	+0.20%	0.0131	2,433	0.1692
8.0	-0.02	+9	+0.04%	0.0093	1,707	0.1700
9.0	+0.06	+318	+1.02%	0.0155	2,909	0.1636
10.0	+0.24	+1,986	+6.48%	0.0462	9,516	0.1583
11.0	+0.02	-135	-0.42%	0.0164	2,919	0.1830
12.0	+0.04	+150	+0.46%	0.0120	2,673	0.1471
13.0	+0.04	+253	+0.77%	0.0124	2,764	0.1479
14.0*	+0.09	+724	+2.21%	0.0192	3,031	0.2125
15.0	-0.08	+871	+2.59%	0.0220	3,925	0.1930
16.0	+0.66	+4,771	+13.84%	0.0535	6,734	0.3116
17.0	+0.08	+538	+1.37%	0.0149	3,725	0.1590
18.0	+0.48	+3,206	+8.06%	0.0532	8,640	0.2648

development. A total of 1.52MLOC (+32.00%), the largest increase since the original release of Firefox 1.0, were added during that period. Also, 11,417 methods (+36.16%) were added, which is also represents the largest increase in this category.

A careful look at tables 9.1 and 9.2 highlights an important fact: almost all of these changes (more than 80%) occurred in version 16.0, 17.0 and 18.0, or in the last 126 days. This represents 19.35% of all the development time since the release cycle change. Consistently, we observe an average difference and an average NZ difference of 0.1998 and 0.3046 between versions 18.0 and 5.0.

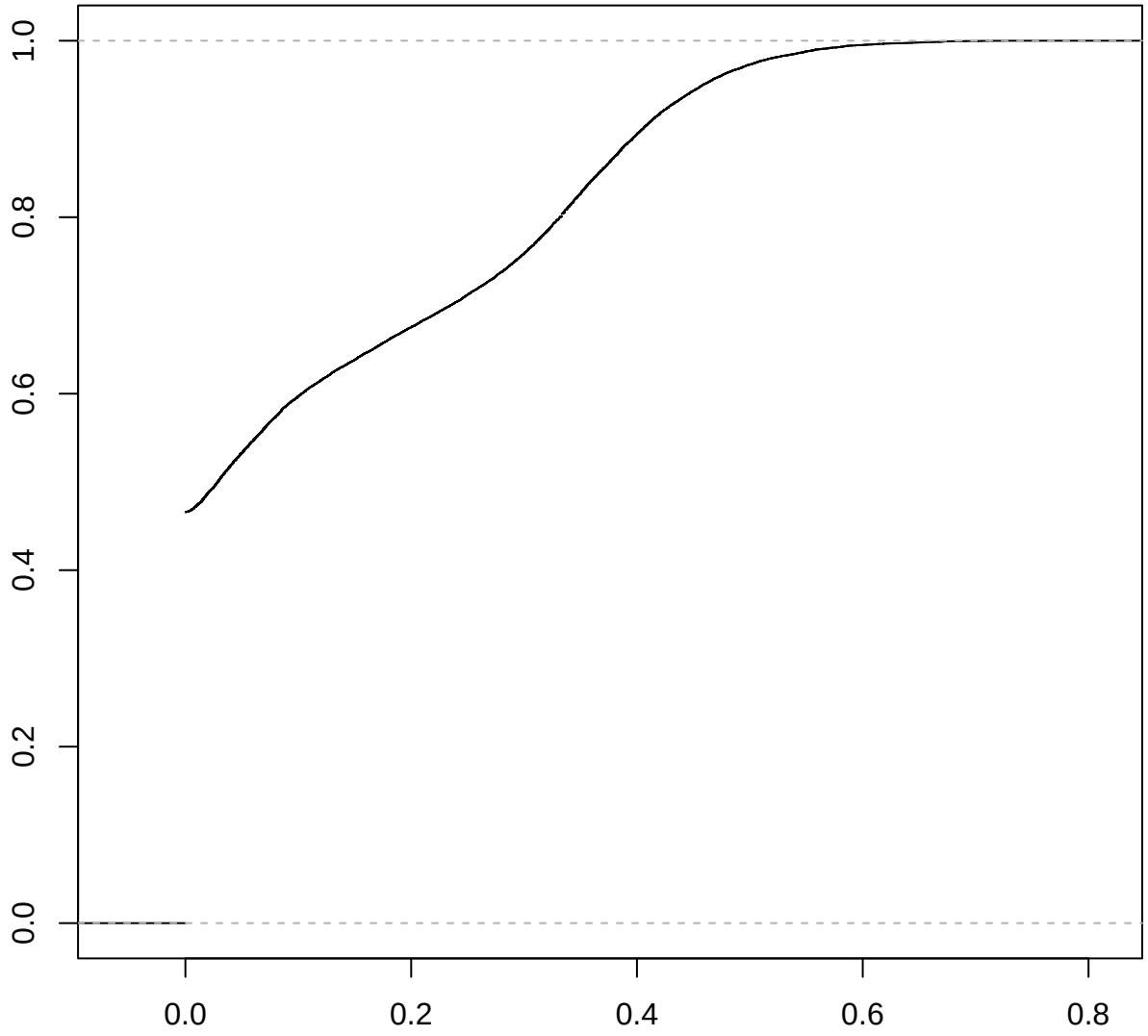


Figure 9.1 Cumulative distribution of the differences between methods of Firefox 3.0 and their respective closest match in Firefox 2.0.

In order to better understand the distribution of changes between versions, figures 9.1, 9.2, 9.3, and 9.4, show exemplary change cumulative distributions from three different interesting

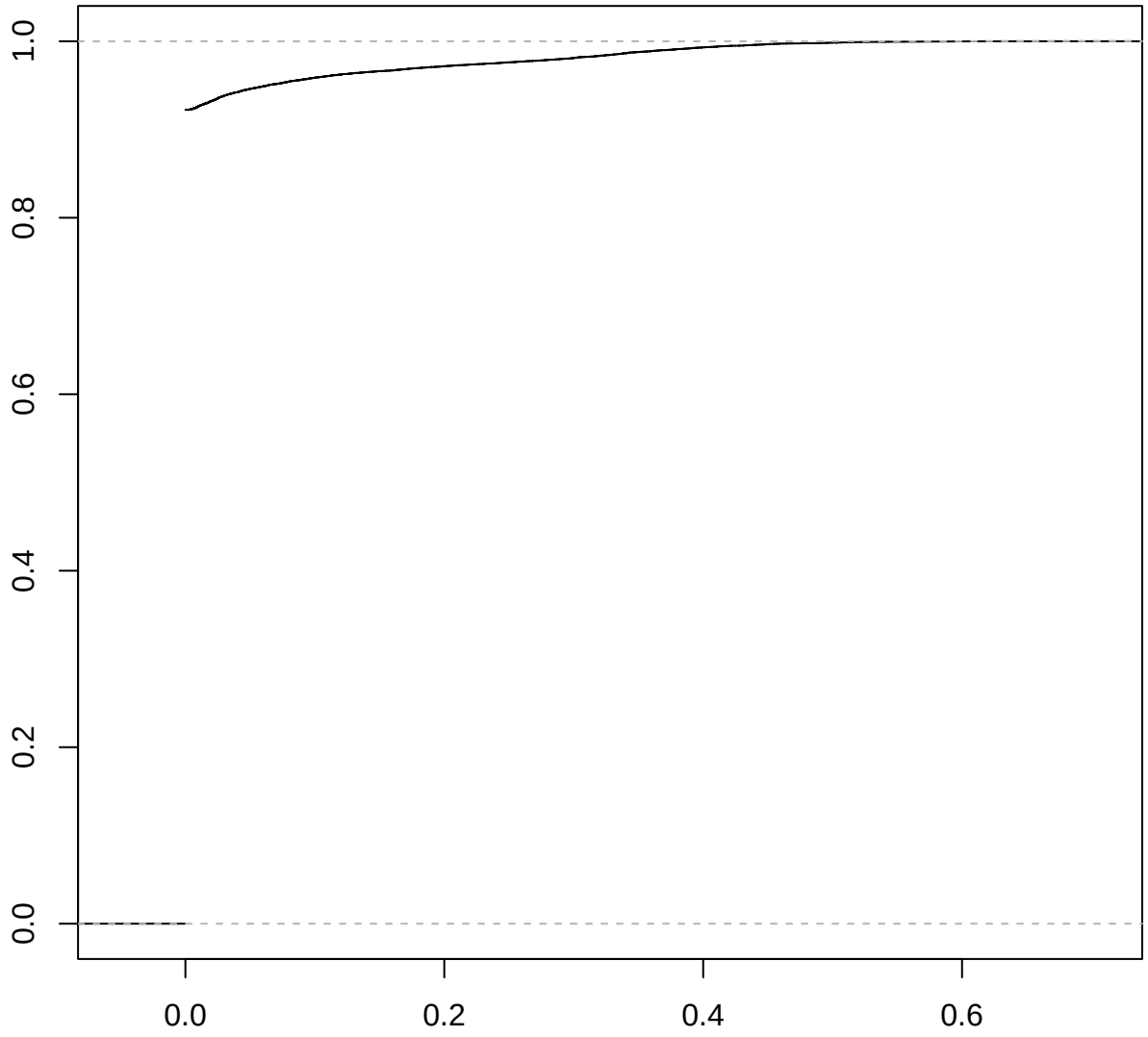


Figure 9.2 Cumulative distribution of the differences between methods of Firefox 7.0 and their respective closest match in Firefox 6.0.

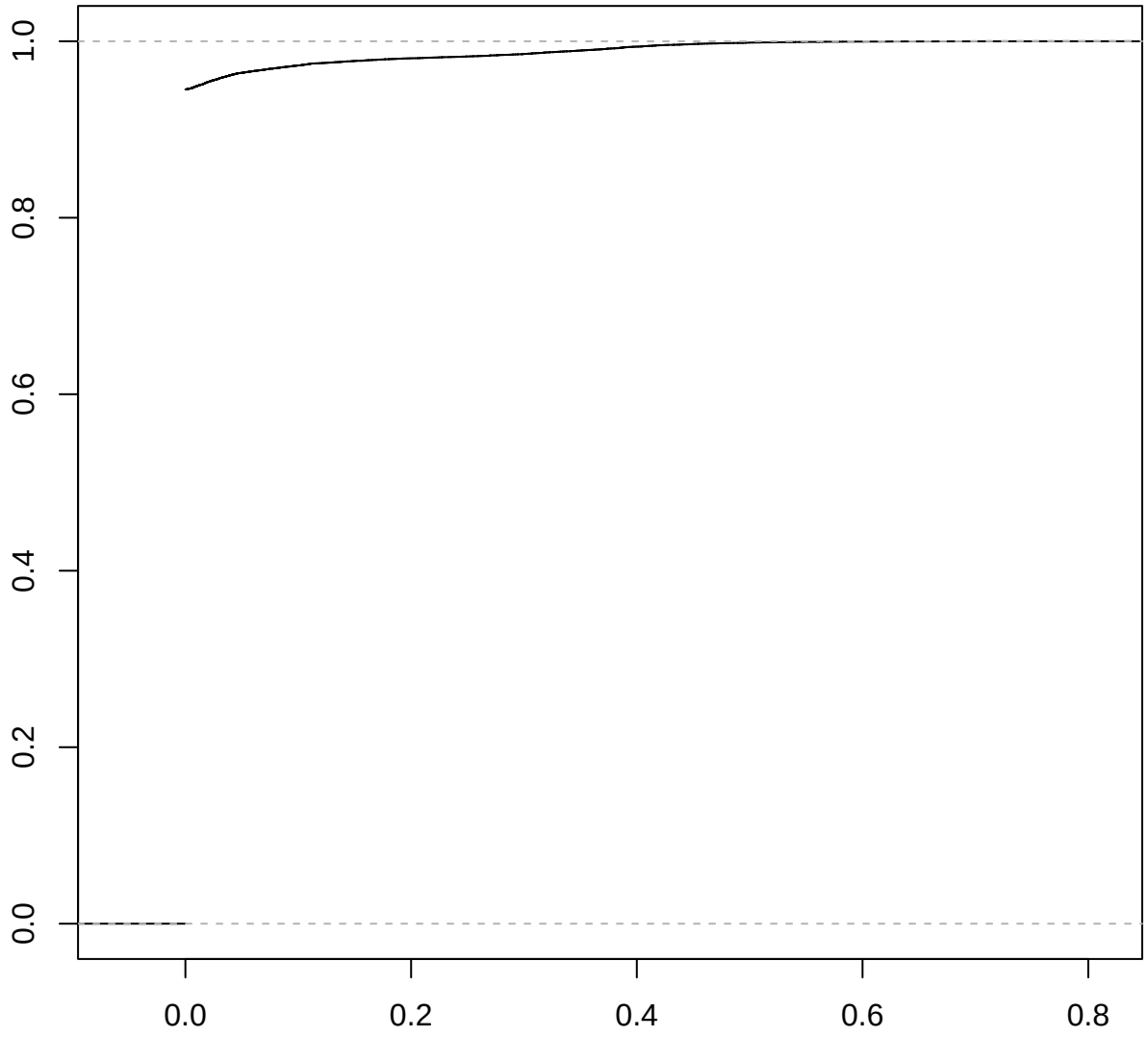


Figure 9.3 Cumulative distribution of the differences between methods of Firefox 8.0 and their respective closest match in Firefox 7.0.

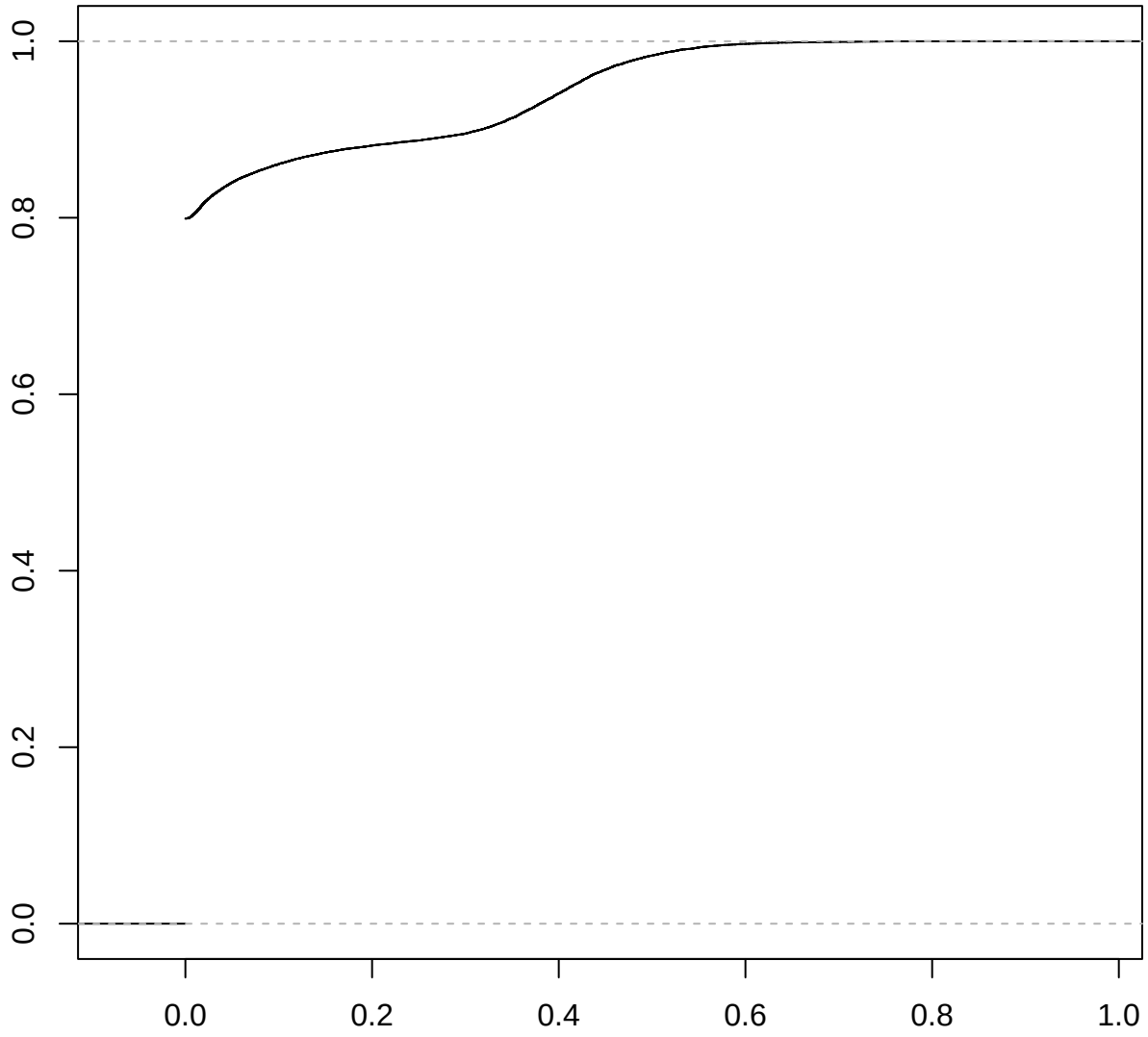


Figure 9.4 Cumulative distribution of the differences between methods of Firefox 18.0 and their respective closest match in Firefox 17.0.

Firefox versions. On all of them, the y-axis is the probability and x-axis is the normalized change measure. The first, figure 9.1, shows the distribution of version 3.0 (in-comparison to its predecessor version 2.0), where we believe major re-engineering activities occurred. The second and third, figure 9.2 and 9.3, show the distribution of version 7.0 and 8.0, a period where the release cycle seems unstable. We presume this instability to be caused by the transition from one release cycle to another. The last, figure 9.4, shows the distribution of the latest release (18.0), one that has seen the most changes in a long time.

Comparing all four figures, we conclude that the distribution of changes from version 7.0 and 8.0 is completely different from versions 2.0 and 18.0, though these last two curves are not too different as they share common growth behavior. Interestingly, change distributions for all versions between 5.0 and 15.0 are very similar to version 7.0 and 8.0. Their curves have the same monotonic rapid growth behavior with a steadily declining growth rate. For versions 2.0, 18.0, and all other versions not in the 5.0 to 15.0 span, the curves exhibit the same pattern: rapid growth at the beginning with a decreasing growth rate, followed by an increase of the growth rate and progressively a second phase of rapid growth, finally followed by a steady decline of the growth rate. This seems to indicate a clear dichotomy of all changes in the following two categories: changes mostly occurring in the existing code base (the left part of the curve covering the smaller differences) and changes which imply new functionalities (the right part of the curve covering the greater differences). However, even if this dichotomy holds, this does not enable us to identify the nature of the changes inside each of the two categories. Nevertheless, the existence of curves that do not exhibit the presence of the dichotomy of changes (i.e. figure 9.2 and 9.3) support the hypothesis that most of the changes that occurred in the associated versions are most likely to be maintenance activities.

9.5 Discussion

Before the final discussing our results, we will now introduce additional facts known from before and after the release cycle change in order to add a qualitative aspect to our analysis. All these facts are drawn from [63]. In that paper, the authors investigated Firefox from version 3.0 to 13.0 in order to determine effects of the release cycle change. Their results show a stable trend and we will make the hypothesis that, extrapolating from their results, the behavior of the missing versions should not differ that much as only 210 days have passed in between. The following two facts are the ones best completing our analysis.

First, the median time before failure is significantly different before and after the release cycle change. For the first 4 versions, the median time is about 720 minutes. For the following releases, it is about 360 minutes. Since a higher median time before failure is better from a

resilience point of view, it appears that Firefox became less resilient after the release cycle change.

Second, the proportion of code submission related to bug fixes increased of at least 25% and now represents more than 95% of all submissions after the release cycle change. Although this does not indicate whether there has been an increase in the number of bugs or an increase in bug fixing of older latent bugs, it does suggest that a larger proportion of the work on the already existing code base focused more on bug fixing rather than re-engineering or code improvement.

Keeping the above in mind, we will now proceed with the analysis of our results.

As shown in section 9.4, a rapid decline in both the number of methods changed and the average difference between methods suddenly occurred after the change of release cycle type. Moreover, a steady increase of bug fixing activities has been observed since the transition to the new release cycle. As patterns of changes in the figures of section 9.4 have shown, the transition from one release cycle to another tend to focus the developers time on maintaining the code, which is concordant with previous observations from [63]. This also supports our claim that maintenance activities generate lower differences in code in general. However, it also suggests that smaller changes became a more important risk after the transition in the cycle.

Interestingly, after almost 2 years of fast cycle development, the total amount of change becomes comparable to the total amount of changes between older major versions. This leads to the observation that even if the release cycle changed, the total amount of work put in the same period of time may still be similar. Following the transition, much less functionality expanding activities have occurred and modifications in the existing code base are suspected to have been orientated mainly around bug fixing. After almost a year, functionality expansion as well as re-engineering activities seem to have resumed as both the number of new methods and the average difference have increased. These observations lead to an important question: are some modifications being developed under a longer, hidden release cycle ? In other words, are some developers delaying the introduction of some modifications in a way that the actual fast release cycle may have some latent remains of the original development cycle ? Our data are not sufficient to provide a definite answer, but they strongly encourage to investigate further in order to understand what has really changed.

9.5.1 Possible Conclusions

To summarize our results and the preceding discussion, we propose the following conclusions to be the most likely to hold.

- Even though the changes are smaller, the fast development cycle seems to made them

riskier. As observed, the average change in NZ difference methods between versions is smaller in the fast development cycle. However, the increase in bug fixing activities as well as the decreased median time before failure point towards changes with a higher risk of default.

- The change in the speed of the release cycle made it harder to perform re-engineering activities. As noted before, the relative number of code submissions exclusively related to bug fixes increased significantly after the change in the release cycle. Overall, less important changes have been made between versions, and whenever this is the case, there is a raise in the number of introduced methods, it suggests that less re-engineering activities have been performed. Also, compared to version 3.0, no other version has seen a significant decrease of methods with an important average change. Thus, we conclude that most of the development activities since version 5.0 are either bug fixing or functionality expanding. However, the observed hindering in re-engineering may be resolved in future versions as the latest versions show an increased amount of activity that may pertain to re-engineering.

- The semantic of a major version release changed with the transition to the new release cycle. In fact, a lower amount of change usually manifests itself between the new major versions, except for recent versions 16.0, and 18.0. As a matter of fact, from a software similarity point of view, all the releases put together are equivalent to only one major version since the transition. If this phenomenon is still a persisting consequence of the transition from 1.5 years ago, to be resolved in the future, or will prove itself to be a trend remains to be seen.

9.5.2 Applicability of Clone and Similarity Analysis

In this experiment, many variables that account for changes between software versions have been observed. However, without computing the average changes on a method basis, it would be difficult to tell what actually happens in the system between versions. Our clone analysis helped establish a base line of quantitative changes between major versions in the original release cycle in order to better understand the change behavior in the fast release cycle. Even if the clone detector was not used to perform a traditional clone analysis, the ability to trace the global pattern of changes throughout different versions still gives insightful information concerning the evolution of a system. In particular, the clone analysis helped identify that even if changes keep methods closer to their most likely ancestor, they tend to be riskier.

Perhaps, our most interesting results are the curves generated from the complete distribution of changes between versions. From these curves, it is easy to isolate versions dominated

by maintenance and bug fixing activities from versions with re-engineering and implementation of new features. Whether this holds or not for other systems in other contexts is a valuable research question.

9.5.3 Threats to Validity

There are some threats to validity and the scope of conclusions presented in this paper.

Regarding the variation of the number of methods of interest, this number may be influenced by both newly added methods and methods that exceed or fall below our threshold of significance between two versions. However, it is not expected that most of the observed added methods are due to jittering moves around the threshold of significance.

An other threat to validity is the hypotheses we have made regarding some expected behavior from developers. Even though no experimental evidences are provided to reinforce those hypotheses, personal experiences support their validity.

The assumption that our external data extrapolates well to the current time frame is also a threat to validity.

9.6 Related Works

Clone detection state of the art includes different techniques. For type-1 and type-2 clones, AST-based detection has been introduced in [21]. Other detection methods for type-1 and type-2 clones include metrics-based clone detection as in [86], suffix tree-based clone detection as in [45], and string matching clone detection as in [38]. For a detailed survey of clone detection techniques, a good portrait is provided in [98].

A good example of a recent study of similarity between common code base is in [51]. That work identified many replica of common parts from a mobile operating system. In the case of large scale development, these duplications, which never merged together, generate more maintenance and co-maintenance tasks. This study shares common features with ours as it involves many code bases and it aimed at identifying difficulties and impacts based on the inter-similarity of the code bases. Version evolution is however different as it is related to a unique trunk of development. Our study also differs since it goes beyond simple clone analysis and tracks general similarities.

Another good example of an origin study is [47]. However, our study differs a lot as our aim was to get a high-level abstraction of the changes in the software and not to retrieve the exact original context of the changes, merges and splits of methods.

9.7 Conclusion

In this paper, we presented an application of clone detection to resolve some difficulties of understanding version evolution in two different release cycles. The approach successfully gave insights of how a change of release cycle may influence the evolution of a software. Moreover, it helps identify dominating activities between different versions. Corroborated by external data, our analysis also shown a tendency of increased risk in fast release cycles even if the code modifications are in general smaller.

Of course, this paper was only exploratory, but it suggests to investigate further the impact of fast release cycles. Finer analysis based on sub-components similarity instead of entire systems is a possible extension to this work. Analysis of different projects in order to see if the observed patterns in Firefox will manifest themselves is also another possible extension.

Acknowledgment

The authors wish to thank François Gauthier for his comments and many usefull discussions around the topic of this paper. This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program and the Fonds Quebecois Nature et Technologie (FQRNT).

CHAPTER 10

COMMENTS ON PAPER 4

This paper relies on the following hypothesis: if a function exists in version N+1 of a system, then its closest match in version N is likely to be the same function if they are close enough. This hypothesis was found to be reasonable in the paper presented later in chapter 12. One must recall that for this type of experiment, the likelihood of this hypothesis holding is higher if the clone detection is performed on string image and not token types.

Many observations made in the paper were preliminary and would require further investigation. What is believed to support most of the claims should be interpreted as evidences supporting further investigation and not definitive conclusions. For example, while we knew beforehand that the number of bugfixing activites had increased in firefox and that we suspected those activities have little impact over the similarity between versions, we didn't actually quantify the exact number of methods that were affected by maintenance and bugfixing activities. We believe it is possible that a strong link may be established between the behaviour of similarity between versions and the type of engineering activities that occured between those versions, but this is out of the scope of this paper.

CHAPTER 11

PAPER 5: TTCN-3 TEST SUITES CLONE ANALYSIS IN AN INDUSTRIAL TELECOMMUNICATION SETTING

11.1 Abstract

This paper presents a novel experiment focused on detecting and analyzing clones in test suites written in TTCN-3, a standard telecommunication test script language. Results are presented in figures and diagrams reporting clone frequencies, types, and similarity distributions. Processing times are presented including parsing, clone clustering, and Dynamic Programming visualization. A statistical comparison of the results is presented and shows that the TTCN-3 clone distribution largely differs from previously investigated software. Clone management and refactoring opportunities are discussed according to the distinguished characteristics of the TTCN-3 clone population. A discussion of the lessons learned and future research is also presented. Practical applications of clone detection in test scipts from an industrial point of view, along with implementation details specific to industrial settings, are discussed.

Keywords: clone detection, telecommunications software, test suite, case study

11.2 Introduction

Clone analysis involves finding similar code fragments in source code as well as interpreting and using the results to tackle design, testing, and other software engineering issues, such as those presented in [16, 36, 49].

Clones are usually divided into three categories, and possibly more, as in [103, 97]. Type 1 clones are identical code fragments, type 2 clones are parametric fragments, and type 3 clones non-identical similar code fragments, also called gapped clones and near-miss clones.

While experiments and techniques for clones in source code have been published extensively, this is, to the best of our knowledge, the first investigation of clones in test scripts in the literature. Tests are written using script languages, differing from programming languages in many aspects.

This project is a collaboration between Ericsson and Ecole Polytechnique of Montreal. As a proponent of TTCN-3, Ericsson develops TITAN, a plugin framework and working environment for TTCN-3. The TTCN-3 language is also Promoted by the 3rd Generation

Partnership Project (3GPP) and as part of ISO 9646, this language is widespread throughout the telecommunication industry.

Interestingly, personal experience and discussions with teams at Ericsson suggested that clones may exist in TTCN-3 scripts. Some even suspect that one of the main idioms for code reusability in the current testing process is to reuse by cloning. However, the management suspects it increases the cost of maintenance and also leads to possible incoherence between tests and evolving software versions. In the long run, Ericsson hopes to reduce the effort in maintenance, design, and understandability of their TTCN-3 test scripts in order to improve the quality of the maintenance process. As Ericsson had previous experience with clone detection technologies [90] and they believe reduction in cloning could benefit some of the aspects they seek to improve, they elected to do clone analysis on TTCN-3. Before exploring applied solutions, a rigorous verification by means of quantitative figures was in order to measure the extent of the existence of clones in TTCN-3.

In this paper, we make an industrial case study of the clones in test scripts written in the TTCN-3 language and compare them to clones in C/C++ and Java. This study has the following goals:

- To verify the existence of clones in TTCN-3 scripts
- To quantify to what extent clones exist in TTCN-3 scripts
- To categorize the clones according to their type to measure which type is predominant in TTCN-3 scripts
- To compare the distribution of the clones in TTCN-3 with clones from C/C++ and Java systems

To achieve these goals, we report distributions and statistics of the clones in 500kLOC of TTCN-3 scripts. Moreover, we complement our analysis by a statistical comparison of the distribution of the clones in TTCN-3 with the ones previously identified in Ericsson's C/C++ and Java code, as reported in [90]. This previous work showed a detailed distribution of the clones in C/C++ and Java code in the same industrial setting, and the numbers used for comparison are directly drawn from that published work.

The paper is organized thus: Section 11.3 describes the adapation of *CLAN* to TTCN-3; Section 11.4 presents experiments and results; Section 11.5 discusses the results and presents lessons learned; Section 11.6 presents related research; Section 11.7 discusses further research objectives, while section 11.8 concludes the paper.

11.3 Adapting Clone Technology For TTCN-3

The Testing and Test Control Notation Version 3 (TTCN-3) [5] is a language designed to write tests for telecommunication applications. It is standardized and maintained by the European Telecommunication Standards Institute (ETSI) and is also adopted by the International Telecommunication Union (ITU-T). As stated in its introduction [6]:

> TTCN-3 provides all the constructs and features necessary for black box testing. It embodies a rich typing system and powerful matching mechanisms, support for both message-based and procedure-based communication, timer handling, dynamic test configuration including concurrent test behaviour, the concept of verdicts and verdict resolution and much more.

Thus, not only is it natural for a company like Ericsson to use TTCN-3, it is also mandatory.

Ericsson has a small team dedicated to developing tools to support TTCN-3 users inside their company. That team has created a complete programming environment as an Eclipse plugin called Titan. That plugin includes API to access a TTCN-3 parser and some of its components, notably the AST. The main problem was to make *CLAN* able to extract data from Titan in order to perform clone detection in TTCN-3.

Even though the TTCN-3 parser within Titan was accessible via API calls, its extensive use of the Eclipse framework prohibits the parser from being executed from outside of the Eclipse environment. Since the current clone detection architecture deployed at Ericsson runs on a virtual machine on a distant server, this proved to be a major obstacle to overcome.

The most important problem was the large memory overhead of Eclipse, which was incompatible with the virtual machine's limited ressources. To reduce ressource consumption, we designed a headless plugin meant to start only the bare essential of Eclipse required to run the Titan environment. This allowed us to interface the parsing architecture of *CLAN* with the parser of Titan. However, in the end we still had to increase the total amount of main memory on the virtual machine, from 3GB to 8GB. Considering the average size of a TTCN-3 project is much smaller than a standard telecommunication application, the headless plugin with the increased size of the main memory is expected to be enough to handle most TTCN-3 projects.

Contrary to many compiling suites, Titan does not provide any API to get tokens from its lexer. Since *CLAN* relies on tokens to visualize clones, we had to add an extra post-processing step. While designing an independant TTCN-3 lexer to extract the tokens from the selected fragments at the parsing step would have been possible, it was deemed too time consuming for this project. We relied on the fact that TTCN-3 was strongly inspired by Java, and chose to use the previously deployed Java lexer in the *CLAN* environment. Since the

tokens are only used for the fine-grained matching and the visualization, we did not expect to encounter any noticeable distortion. The quality of the results, shown in section 13.1.1, was high enough to confirm that intuition.

In the end, making Titan and *CLAN* work together made it possible to integrate clone detection in TTCN-3 scripts in a transparent fashion, without the detection process for C/C++ and Java interfering with the new component and vice versa. High flexibility in *CLAN* design was key into achieving that integration. For Ericsson, that flexibility was important as well as keeping the integrity of Titan. It was imperative that we make no requests to change any part of Titan to deploy our software.

Once information was extracted from Titan, the current clone detection environment was able to work as it is currently installed. We now present the experiments and the results.

11.4 Experiments

11.4.1 Experimental Setup

Hardware, Software Environment, and Dataset

The system architecture for the experiment is based on a virtual machine with Ubuntu Server 11.10, 1 CPU, 8GB of RAM and 200 GB of hard drive disk space and relies on Apache/MySQL/PHP, Java 7 and *CLAN*.

Three sets of test scripts for different projects have been analyzed. Table 13.2 shows the size of each set, and overall, they are all of moderate size. As is expected for systems of that size, *CLAN* clustering algorithm execution is very fast. The exact execution time for all the systems together is reported in Table 11.2.

In an industrial setting, careful considerations need to be taken to manage hardware constraints. Compared to our previous experiment [90], we had to more than double the RAM of the virtual machine, from 3GB to 8GB. This increase in resource requirement is due solely to the use of the Titan plugin in the Eclipse environment; Titan itself is not memory intensive, but requires usage of Eclipse's framework which is much more memory hungry than the slim tools used in previous work. Even in headless mode, Eclipse remains the memory bottleneck of the entire process. Thus, creating a viable experimental setup depended on the small memory footprint of our tool to accomodate the memory needs of other necessary components of the process.

Table 11.1 Sizes of Test Script Sets

Project	TTCN (*LOC*)
A	186,349
B	181,518
C	199,836
Total	567,703

Table 11.2 All systems clone clustering and *DP* computation time

Language	Clusters (secs)	*DP* (mins)
TTCN	0.548	58.670

Clone Detection Process and Configuration

The *CLAN* tool used in this experiment computes code fragments' similarities by considering syntactically based software metrics and it can be used for identifying code duplication in large software systems. The process used to study clone detection is outlined in Figure 11.1 and consists of the following sequential steps:

1. TTCN-3 Lexical and syntactic analysis
2. Extraction of the metrics
3. Clone clusters identification
4. Dynamic Programming Analysis and Visualization

In Figure 11.1, this clone detection process with *CLAN* is depicted graphically. The following is a quick overview of the process as shown in the figure. The first box at the far left shows projects with test script files. All the files in the different projects are first analyzed lexically and syntactically. A set of metrics, listed below, are then extracted and used to compute clone clusters. Finally, clone pairs are fetched from the clusters and the Longest Common Subsequence (LCS) between the two fragments is computed with dynamic programming; this step is referred to as *DP*-matching. The rest of the section describes in detail each step of the process.

Similarity analysis using software metrics requires a particular fragment granularity and a set of metrics to characterize the fragments. Functions and methods are the chosen granular-

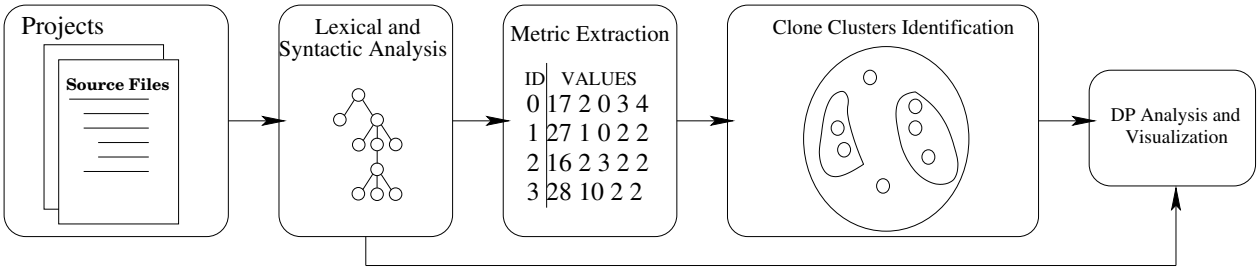


Figure 11.1 Clone detection process with *CLAN*

ity for the fragments in this setup. The lexical and syntactic analysis identifies all functions and methods in TTCN-3 projects and lists them for further reference in the process.

The extraction of the metrics is then performed on the list of all the functions and all the methods. The set of metrics in this experiment are: the size (measured by the number of AST nodes), the number of passed parameters, the number of if statements, the number of while statements, and the number of local variables.

The fragments are then clustered by *CLAN*. In our experiment, fragments are clustered together if all their metrics are equal, except for the size of fragments. This is measured by the number of AST nodes, which is allowed to vary by 100 units.

The last step is *DP*-matching, which is executed using the implementation of an algorithm similar to that described in [68]. Fragments are reported as similar, if the set representation of the LCS computed with *DP*-matching has a Jaccard coefficient on matched token types greater or equal to 0.7.

Fine-grained analysis using *DP*-matching allows the classification of clones as identical (type-1), parametric (type-2), or different (type-3).

Identical clones are identified by a 100% image similarity, since token types and token images are identical.

Parametric clones are fragments that have an identical sequence of token types, but less than 100% image similarity (due to identifiers, constants, etc.). They are good candidates for re-factoring.

Other clones, of type-3, have both image and token type similarity under 100%.

The results presented in the next section follow this classification.

11.4.2 TTCN-3 Results

Detailed statistics from the clone detection step are presented in table 11.3. The maximum cluster size is 29 and, after manual inspection, that cluster turned out to contain a number of false-positives. The average cluster size is 2.39. It is surprisingly small and indicates that

TTCN EXPERIMENTS	N	MAX SIZE	AVG SIZE
Clusters	2,248	29	2.39
Fragments	16,318	661	24.26
Cloned fragments	4,025	661	24.32
Identical (Type-1) cloned fragments	3,336	364	23.89
Parametric (Type-2) cloned fragments	616	661	26.72
Higher (Type-3) cloned fragments	73	203	23.72

Table 11.3 TTCN Detailed Clone Clustering

most clones only exist in pairs. A total of 4,025 fragments were identified as clones from a total 16,318 analyzed fragments, resulting in a 24.67% cloning ratio over all of the considered fragments. This translates to approximately 100kLOC of cloned code. Reported ratios in the literature [100] for open source C/C++ and Java projects vary on average from 2% to 10%, with outliers around 20%. Our previous experiment [90] showed a cloning ratio around 10% in the same industrial setting. This suggests that cloning practices in TTCN-3 may differ significantly from practices in other languages used within the same industry and in open source projects. In section 11.4.3, we show that this difference is indeed statistically significant within our industrial context.

Identical or type-1 clone is the predominant variety of clones in this experiment. 3,336 fragments out of the 4,025 identified similar fragments also have type similarity equal to 1.0.

Parametric or type-2 clones have type similarity equal to 1.0, but an image similarity different from 1.0. We counted 616 fragments of this type. The remaining 73 fragments are clones of type-3 or higher, but are marginal in this experiment.

In the following figures and diagrams, cluster size shows the cardinality of the set of mutually similar fragments

Figure 11.2 shows the distribution of cluster sizes with respect to cluster size ranking,

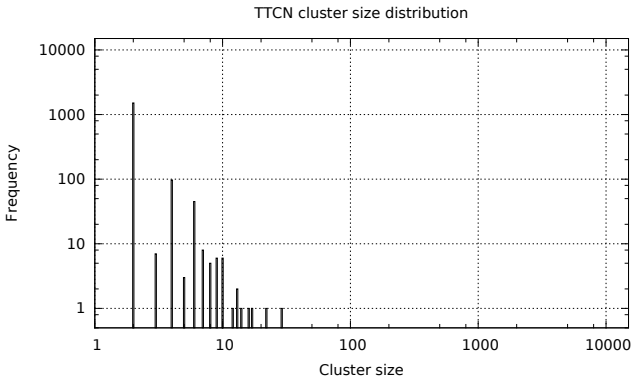


Figure 11.2 Cluster size distribution

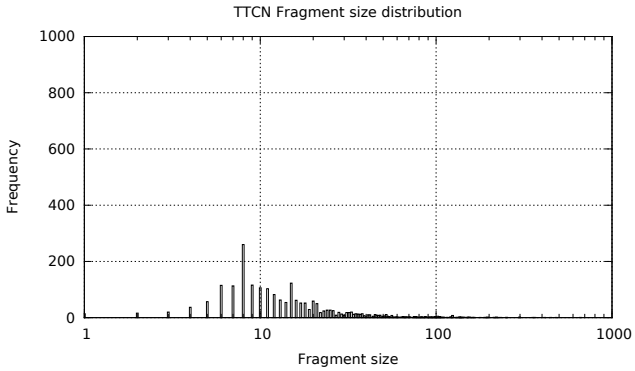


Figure 11.3 Fragment size distribution

starting from the smallest ranking. The figure clearly displays the high number of clusters that are actually only pairs of clones instead of large clusters.

Figure 11.3 indicates the fragment size distribution of fragments involved in a clone relation. Sizes are measured in LOCs. Fragments are ranked in order of increasing size.

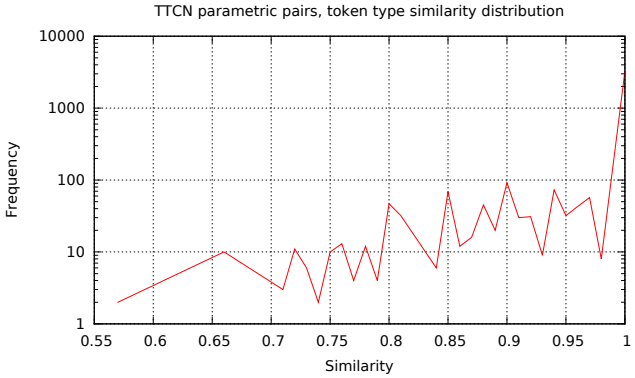


Figure 11.4 Connected histogram of parametric similarity distribution

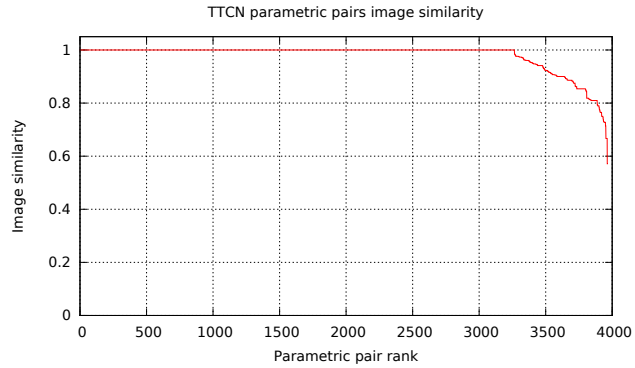


Figure 11.5 Decreasing cumulative curve of parametric similarity distribution

Figures 11.4 and 11.5 report the distribution of token type similarity between fragment pairs in clusters. Pairs are ranked by increasing degree of similarity, from completely similar pairs with a similarity equal to 1.0 to the less similar pairs with a similarity equal to 0.7. Figure 11.5 clearly shows that type-3 clones are marginal and could almost be considered statistical aberrations.

Figures 11.6 and 11.7 gives more details about the distribution of totally type similar pairs, by showing the distribution of their image similarity. Pairs are ranked from the most similar images to the least similar ones, while preserving total type similarity. Image similarity ranges from 100% down to 57%. The most interesting feature of figure 11.7 is the high number of non-identical clones sharing more than 80% of their image. Starting slightly above 3,336,

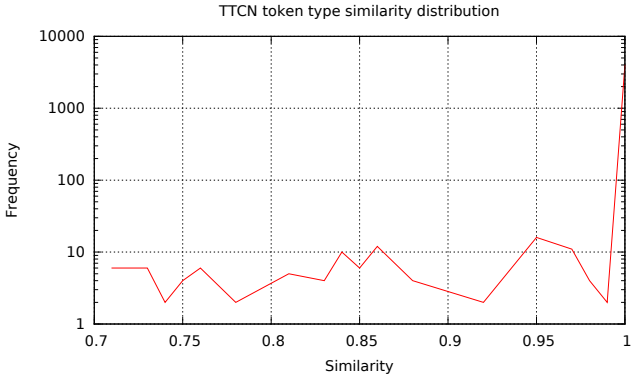


Figure 11.6 Connected histogram of similarity distribution

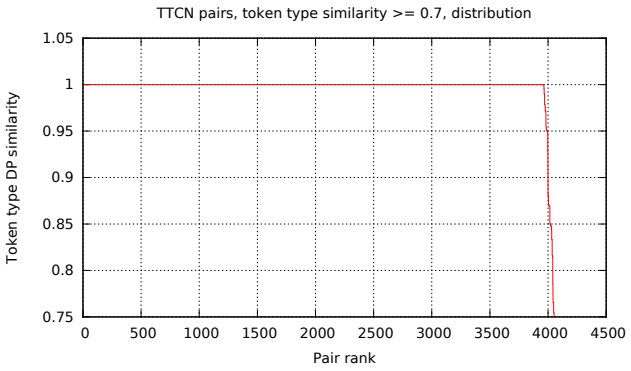


Figure 11.7 Decreasing cumulative curve of similarity distribution

the curve dips under 1.0 similarity and steadily declines to reach 0.8 at around 3,800, after which the curve displays a steeper decrease. There is around 500 pairs in the region described above, and since all of those pairs have 1.0 type similarity, these 500 parametric pairs only have a very few number of parameters, since they share 80% or more of their image.

11.4.3 Statistical analysis

In this section, we investigate the statistical significance of the differences between the TTCN-3 clone similarity distribution presented in figure 11.7 and the results obtained previously for C/C++ and Java in [90]. We first apply a Mann-Whitney U-test to show the TTCN-3 distribution differs from the distribution in C/C++ and Java . Second, we apply a Fisher exact test to verify whether the proportion of parametric clones in TTCN-3 is higher or not than in the other two languages. The detailed steps are explained in the following sections.

Mann-Whitney U test

The Mann-Whitney U test is used to determine if two samples are drawn from the same population. It is usually used when the samples are of significantly different sizes. It is a nonparametric test and is also called Wilcoxon rank-sum test.

Here, we wanted to verify if samples from the TTCN dataset are generated from a different process than the datasets of C/C++ and Java. The null hypothesis is that, pairwise, TTCN-3 differs from C/C++ and Java. We will reject the null hypothesis if, pairwise, the datasets are from different populations.

The Mann-Whitney U test, uses the ranks of all the observations of a sample. It compares the ranks of the observations of a sample to those of another sample. Computing the p-value

leads to rejection, or not, of the null hypothesis. In the following equations, the p-value for the test between TTCN-3 and C/C++, and between TTCN-3 and Java are presented:

$$p_{TTCN-C/C++} < 5.1604 \times 10^{-234}$$

$$p_{TTCN-Java} < 2.8804 \times 10^{-291}$$

As the p-values are extremely low, we reject the null hypothesis and conclude that TTCN-3 samples are likely generated by a distinct process other than the process underlying C/C++ and Java. We did not elect to chose a specific significance threshold as the p-value tends towards zero and we could have rejected the null hypothesis for all common thresholds.

Fisher's exact test

Tests of statistical significance are of interest for a quantitave comparison of two samples of software clones. Among this category of tests, Fisher's exact test can be used to evaluate the associations between two binary variables for which data can be expressed by a contingency table [108]. For a contingency table as in Table 11.4, Fisher's exact test evaluates the p-value of obtaining the observed distribution of the partition of the sets represented in the contingency table under the null hypothesis that the underlying probabilities of getting a certain arrangement follows an hypergeometric law:

$$p = \frac{\binom{A+B}{A}\binom{C+D}{C}}{\binom{N}{A+C}}$$

$$N = A + B + C + D$$

Table 11.4 Example of a contingency table

	Variable 2		
Variable 1	Value 1	Value 2	
Value 1	A	B	$A+B$
Value 2	C	D	$C+D$
	$A+C$	$B+D$	N

Applied in the same pairwise pattern as the Mann-Whitney U-test, we compare TTCN-3 and C/C++, and TTCN-3 and Java, by classifying the clones as either type 1 and 2 or type 3. Type 1 and 2 clones are those with a parametric similarity coefficient of 1.0, and type

3 clones are the others. Table 11.5 presents the comparison between TTCN-3 and C/C++, and Table 11.6 is the comparison between TTCN-3 and Java.

Table 11.5 Contingency table for TTCN and C/C++

	Systems		
Clones	TTCN	C/C++	
Types 1-2	3 964	254 414	258 378
Types 3	102	86 151	86 253
	4 066	340 565	344 631

Table 11.6 Contingency table for TTCN and Java

	Systems		
Clones	TTCN	Java	
Types 1-2	3 964	13 976	17 940
Types 3	102	6 135	6 327
	4 066	20 111	24 267

We chose Fisher's exact test over the Chi-Squared test as the latter has some distortion when the number of elements in a set is either low or is significantly lower than the others sets, which is the case in our experiment [50]. As the amount of type 1 and 2 clones is significantly higher than the number of type 3 clones in all systems, the Chi-Squared test is, de facto, discarded. Also, while Barnard's test would have applied to our experiment, it was impossible to compute it as the computation does not succeed due to the large numbers we have. Thus, Fisher's test was the most approriate.

While the datasets come from the same organization, the languages are different as is the purpose for which they are used: this implies that the independence of the dataset is a reasonable hypothesis. Hence, the use of the Fisher's exact test is sound as it requires independence. Fisher's exact test requires that the two sets of categories are not associated in any way. As a system can only be in a single dataset, this does not violate the pre-requisite of Fisher's test. The p-value of the pairwise test between TTCN-3 and C/C++ and between TTCN-3 and Java are the following:

$$p_{TTCN-C/C++} < 2.2 \times 10^{-16} \tag{11.1}$$

$$p_{TTCN-Java} < 2.2 \times 10^{-16} \tag{11.2}$$

Therefore, under any resonable significance threshold, we reject the null hypothesis in both cases. Hence, we conclude that the proportion of type-1 and type-2 clones are not only

very high in TTCN-3, this proportion is also significantly different from the ones in C/C++ and Java.

11.5 Discussion and Lessons Learned

Although they differ largely from what is reported in the literature, the results are not surprising. The sponsors of the project knew beforehand that a significant amount code was probably duplicated because of the standard way testers write scripts: the principal idiom they use is copying pieces of code and adapting them to fit their needs. Most of the time only parameters change between two tests, it naturally results in many type-2 clones and many type-1 clones between test suites of similar products. There is a natural resistance to migrate from the current paradigm to the use of more evolved abstract constructions and this poses a major challenge to mitigate the current problems: the prohibitve task of maintaining tests properly aligned between releases of two versions of a product.

This experiment differs from previous work because it not only analyzes a new language, TTCN-3, but it deals with different software, that is test suites instead of applications.

Clone classification scheme and figures, such as the one presented in [17], could be applied to the presented results. Informally, observed differences and changes between TTCN-3 clone fragments are consistent with the above mentionned scheme of [17]. However, most schemes are clearly too elaborated for TTCN-3 results, because the clones are clustered in more restricted categories.

Even though observations and deductions from the presented figures suggest that refactoring opportunities exist, a precise count and classification of TTCN-3 clones differences present in this case study is left for further research. However, the near absence of type-3 clones in TTCN-3 scripts and the dominance of type-2 parametric clones give rise to simple and easily automated refactoring opportunities that are in general less clear in application code. Considering that one of the goals was to assess the possibility of applying automated solutions to reduce test evolutions and migrations cost, by either keeping track of the clones or removing them, the clone analysis clearly shows that such a possibility exists.

The *CLAN* configuration was similar to that used for code in C/C++ and Java. Thus at an equivalent level of similarity, TTCN-3 contains a significantly lesser amount of higher type clones and has a preponderant number of type-1 and type-2 clones.

Interestingly, the sponsor tried alternative techniques to clearly identify the maintenance problem within test suites, notably smells. In the end, however, the sponsor chose to try clone detection, as it seems more fit according to the known coding practices. Clone detection

was also perceived both as a diagnostic method and a solution provider instead of solely a diagnostic method.

We also observed that while such an experiment can give the managerial level good insight into the capabilities of our technology, any desire to change a practice or to adopt a new tool must ultimately come from the developpers. This might explain the current decision of migrating clone detection technology from a web-service to a suite of code analysis tools that are designed to run many simultaneous analyzes. This will give developpers a pletoria of data to work with, but most importantly the choice to work with what data they perceive useful. This might help to use our results.

Many companies learn early that a tool that allows many configurations and parameters is not what customers want to deal with. Customization usually comes at a cost of first learning the impact of the parameters on a system and then learning what the best values for a specific condition are. In most cases, only good enough parameter values are required and will get the job done. From that perspective and knowing that developers have very little time to learn new tools, it was of utmost importance that the interface to the clone detection tool exposes the user to the least amount of customization. A small loss of performance is acceptable if it shortens the learning curve for many users. Based on that, it was of utmost importance that the tool presented to the developers for clone detection be as simple as possible to use for clone information extraction and not take up a lot of their time. It was also very important that our tool be adaptable to the TITAN environment.

Compared to previous research, visualization, although appreciated by the managerial level, was not a key factor in adopting the technology. This may be partly explained by the exclusive involvement of the managerial level and no developers. From what we learned, managers are usually more interested in gathering facts, interpretations, and possible solutions than taking a direct look at fine-grained results.

Testers are also not interested in software clones because they generally program using a cloning paradigm and while managers perceive it as a problem, testers agree that it is a best-practice. Therefore, even if they know they have software similar to some other system in the company, they do not want to do anything about it. Visualization or other form of summarized results did not change the view of testers on that matter.

Thus, even with a working technology, workable solutions and evidence supporting the use of those solutions, the cultural setting of the testing groups was enough to severly hinder adoption of a new tool and migration towards a different paradigm.

Also, where in general software researchers argue that awareness of clones is always at least a step in the right direction, awareness is not a problem in TTCN-3 scripts. Indeed, cloning patterns were predicted before the experiment even started and clones were perceived

as a problem. However, even if the problem was acknowledged beforehand and the results are clear and useable, it may still not be enough to investigate the problem further.

The presence of refactoring tools available to developers within their *IDE* of choice (mainly Eclipse is used within Ericsson) is not even enough to encourage actions to reduce the number of clones, even considering that TTCN-3 clones are mostly limited to parametric types and are therefore easier to refactor automatically.

In any large company, making changes affecting the company structure takes some time. The awareness of software clones in TTCN-3 scripts do generate talk among managers, but in the end, social changes are required before technological changes.

11.6 Related Work

Clones are well-established in the literature with hundreds of papers dedicated to their detection and application. A comprehensive survey of the field is in [99]. In the following, we relate our work to already published papers explaining similarities and differences.

11.6.1 Applications in large-scale and industrial context

Large-scale clone detection study in an industrial setting can be traced back to [32]. Large scale, inter-project clone detection was investigated in [70], and [62]. With a growing interest from the industry and the technology achieving scalability, other industrial clone experiments have been published. Notably, authors of [33, 34] from Microsoft did an experiment with code clones and investigated developer feedback. Further developments in industrial applications were produced by [116] and [110] who led experiments on clone application and management in an industrial context. Our research shares the industrial context and large scale projects with those studies. We differ from those previous experiments by being in a broader industrial setting with more than 10 times the lines of code, with many projects coming from different development teams interested in finding clones between projects instead of clones within isolated projects. Also, because of our context, the focus of the experiment was on finding coarse grain sets of clones representing entire common subsystems instead of localized clone management. In that sense, our study is a departure from the traditionnal use of clone detection technology in the literature.

In an industrial context with a focus on different products branching from a common ancestor, inter-system clone detection is a requirement to detect the common parts in the originally cloned sub-systems. Therefore, clone detectors able to operate in an inter-system context may be better suited for industrial applications. Other authors, such as [69] and [29],

like us, have already explored the context of large scale inter-system clone detection. Our industrial results suggest that we should proceed with this approach.

A good example of a study of similarity between common code bases is in [51]. That work identified many replicas of common parts of a mobile operating system. In the case of large scale development, these duplications, which are never merged together, generate more maintenance and co-maintenance tasks. Developers also sometimes end up programming already existing functionalities. This case is closely related to our study, sharing both a common context of telecommunications related applications and heavy similarities centered on operating systems.

Advocating the use of clones as refactoring opportunities is widespread in the literature, as in the recent work of Tsantalis [76]. However, as those works focus on refactoring clone pairs and small clone clusters localized in small regions of the code base, they do not take into account larger cloning phenomenona like the replication of entire sub-modules and thus offer little to actually factor out broader, cohesive software. Moreover, our industrial experiment and feedback from developers suggests that localized clones are readily refactored before being committed to a central repository and thus do not live long enough to become an issue. In the end, our work suggests changing the focus of refactoring activities from localized clones to management of higher abstraction (such as module) clones. This is an important distinction between our findings and the current state-of-the-art of clone analysis.

11.6.2 Summary of Different Clone Detection Techniques

Despite the numerous tools already available today, many new techniques and tools are created and published every year in various conferences and journals. The following will briefly cover the most recent techniques published as well as the established ones still mentioned in the literature.

In recent years, new techniques have tried to solve precision and recall issues with type-3 and type-4 clones. In regards to type-3 cloning, techniques based on n-grams have gained some popularity. Kamiya in [59] has used n-grams to detect type-3 and type-4 clones in Java bytecode. Yuan in [118] has used frequency vectors of 1-gram and the cosine distance to detect type-3 clones. Lavoie in [80] has used frequency of generalized n-grams with the normalized Manhattan distance with space partitioning to detect type-3 clones. Sajnani in [105] has also used similarity on n-grams, but within a parallel framework.

Without relying on n-grams for type-3 clone detection, other tools use distance on token strings or image strings. Murakami in [92] uses a weighted Levenshtein distance, also called Smith-Waterman, to detect type-3 clones. Kamiya also uses token strings in [61].

Some tools use tokens for clone matching, but add syntactic information to help locate

the clones within functional boundaries. This is the case of NiCAD [30], which does syntactic analysis, program transformation, and normalization, and then computes the Longest Common Subsequence (LCS, related to the Levenshtein distance). In this paper, we also use the LCS as the last step of the clone detection process.

Hashing and clustering are also used to detect type-3 clones. Part of the technique we presented in this paper falls into this category. The recent SimCAD [111] also has a hashing and clustering step in its algorithm. ConQAT from [55] is another tool with hashing schemes. Locality sensitive hash (LSH) has been exploited by [57].

Clone detection by suffix tree matching was introduced by the authors of [72]. Although naive suffix tree matching is usually best fit for type-1 and type-2 clone detection, the authors proposed algorithms and heuristics to close gaps between matched segments in order to achieve competitive type-3 clone detection. The technique has been evaluated in [40] and the released tool iClone dates back to the incremental version of the suffix tree in [46].

Older techniques, like [22], used AST-based matching for type-3 clone detection. Although they produced good results, these techniques are less used today because they lack scalability.

Detecting code clones is also possible using by-products of the code instead of the code itself. For example, Program Dependency Graphs (PDG) have been used by [53] and [75]. Analysis of the memory behaviour is another technique and can be found in [64].

Semantic related techniques have emerged in recent years. A prime example of that is in [87], which uses Latent Semantic Indexing (LSI).

Although most of the tools mentioned in this section are incremental, particular attention should be given to [19] which does incremental clone detection on source code repositories, which might be relevant for large scale industrial settings.

11.6.3 Parsing Techniques

Like our work with C/C++ and Java, other clone detectors have tackled difficulties with different languages. Microsoft's C# has been explored by [8], PHP by [39], and Assembler by [35]. Simulink models, although not code, have also been explored with the NiCAD clone detector in [9]. Clone detection in Simulink models has been explored as well in [7] and [37]. Although a minor variant of the existing parsing techniques, the combination of a secondary language lexer with a main parser to infer the original lexical analysis information is a novelty.

11.6.4 Applications

Many applications of clone detection have already been proposed and investigated. Some of them might be relevant to industrial needs. For reference purposes, we provide a quick overview of those applications.

Discordant security clones have been investigated by [39]. Discordance of clones may also be used to detect bugs in programs, like in [84] and in [56].

Application of clone detection technology has also gone beyond source code.

Spreadsheets have gathered interest and were investigated in [52]. Ontology alignment using code clone detectors has also shown promising results in [43].

Among all the currently published applications, we are the first to investigate clones in test suites and clones in TTCN-3.

11.7 Further research

In the previous sections we presented and discussed the findings from the experiments and the lessons learned about clone detection in industrial test suites.

This experiment is already an extension to the work presented in [90], but it also opens new research possibilities.

Automated refactoring of code fragments is already well studied in the literature, but no studies exist on specialized refactoring for test scripts. Because of the very specific nature of the clones in TTCN-3, largely different from standard code clones, it would be interesting to investigate whether or not refactoring techniques perform better on this problem. The presented distribution suggests it should and as such make this extension a promising one.

Automatic classification of the clone stereotypes could be interesting to help refactoring tools. Extensions of the approaches presented in [15, 14] could be advantageously enhanced by targeted user interactions. Such interactions could alleviate the complexity of the analysis necessary in an automatic approach and would give more flexibility to the user.

Investigating whether or not known clusters and possible refactoring of these clone clusters help reduce the total testing effort is another research opportunity. While this was one of the original goals of this project's sponsor, helping testers to actually leverage the gathered clone information is in itself a complete study.

Studies on the differences between code written by developers and testers is also a possibility. It is generally assumed that code writing is done with the same skill set at every step of the development process, but clearly this study suggests otherwise. Understanding why code writing and code idioms differ between different activities in the general software devel-

opment process could be an important step towards a better integration of new technologies like clone detection.

Refinement of the clone detection technique with an emphasis on TTCN-3 could improve the global performance of the tool. Because of the widespread clone stereotypes, some step of the clone detection tool could be optimized to achieve faster results with a smaller memory footprint.

Research in the area of software management and organizational structures to support software development could be carried out to accommodate clone detection and benefit from this information. That could include organizational re-structuring to take advantage of clone analysis, introduction of new technologies into an organization, and defining organizational structures, groups, responsibilities, and organizational processes.

11.8 Conclusion

We reported results of an experiment on clone detection within industrial test suites in TTCN-3. Figures shown in the results section exhibit different characteristics of the clone population in TTCN-3. We found that around 23% of the test suites are cloned, which is a very high proportion of clones compare to what is generally found in source code. We also found that the difference in proportion of parametric clones is statiscally different and thus the proportion of parametric clones is significantly higher in TTCN-3 than in source code.

We want to emphasize the benefits of cooperation between academia and industry. These two groups have very different perspectives on software development. However, these viewpoints are frequently complementary. Time is needed to establish a good relationship built upon mutual understanding, but once such an understanding is achieved each group will benefit from the best of what the other brings to the table. Anyone following the path of cooperation we exemplified should not expect immediate understanding or benefits from the exercise. Yet, with time, the experience may be positive for both parties.

The development of this project allowed us to learn different lessons about deploying a technology withing a group of testers instead of developers. Despite potential benefits, a lot of work is still required for such a technology to become a standard tool in software testing.

Acknowledgements

This research was funded by Ericsson. The authors wish to thank Renaud Lepage and Mario Bonja for their contributions to this paper and the related discussions. They also wish to thank Kristof Szabados for his technical assistance and insights into the Titan environment.

CHAPTER 12

PAPER 6: INFERRING REPOSITORY FILE STRUCTURE MODIFICATIONS USING NEAREST-NEIGHBOR CLONE DETECTION

12.1 Abstract

During the re-engineering of legacy software systems, a good knowledge of the history of past modifications on the system is important to recover the design of the system and transfer its functionalities. In the absence of a reliable revision history, development teams often rely on system experts to identify hidden history and recover software design. In this paper, we propose a new technique to infer the history of repository file modifications of a software system using only past released versions of the system. The proposed technique relies on nearest-neighbor clone detection using the Manhattan distance. We performed an empirical evaluation of the technique using Tomcat, JHotDraw and Adempiere SVN information as our oracle of file operations, and obtained an average precision of 97% and an average recall of 98%. Our evaluation also highlighted the phenomena of implicit *Moves*, which are, *Moves* between a system's versions, that are not recorded in the SVN repository. In the absence of revision history and software experts, development teams can make use of the proposed technique during the re-engineering of their legacy systems.

12.2 Introduction

Software companies depend on Version Control Systems (VCS) to track and manage modifications on their software systems. The information recorded by VCS is key to the success of many software development activities. For example, during a software reengineering, development teams rely on the revision history of the software system to recover its design and transfer its functionalities. However, for many legacy systems, VCS are not available and developers are left to rely solely on their knowledge of the system to uncover the hidden history of modifications. Developers knowledge of a system is often incomplete [94]. Therefore, relying solely on this knowledge is likely to be ineffective. Nevertheless, since source code based evolution analysis can be performed also at release level (*i.e.*, using the source code of the versions released to customers), in the absence of a reliable VCS, it is possible to infer information about files structure and the history of their modifications. Inspired by clone detection techniques, we present an original technique to recover file *moves* information between released versions of a system. The new technique relies on a nearest-neighbor clone

detection approach using the Manhattan distance on frequency vectors of code fragments. To assess the effectiveness of the proposed technique, we perform a case study using the open source software systems Tomcat, JHotDraw and Adempiere. We answer the following research question:

RQ1: *Can the proposed technique recover the revision history of a software system?*

> Using the proposed technique, we are able to identify *moves* of files across the releases of a software system with an average precision of 97% and an average recall of 98%. The proposed technique also identifies *implicit* files movements that are not recorded by VCSs. These *implicit* files movements are generally to results of files cloning.

In the absence of a reliable VCS, development teams can make use of the proposed technique to recover important knowledge about the modifications of their software systems. The proposed technique can also be used to enrich the meta-data of existing VCSs with information about *implicit moves* that are not recorded currently.

The rest of the paper is divided as follows: Section 12.3 summarizes the related literature concerning clone detection and code fragments evolution analysis, Section 12.4 details the proposed algorithm, Section 12.5 presents our experimental setup and our methodology, Section 12.6 reports the results of our experiments, Section 12.7 discusses the obtained results and potential threats to their validity. Finally, Section 12.8 summarizes our work and outlines avenues for future work.

12.3 Literature

This section summarizes the related literature on clone detection techniques, provenance, and clone evolution analysis.

12.3.1 Clone Detection Techniques

The state of the art on clone detection includes many different techniques. For identical and parametric clones, they range from AST-based detection techniques [21] to metrics-based [86], suffix tree-based [45], and string matching [38] detection techniques. Roy and Cordy [104], Göde and Koschke [45], and Lavoie and Merlo [78] have also proposed detection techniques for near-miss clones. A detailed survey of clone detection techniques is presented in [98]. In this paper, we improve on our own clone detection technique [79], and extend it to track similarities between all code fragments (including non cloned code) using the same approximation of the Levenshtein distance. We do this because we believe that clones are only a special case of similarity in software systems.

12.3.2 Provenance and Clone Evolution Analysis

The work presented in this paper is also related to provenance analysis since we track the provenance of code fragments across the versions of a software system. Searching for the provenance of code fragments in a preceding version is comparable to tracking a clone throughout the evolution of a software system. The problem of code provenance analysis is a special case of clone evolution analysis [71].

Code provenance analysis are becoming more and more popular in the software clone community. Recently, Godfrey et al. [48] published a position paper in which they highlight some key issues related to the problem of code provenance. They recommend the development of simple and lightweight techniques capable of reducing the search space of the potential origins of candidates pieces of code. Other relevant literature on provenance analysis include [26, 44]. Kim et al. [65] proposed the first work on software clone evolution. They analyzed clone classes and defined patterns of clone evolution. Using two java systems and the CCFinder clone detection tool, they performed a case study of the evolution of clones in software systems and concluded that at least half the of clones in a software system are eliminated within eight check-ins after their creation. A systematic review of clone evolution studies is presented in [95]. The work presented in this paper differs from previous work in the way that we track the provenance of all code fragments in a version instead of searching for the provenance of clones only. The next section elaborates more on the details of our technique.

12.4 Problem, Algorithm, and Method

12.4.1 Definition of the problem

The goal to achieve is to infer the underlying file structure modifications between two versions of a system. There are three file structure modification primitives:

- *Add*: A new file is added in version N+1
- *Delete*: A file from version N is deleted in version N+1
- *Move*: A file from version N has been moved in version N+1

Conceptually, a *move* can be represented as a succession of a *Delete* of the original file and an *Add* of a file which is a copy of the original file with a different name. Indeed, Version Control Systems represent a *move* using these two primitives. In these systems, raw *Add* and *Delete* are either part of a *Move* or strict *Add* or *Delete*. To identify *Add* and *Delete*

operations properly, it is necessary to first identify the *move* operations. Thereafter, the remaining *Add* and *Delete* are the true *Add* and *Delete* operations. Thus, the important task in inferring repository file structure modifications is the detection of the *move* operations.

Assuming the above, the rest of this section discusses only the algorithm to track *move* operations. It follows naturally that the detection accuracy of the *Add* and *Delete* is directly dependent on the accuracy of the *move* identification step. Consequently, the evaluation will also only focus on the *moves*.

As already mentioned in section 12.3, the technique is an extension over [79]. The key element in this extension is the use of a nearest-neighbor search instead of range-query search. Both operations are faster if executed in a metric tree rather than using a brute-force approach. Considering this fact, before introducing our algorithm called Move Inferring Algorithm (MIA), let us review some basic concepts about the metric tree data structure.

12.4.2 Metric tree definition and construction

Since the clone detection operation relies on finding a set of fragments with a certain distance property, it follows from [78, 79] that it is worth using a specialized data structure to optimize the search space. The metric tree, originally presented in [112], and then notably used in [25, 28], is well suited for this task. Other structures, such as the kd-tree and the cover-tree might be worth exploring, but since the metric tree supports arbitrary metrics, it is best suited for clone detection as precised in [78, 79].

Using a metric tree restricts the similarity measures allowed to the set of distances satisfying the metric axioms. The following is a brief reminder of those axioms:

$$\delta(x, y) \geq 0 \ (non\ negativity) \tag{12.1}$$

$$\delta(x, y) = \delta(y, x) \ (symmetry) \tag{12.2}$$

$$\delta(x, y) = 0 \Leftrightarrow x = y \tag{12.3}$$

$$\delta(x, y) + \delta(y, z) \geq \delta(x, z) \ (triangle\ inequality) \tag{12.4}$$

Many common distances like the Jaccard distance, the Levenshtein distance, the Euclidean distance and the l_i norm family satisfy these axioms. Colloquially, they are called distances even though they are metrics. Also, in this paper distance will always be used in the same sense as metric.

Nodes in the metric tree contain one or two elements. For similarity analysis, these elements may be code fragments such as files, classes, methods or any mapping of these fragments to other representation such as frequency vectors. The variables f and f' in the

following figures always represent a fragment or a representation of a fragment, according to the context. A node containing only one element is a leaf. A node containing two elements may be a leaf or not. Nodes with two elements, called pivots, split the search space into four regions according to the distance d between the two pivots. These four regions themselves are individually nodes that contain up to two pivots and recursively divide the space. Edges in the tree link nodes to their four children regions.

The metric tree supports three important primitives: $insert(f)$ and $rangeQuery(f, \epsilon)$ and $findnn(f)$. The $insert(f)$ primitive takes a fragment and inserts it in the tree. The primitive $rangeQuery(f, \epsilon)$ takes a fragment f and a real number ϵ as parameters and returns the set of all fragments f' in the tree for which $\delta(f, f') \leq \epsilon$, as follows:

$$rangeQuery(f, radius) = \{f^{'} |\ \delta(f, f^{'}) \leq \epsilon\} \tag{12.5}$$

The $findnn$ primitive searches for the nearest-neighbor of a fragment f in the tree, i.e.:

$$findnn(f) = f' |\ \delta(f, f') = min(\delta(f, f_i)) \forall f_i \in F \tag{12.6}$$

An outline of the *insert* primitive is presented in Figure 12.1.

In the insertion algorithm, *node* contains fragment x and y and the distance d between x and y. Line 2 of the algorithm starts by initializing a variable $target$ with the node n_0 assumed to be the root of the tree. The variable $target$ represents the node in which we will insert the new fragment. The main loop in the algorithm will assign different nodes to $target$ as the algorithm progresses. The first two steps of the loop at lines 5 and 6 compute the distance of f with the two fragments already assigned to the node, called the pivots. From lines 7 to 21, the distance of f to the two pivots is compared to the distance between the two pivots. A region is selected according to criteria based on those distances. The loop continues to search nodes until it finds a node for which at least one pivot is undefined. Lines 22 to 27 then check which of the pivot is undefined and setup the node accordingly with the new fragment.

12.4.3 Building the token frequency vectors

To compute fast matching, it is required to find a proper representation of code fragments to insert in the metric tree. We chose to represent each fragment with a frequency vector of its tokens n-gram. The proposed algorithm combines different known ideas for clone detection along with new ones. Token-based clone detection was proposed by [60]. Other authors [20, 104] have since used tokens with different algorithms and token manipulation is now widely used by many tools at different step of the process of clone detection. Code metric

```
1: insert (f)
2: target = n_0
3: region = 0
4: while target.d ≠ UNDEFINED do
5:    d_1 = δ(target.x, f)
6:    d_2 = δ(target.y, f)
7:    if d_1 < target.d then
8:       if d_2 < target.d then
9:          region = 0
10:      else
11:         region = 1
12:      end if
13:   else
14:      if d_2 < target.d then
15:         region = 2
16:      else
17:         region = 3
18:      end if
19:   end if
20:   target = target.c[region]
21: end while
22: if target.x ≠ UNDEFINED then
23:   target.x = f
24: else
25:   target.y = f
26:   target.d = δ(target.x, target.y)
27: end if
28: return
```

Figure 12.1 Insertion algorithm in metric trees

based clone detection was introduced by [86] and developed in [88, 89]. The idea of code metric clone detection is to choose software syntactic features, such as statements, branching instructions, function calls, etc., and to build a vector for which each dimension is a specific feature. The value of each vector component is the frequency of the corresponding feature. Syntactic analysis is first done to compute the frequencies to then build the vectors. These are then compared using a similarity criterion, such as the Euclidean distance, cosine distance, etc. The original technique of [86] used space discretization for clone clustering. The new algorithm presented in this paper combines token analysis and code metric to create a vector: it builds vectors of token frequencies. It is not limited to the use of single tokens, but can also be extended to n-grams which we call windowed-tokens. With the new vectors, similarity is computed according to the Manhattan distance, also known as the l_1 metric.

To build these vectors, the first step of the algorithm is to extract the tokens from the software source code. This is done with a lexical analyzer on a file basis. Using lexical analysis instead of syntactic analysis has some advantages. It is faster, first, since most of the times it relies only on regular expression matching instead of context-free grammar matching, and second, it is also usually easier to write a lexer than a parser.

The second step uses the extracted tokens from the files to build frequency vectors. The base case is to use single tokens or windowed-tokens of length 1. A unique *ID* number corresponding to its corresponding dimension in $\mathbb{R}^n$ is assigned to every different token type. The tokens *ID* are generated dynamically. Each time a new token type is encountered, the next available *ID*, starting with *ID* 0, is assigned to the newly discovered type. After the token type has been identified, the corresponding vector component is incremented by one. Every component in the vector has an initial value of 0. For better memory storage, the frequency vectors are not stored in a vector-array data structure but rather in a hash table. Since code fragments have a frequency of 0 for most token types, the hash table will reduce memory usage. This may not be trivially apparent, but if we allow windowed-tokens of length 2 or more, storing data in vector-arrays is not an option because of storage capacity. For example, in a language with 200 token types, the required array length to store every components explicitly is 200 for window length 1, 19 900 for length 2, and 1 313 400 for length 3. In general, the vector length is described as:

$$\frac{t!}{(t-l)!} \tag{12.7}$$

where t is the number of token types in the language and l the window length. This equation is of course growing exponentially with respect to the length of the window and thus compels for a better memory storage. Since to any fragment there cannot be more

token types associated than its number of tokens, the hash table will use a storage linear in the number of tokens.

To extend the base case to a window length above 1, the same procedure is used but token type identifiers are assigned on an n-gram basis. For example, let the tokens of a language be $\{A, C, G, T\}$, and let s_0 be:

$$s_0 = ATGCGTCGGGTCCCAG \tag{12.8}$$

a random string. With a window of length 1, *ID* assignment would be:

$$(A, 0)\ (T, 1)\ (G, 2)\ (C, 3) \tag{12.9}$$

and s_0 frequency vector v_{s_0}:

$$v_{s_0} = (2, 3, 6, 5) \tag{12.10}$$

With a length 2 window, *ID* assignment would be:

$$(AT, 0)\ (TG, 1)\ (GC, 2)\ (CG, 3)\ (GT, 4) \tag{12.11}$$

$$(TC, 5)\ (GG, 6)\ (CC, 7)\ (CA, 8)\ (AG, 9) \tag{12.12}$$

and s_0 frequency vector v_{s_0}:

$$v_{s_0} = (1, 1, 1, 2, 2, 2, 2, 2, 1, 1) \tag{12.13}$$

and so forth with higher window lengths. The reader should note that the total number of token types in the second example, 10, is less than the theoretic maximum, 12. This is almost always the case with higher window lengths and thus it supports the idea that a hash table will consume much less memory than a vector-array.

The third step, after extracting the tokens from software and building their corresponding frequency vectors, is to build a metric tree [28, 78] with all the resulting vectors under a chosen distance. The l_i metric family is a natural choice. The goal of the algorithm is to be as precise as possible and to have a fine grain adjustable threshold on the similarity criterion. The l_1 metric, or Manhattan distance, is the best choice in the l_i family according to our criterion. The motivation behind this choice is a simple geometric observation. To get the finest grain threshold, the space enclosed by the distance at a specific threshold should be as small as possible. That space is of course a sphere defined by the chosen l_i metric. Using a quick

proof, it will be shown that at each step, the l_1-sphere is always smaller than the l_2-sphere. An analogous reasoning can then be applied to show that an l_i-sphere is always smaller than an l_j-sphere if $i < j$ for any integer value, making the l_1-sphere the smallest. Lets define an l_i-sphere $\mathbf{S}_{i,\delta}$ as the set $x \in \mathbb{R}^n | l_i(x - y) \leq \delta$ for a fixed n. In $\mathbb{R}^2$, the l_1-sphere $\mathbf{S}_{1,\delta}$ is thus a $\frac{\pi}{4}$ radians rotated square of side-length $\sqrt{2\delta}$ and the $l_2 - sphere$ $\mathbf{S}_{2,\delta}$ is a circle of radius δ. In $\mathbb{R}^n$, every point x in $\mathbf{S}_{1,\delta}$ has a distance to the origin equal to:

$$l_1(x, 0) = \sum_{i=0}^{d} |x_i| \tag{12.14}$$

and the distance of every point in $\mathbf{S}_{2,\delta}$ to the origin is:

$$l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{12.15}$$

Now we have:

$$l_1(x, 0) = \sum_{i=0}^{d} |x_i| > l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{12.16}$$

$$l_1(x, 0) = \sum_{i=0}^{d} \sqrt{x_i^2} > l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{12.17}$$

which holds $\forall d \geq 2$ and $x_i \neq 0$. Thus, an l_1-sphere covers less space than an l_2-sphere since there are some points in the l_2-sphere that do not belong to the l_1-sphere, but the l_1-sphere is entirely comprised in the l_2-sphere. It should be obvious that this argument holds for every metric in the l_i family. It follows that the best choice to have a fine control over the sensibility of the algorithm is l_1. One may argue that other spheres could have better semantic properties, but our personal experience suggests that the finer the control, the better the results.

The Manhattan distance, l_1 metric, between two vectors u and v is defined as:

$$l_1(u, v) = \sum_{i=0}^{d} |u_i - v_i| \tag{12.18}$$

One can recall that higher metrics require root extraction which is an expansive operation. Clearly, l_1 is the fastest to compute in the l_i family. Being the best for precision control and the fastest to compute, it is a natural choice. It can be found in many geometry textbooks that

l_1 satisfies the metric axioms (non-negativity, symmetry, identity and triangle inequality). Fulfilling such axioms allows us to use it in a metric tree.

However, to include the impact of the fragments relative size, it is best to normalize the metric. Thereafter, all queries will be specified with a real number in the interval $[0, 1]$. The chosen normalized Manhattan distance is the following:

$$\epsilon(u, v) = \frac{\sum_{i=0}^{d} |u_i - v_i|}{\sum_{i=0}^{d} max(u_i, v_i)} \tag{12.19}$$

Incidentally, this normalization of the Manhattan distance coincides with the Jaccard distance of the two sets, under certain hypotheses. Other Jaccard distances could be defined over the sets of fragment tokens, but this one suits our needs better since it is derived from the Manhattan distance.

12.4.4 Using a Nearest-Neighbor approach

Contrary to the approach presented in [78], this paper relies on finding the nearest-neighbor instead of performing a range-query to find the closest match. For the *move* retrieval problem, we need to search for a file that is likely to have generated the moved file. Between the time the file is originally moved and the time we try to retrieve its generator, the file might be altered in a way that it is no longer very close to the original. Thus, using a range-query to find multiple close candidates could result in missing those far-away *moves*. However, it is very likely that even if a file is deeply modified, the file that is the closest to it in the search space is the more likely to have generated it, no matter the distance. This problem is solved by finding the file's nearest-neighbor. Figure 2 outlines the procedure in a metric tree.

The algorithm is split-up in two core parts. Lines 1 to 16 compute the distance from the query to the pivots of the current node. It then selects the best match as the new nearest-neighbor if that match has a distance smaller than the current known nearest-neighbor. The rest of the algorithm recursively traverses each node that may have better candidates. Each condition tests whether or not it is possible to find a better match in the sub-tree rooted at that node. The criteria to visit each region are summarized in Table 12.1.

Combining the metric tree, the frequency vectors and the nearest-neighbor search, we can now describe the Move Inferring Algorithm (MIA) shown in Figure 12.3. The procedure is straightforward. In MIA, a code fragment is a file. MIA is provided with two arguments: the *source*, which contains all files from a system at version N, and the *destination*, which contains all files at version N+1. It is assumed that every file has already been analyzed and transformed into frequency vectors. MIA iterates over all files in *destination* and queries the

Table 12.1 Criteria for region selection in the findnn primitive.

Region 1	$\delta(x, f) < \epsilon + d \wedge \delta(y, f) < \epsilon + d$
Region 2	$\delta(x, f) + \epsilon \geq d \wedge \delta(y, f) < \epsilon + d$
Region 3	$\delta(x, f) < \epsilon + d \wedge \delta(y, f) + \epsilon \geq d$
Region 4	$\delta(x, f) + \epsilon \geq d \wedge \delta(y, f) + \epsilon \geq d$

metric tree of *source* to find its nearest-neighbor in it. Then, it checks if the nearest-neighbor path exists in *destination* and if the path of the query fragment is in *source*. If not, MIA indicates that the file added in *destination* has a closely related file in *source*, but that this file no longer exists. This strongly indicates the presence of a *move*; the pair of paths is added to the results. At the end, MIA returns the list of all inferred *moves*.

12.5 Experimental setup

12.5.1 Analyzed systems and computational equipment

All computations were performed on an Intel Core 2 Duo 2.16MHz, with 4GB of RAM, using an SSD hard-drive under Linux Fedora 15 OS. Clone detection executable binaries were compiled using g++ version 4.6.3 with the -O3 flag.

```
1: findnn(node,f)
2: if node == NULL then
3:    return ERROR
4: end if
5: result = ∅
6: ε = ∞
7: d_1 = δ(node.x, f)
8: d_2 = δ(node.y, f)
9: if d_1 < ε then
10:   result = node.x
11:   ε = d_1
12: end if
13: if d_2 < ε then
14:   result = node.y
15:   ε = f_2
16: end if
17: if δ(node.x, f) < ε + node.d ∧ δ(node.y, f) < ε + node.d then
18:   (candidate, ε_0) = findnn(node.region1, f)
19:   if ε_0 < ε then
20:      result = candidate
21:   end if
22: end if
23: if δ(node.x, f) + ε ≥ node.d ∧ δ(node.y, f) < ε + node.d then
24:   (candidate, ε_0) = findnn(node.region2, f)
25:   if ε_0 < ε then
26:      result = candidate
27:   end if
28: end if
29: if δ(node.x, f) < ε + node.d ∧ δ(node.y, f) + ε ≥ node.d then
30:   (candidate, ε_0) = findnn(node.region3, f)
31:   if ε_0 < ε then
32:      result = candidate
33:   end if
34: end if
35: if δ(node.x, f) + ε ≥ node.d ∧ δ(node.y, f) + ε ≥ node.d then
36:   (candidate, ε_0) = findnn(node.region4, f)
37:   if ε_0 < ε then
38:      result = candidate
39:   end if
40: end if
41: return (result, ε)
```

Figure 12.2 Nearest-neighbor algorithm in metric trees

1: **MIA** $(source, destination)$
2: $moves = \varnothing$
3: **for all** $f \in destination.fragments$ **do**
4: $nn = findnn(f, source.tree.root)$
5: **if** $nn \notin destination.file \wedge f \notin source.file$ **then**
6: $moves = moves \cup (nn, f)$
7: **end if**
8: **end for**
9: *return moves*

Figure 12.3 Move inference algorithm (MIA)

Table 12.2 Systems versions statistics

System	# Version	Date	# Files	LOCs	# Moves	# True Moves	# Ghost Moves
JHotDraw	8	2007-01-10 - 2011-01-09	1457-1665	217 372-281 082	299	186 (62%)	113 (38%)
Tomcat	19	2009-06-03 - 2012-04-05	1526-1681	402 843-433 148	19	2 (11%)	17 (89%)
Adempiere	6	2007-07-26 - 2010-06-14	3623-4217	652 261-1 186 149	847	766 (91%)	81 (9%)

The systems on which MIA was evaluated are presented in Table 12.2. The testbench is comprised of the Java web service manager Tomcat, the drawing application JHotDraw and the enterprise resource management Adempiere. For Tomcat and JHotDraw, we identified the release dates of many consecutive versions and downloaded the corresponding source code from their repository. For Adempiere the same procedure was followed, except for the first version, for which we had to infer a release date for fictitious versions, called 2.x.x and 2.y.y. It was necessary to do so because many *moves* in Adempiere didn't seem to coincide with one of the later releases and the history of early releases is incomplete. Nevertheless, many repository commits were done between 2.x.x and 2.y.y and this should maintain the data validity. All systems have Subversion (SVN) repositories which may be found at [1, 2, 4].

12.5.2 Computing the nearest-neighbor

For each system, the nearest-neighbor of every file was computed using the algorithm presented in section 12.4.2. For comparison purposes of our results, we actually computed two nearest-neighbors using a slight variation of the presented approach. We first computed the nearest-neighbor using the token *ID*, and then computed the nearest-neighbor using the token *image*. Although the token *ID* is usually the only representation used for clone detection, textual similarity might be more accurate for the search of *moves*, we decided to include results using token *image*. Both variants have been evaluated and compared.

12.5.3 Building the repository oracle using VCSMiner

We use our framework VCSMiner to extract file movements information from the source code repository of each subject system. VCSMiner extracts commit information from source code repositories (*e.g.,* CVS, SVN, and GIT) and stores the information into a database. We query this database to identify *moves* of files across the revisions of our studied systems. For VCSs such as CVS which reports *moves* of files implicitly, VCSMiner compares MD5 hashes of files to identify file movements. Using the extracted *moves*, we built an oracle for our problem. That oracle excludes every *move* outside the trunk of the repository. That experimental design choice was made to avoid interference between the many versions and the *tags* and *branches* directories. Since *tags* and *branches* are partial clones of the trunk directory, finding the nearest-neighbor in the whole repository could lead to find the nearest-neighbor in a random version, because there is no way for MIA to make the difference between identical files in two different paths. This choice does not alter the claim to identify *move* operations between versions, since the *tags* and *branches* directories are not part of the current version. Consequently, their exclusion is a reasonable choice.

12.5.4 Categorizing the oracle moves

Since the experiment is version-based instead of commit-based, it might happen that *moved* files cannot be identified as such because of file creation and deletion. We classified the reported repository *moves* in the following two categories:

- *True move* : The original file is in version N, but not in N+1, and the resulting file after the *move* is still in version N+1, but was not in N
- *Ghost move* : Both the original and the resulting file are missing from versions N and N+1

For the purpose of computing precision and recall on *move* identification, we consider only the oracle *moves* tagged as *true move*. The *ghost move* operation is impossible to identify when using source code of released versions only, thus precision and recall for this primitive is not reported. However, the total number of *moves* along both with the number of identified *true move* and *ghost move* is reported for each system. The reported percentage for the two different types of moves are relative to the total number of moves. From one version to another, as it is displayed in table 12.2, the number of *ghost moves* is not generally significant, and there are enough *true move* to conduct a sound experiment.

Also, if the tool reported a move that wasn't reported by the VCS, then we classified it as an *implicit move*. An example of a legitimate implicit move would be a file movement not recorded by a developer in the VCS.

12.5.5 Methodology to compute precision and recall

Using the oracle produced by tool VCSMiner and classified according to the scheme introduced in section 12.5.4, we computed the recall of MIA.

As it will be discussed in section 12.7, the tool precision couldn't be evaluated with the chosen oracle and it had to be inferred statistically.

For recall, we compared all the possible $destination - source$ pairs, including pairs with the same *source* parameter.

12.6 Results

Tables 12.3, 12.4, and 12.5 show the detailed results for each system using the two variants of the algorithm. On average, using token *image* instead of *ID* gives a better recall. It also reports more implicit *moves*. However, after a quick inspection of the reported implicit *moves* in both cases, we concluded that more false-positives were produced by the *ID*-based

Table 12.3 JHotDraw recall of inferred moves, with implicit found moves.

Versions	Recall		#Implicit moves	
	Token Id	Token Image	Token Id	Token Image
7.0.8-7.0.9	0.0	0.9286	0	4
7.0.9-7.1.0	0.0	0.8571	0	3
7.1.0-7.2.0	0.0	0.7500	0	8
7.2.0-7.3.0	0.0	1.0000	0	1
7.3.0-7.4.1	0.0	0.9912	0	17
7.4.1-7.5.1	0.0	0.8667	0	6
7.5.1-7.6	0.4587	1.0	2	0
Summary	0.0765	0.9647	2	39

algorithm. For the *image*-based algorithm, almost all pairs of $destination - source$ in implicit *moves* had the same file name at the end of the path, which was not the case for the *ID*-based version. Thus, qualitatively, the precision of the *image*-based algorithm is greater than the *ID*-based one. For this reason, we did not investigate further the results of the *ID*-based *moves* as they were clearly worse than the one reported by the *image* variant.

Table 12.6 shows that for JHotDraw and Adempiere, on average the similarity of two different *move* types is really close. The averages for Tomcat cannot be compared in a meaningful way because it only contained 2 *true moves*. Despite this small number of *true moves*, results on Tomcat are not excluded because they help evaluate a limit case for the false-positive rate: they provide an answer to the question "how many results are reported when only few are expected ?" Evaluating limit case is as important as evaluating the average one. Figure 12.4 shows a box-plot of the implicit *moves* distance distribution excluding the *moves* with distance 0.0. As the analysis of their averages already suggested, JHotDraw and Adempiere seem to share some common characteristics regarding their *moves*; it may also suggest that MIA gives coherent results independently of the system. Because verifying all the implicit moves manually was not practical due to their large number, we randomly sampled and checked a representative sample of 100 implicit moves out of all the reported implicit moves and manually verified if they were real *moves* using information from both file names and contents. We tagged each sampled *move* as a "hit" or a "miss" by assigning it a real value of 1.0 for "hit" and 0.0 for "miss". The mean of "hit" and "miss" distribution corresponds to the precision of MIA. The mean of the "hit" and "miss" distribution was found to be 0.9700. The chosen sampling allows us to compute the precision of the implicit *moves* with a confidence level of 99%, and a confidence interval of 0.0422 using the Central Limit Theorem [54]. The theorem proves sample averages to be normally distributed if the sample is random, which is the case here. Therefore, the confidence interval is built using standard

Table 12.4 Tomcat recall of inferred moves, with implicit found moves.

Versions	Recall		#Implicit moves	
	Token Id	Token Image	Token Id	Token Image
6.0.20-7.0.0	1.0000	1.0000	0	0
7.0.0-7.0.4	1.0000	1.0000	0	0
7.0.4-7.0.5	1.0000	1.0000	1	1
7.0.5-7.0.6	1.0000	1.0000	0	0
7.0.6-7.0.8	1.0000	1.0000	0	0
7.0.8-7.0.10	1.0000	1.0000	0	0
7.0.10-7.0.11	1.0000	1.0000	0	0
7.0.11-7.0.12	1.0000	1.0000	63	63
7.0.12-7.0.14	1.0000	1.0000	0	0
7.0.14-7.0.16	1.0000	1.0000	3	3
7.0.16-7.0.19	1.0000	1.0000	9	9
7.0.19-7.0.20	1.0000	1.0000	1	1
7.0.20-7.0.21	1.0000	1.0000	0	0
7.0.21-7.0.22	1.0000	1.0000	0	0
7.0.22-7.0.23	1.0000	1.0000	0	0
7.0.23-7.0.25	1.0000	1.0000	0	0
7.0.25-7.0.26	1.0000	1.0000	0	0
7.0.26-7.0.27	1.0000	1.0000	2	2
Summary	1.0000	1.0000	79	79

Table 12.5 Adempiere recall of inferred moves, with implicit found moves.

Versions	Recall		#Implicit moves	
	Token Id	Token Image	Token Id	Token Image
2.x.x-2.y.y	0.8571	1.0000	7	5
2.y.y-3.3.0	0.8490	1.0000	250	131
3.3.0-3.3.1	1.0000	0.9972	4	3
3.3.1-3.4.0	1.0000	1.0000	0	0
3.4.0-3.5.3	1.0000	1.0000	1131	970
3.5.3-3.5.4	0.0455	0.9091	31	22
3.5.4-3.6.0	1.0000	1.0000	3	0
Summary	0.8217	0.9844	0	1131

equations also provided in [54]. Building this interval does not require a specific test. The lower bound on precision of MIA is 0.92788 (*i.e.*, $0.9700 - 0.0422$). We combined the average recall obtained over all our subject systems (*i.e.*, 0.98) with the lower bound of the precision (*i.e.*, 0.92788) to compute the F-value. We found the F-value to be 0.9533. Hence, we conclude that MIA achieves a very high accuracy. We answer our research question *RQ1* positively. Our proposed technique MIA can successfully recover the revision history of a software system.

Table 12.6 Average similarity of True Move and Implicit Move for all systems

System	True moves average distance		Implicit moves average distance	
	Token *ID*	Token *image*	Token *ID*	Token *image*
JHotDraw	0.0509	0.1651	0.2336	0.1951
Tomcat	0.1997	0.2546	0.0261	0.0359
Adempiere	0.0687	0.0812	0.0921	0.0930

Figures 12.5, 12.6, and 12.7 display the running time of the algorithm, including the tokenizing preprocessing step. with respect to the lines of code in each version. The longest execution time is 600 seconds from an Adempiere version over 1 MLOC, but this data seems to be an outlier with respect to the other points gathered for Adempiere. After investigation, that point possibly had more cache misses in hard-drive read operations than the others and the occurrence seems to be random and not linked to that particular version.

12.7 Discussion

The analysis focused on finding the closest code fragment from one version to what is the closest related fragment in another version. This is the first similarity analysis that uses a nearest-neighbor approach with a continuous distance instead of a binary matching without an underlying explicit distance. Using this proximity query allowed to find the most likely generator in version N of a fragment in version $N + 1$. According to the results presented in section 12.6, it compares well to the *move* information provided by the repository of our system. It also supplies more information about implicit *moves* not recorded in the repository. Such conclusions establish the soundness of the algorithm to infer file structure modifications. As noted in section 12.2, this implies the possibility of recovering repository information for systems that do not have one or to fix existing one.

It was unexpected that the implicit *moves* would account for such a large proportion of all the *moves*. This makes the assessment of the precision complicated. By looking closely at our results, we observed that a large proportion of the implicit *moves* has a very small

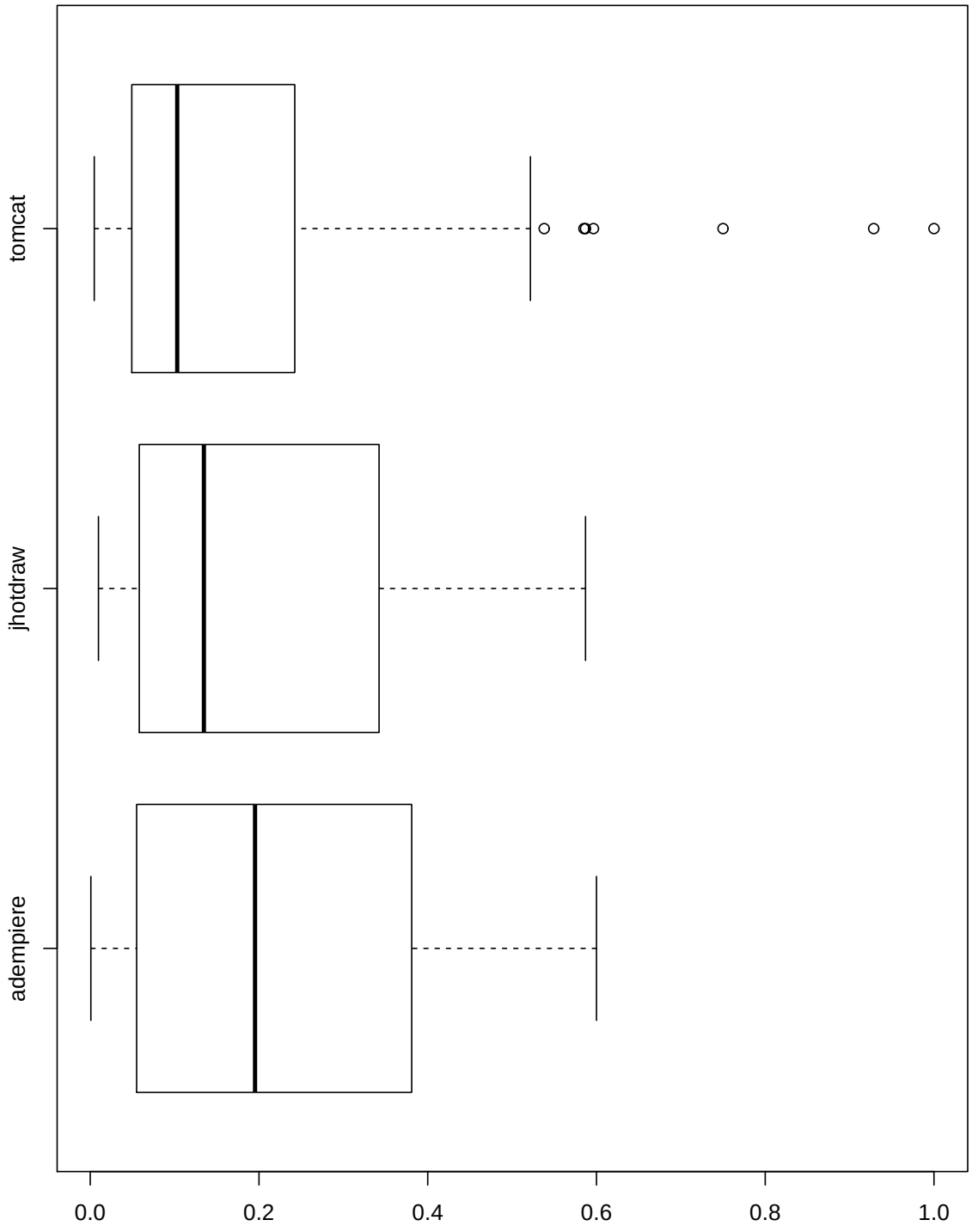


Figure 12.4 Box-plot of distance distribution of inferred moves excluding identical matches for all systems

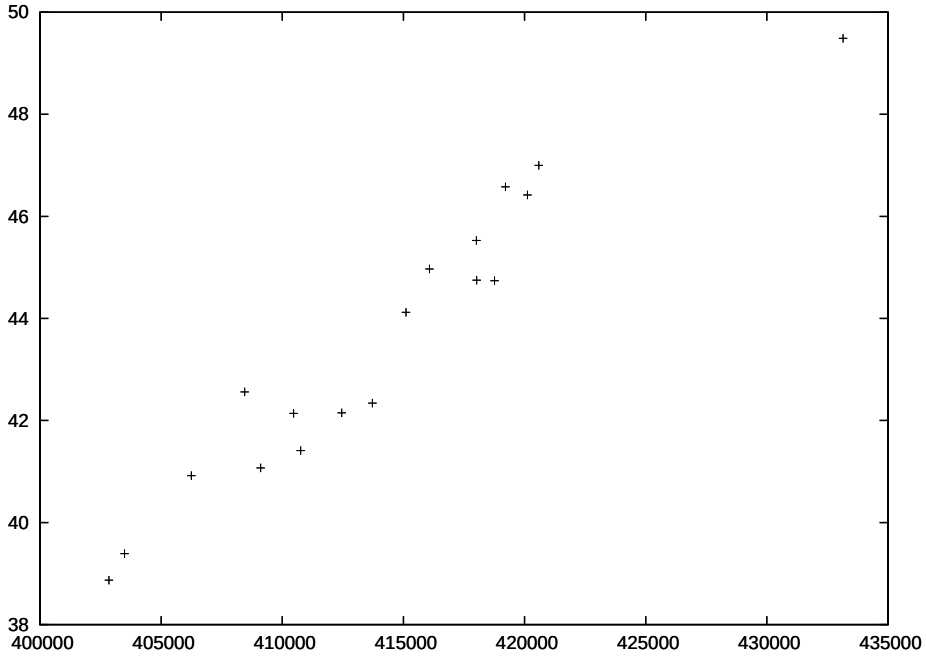


Figure 12.5 Execution time (s.) of the Nearest-Neighbor clone detection for Tomcat with respect to the number of lines of codes in each version

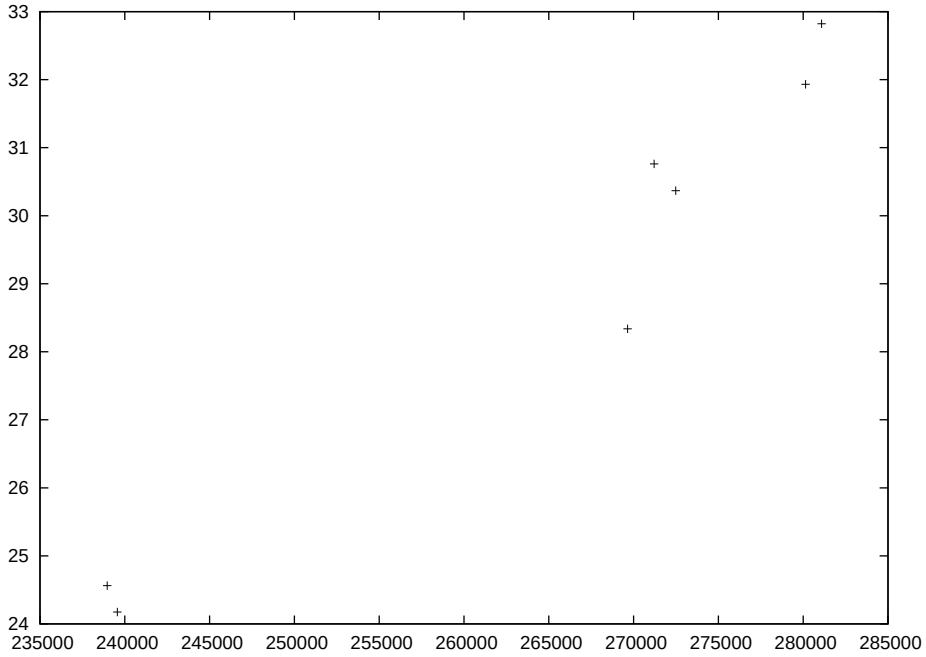


Figure 12.6 Execution time (s.) of the Nearest-Neighbor clone detection for JHotDraw with respect to the number of lines of codes in each version

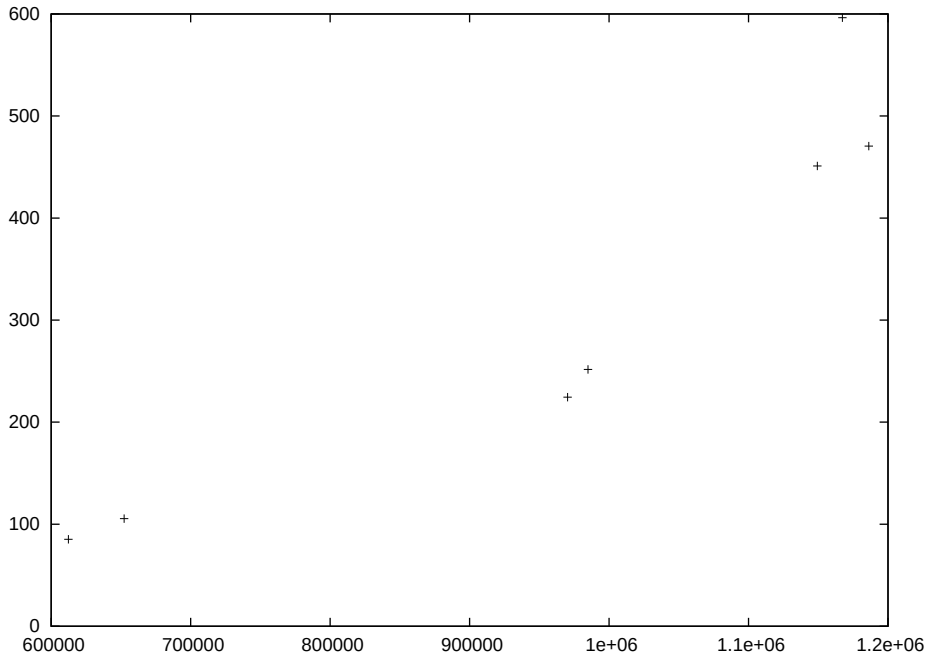


Figure 12.7 Execution time (s.) of the Nearest-Neighbor clone detection for Adempiere with respect to the number of lines of codes in each version

distance. Assuming our distance is accurate, it is more likely that these *moves* were done without the developers recording them in the VCS. If that is true, the information provided by the repositories may not be reliable as a complete oracle. This is the principal threat to validity to our experiment. Assessing recall like what was done in section 12.6 is necessary but not sufficient to prove true recall to be high because some *moves* seem to be missing from the oracle. Also, the proportion of implicit *moves* seems too large to do a precision analysis, since with reasonable arguments our results seem precise. Therefore, our study indicates the need for more data to compare repositories information to draw more conclusions on the general pattern of *moves* recording. Nevertheless, our experimental results show with strong evidences that MIA achieves results at least as good as the information provided by the existing repositories, if not better.

All experiments were done on systems of size above 100 KLOC, with the biggest of size 1.5 MLOC. Execution time of the clone detection step never exceeded 10 minutes for token *image* based analysis, and the lexical analysis step stayed below 5 minutes. Even if the technique may not be applied in real time, it is reasonable to say that it is scalable, since it runs under 10 minutes for systems of few MLOCs. As this experiment did not have any previous knowledge to enhance the searches, it was sound to use a nearest-neighbor approach even if it is time-expensive. However, as it turns out that the inferred *move* information is

very akin to clone detection information, it may be possible to use a much faster range-query to find generator candidates and retain the closest result afterwards.

Finally, the move inference problem has proven to be solvable using techniques from clone detection. Moreover, observing the evaluation methodology proposed here might give a clever insight to challenge the clone detectors evaluation problem. Even if the proposed oracle seems to have flaws in its construction, it provides an interesting set of naturally occurring clones. Testing against the *move* reference might not be necessary and sufficient, but it can be an interesting indicator about tools capacity to detect small-gap and large-gap clones. Despite the evident flaws we already outlined, disregarding it would mean to eliminate a naturally occurring set of clones which might serve as a reference set.

Under the hypothesis that our tool is precise, the many *moves* identified that were not present in the repository information suggest that the technique may be used to suggest to developers to record certain modifications. It might be integrated in a version control system to automate the process of identifying *moves* between commit instead of asking developers to provide the information manually. This might help storing more meaningful information about software evolution in repositories. Automating the process of recording file structure modifications may also help developers to manage local sandboxes with greater ease and avoid file operation mistakes that are generally arduous to fix in VCS.

Moreover, the information contained in version control systems or repositories, such as Subversion and GIT, may also be used to deduce more information, as described in [18]. Repository information may also be used to provide version history editing as suggested by [13].

12.7.1 Threats to validity

This section discusses the threats to validity of our study following the guidelines for case study research [117].

Construct validity threats concern the relation between theory and observation. In this work, the construct validity threats are mainly due to measurement errors. We extracted *moves* information from repositories but discovered that information is probably incomplete. This limits the interpretation of the recall as an upper bound instead of a definitive value and makes it impossible to evaluate the precision against the oracle. The alternative to measure precision may be biased by the opinion of the expert who inspected the implicit moves. To mitigate this bias, we inspected the name of the reported file to verify whether there was a coherence pattern and indeed there was. Thus, it increases the confidence in the reported precision value.

Threats to internal validity concern our selection of subject systems, tools, and analysis

method. In [79], we measured the distortion of the Manhattan distance with the Levenshtein distance for clone detection and found the distortion to be very low in general. However, the seldom cases for which the distortion is not very low could affect our findings.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. The source code repositories of Tomcat, JHotDraw and Adempiere are publicly available to obtain the same data for the same releases. We also provided all the algorithmic versions of our code with all the software experimental parameters and hardware configuration. Moreover, we published on-line[1] the raw data to allow other researchers to replicate our study and to test other hypotheses on our data set.

Threats to external validity concern the possibility to generalize our results. On the two systems with the most *moves*, we obtained similar precision and recall and a small difference between the average distance of the oracle *moves* and the implicit *moves*. Nevertheless, the validation of our technique is limited to three open source software systems written in Java, therefore, we cannot generalize our findings to other programming languages. However, the results are very encouraging and suggest further studies on different systems written in different programming languages to make our findings more generic.

12.8 Conclusion

In this paper we have introduced a new technique to recover file structure modifications information and file movement informations in source code repositories. The technique proved to be highly precise (F-value = 0.9533) and scalable. The technique may also be applied to other reverse-engineering tasks that require file structure modification information. Further research could include, comparing the nearest-neighbor with range-query primitive as well as investigating the different proposed applications. In practice, our proposed technique can be integrated in existing VCSs to enrich their meta-data with information on *implicit moves*; easing file structure manipulations and preventing mistakes that may require fixes that are hard to handle. The oracle used in the evaluations our proposed approach provides an interesting set of naturally occurring clones that can be reuse to evaluate clone detectors.

[1] https://dl.dropbox.com/u/14931955/wcre12.zip

12.9 Acknowledgments

We would like to thank the anonymous reviewers of WCRE for their valuable feedback that helped us to improve the quality of the paper. Many thanks also go to Francois Gauthier and Anael Maubant for their many good advices on the writing of this paper. This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program and the Fonds Quebecois Nature et Technologie (FQRNT).

CHAPTER 13

ADDITIONAL RESULTS AND APPLICATIONS

The main contribution of this thesis was presented in the previous chapters within the accepted and submitted papers written on the topic. However, additional results that were not published under my lead authorship are worth presenting. Hence, this chapter will focus on summarizing these auxiliary results. The first summary, preceding the experiment undertaken in chapter 11, covers work from the inception of our collaboration with Ericsson. The second are results on cross-matching cloning information with security-based pattern traversal and flow analysis. Both series were published in the following conference proceedings:

E. Merlo, T. Lavoie, P. Potvin, and P. Busnel. Large scale multi-language clone analysis in a telecommunication industrial setting. In *Software Clones (IWSC), 2013 7th International Workshop on*, pages 69–75. IEEE, 2013

F. Gauthier, T. Lavoie, and E. Merlo. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 209–218. ACM, 2013

13.1 Clones In A Large Telecommunication Industrial Setting

Ericsson is a telecommunication company with a very large code base. At their request, we deployed within their architecture a clone detector named CLAN, which is the predecessor of the metric tree clone detector. This section reports the different clone distributions found within a code basis of around 100MLOC written in Java and C/C++. A discussion of the different lessons learned in an industrial setting is also provided.

13.1.1 Experiments

Table 13.1 All systems clone clustering

Language	User (secs)	System (secs)	Elapsed (mins)
C/C++	173.408	0.682	2:55.46
Java	8.457	0.168	0:08.69

Table 13.2 Features

Project	Java (LOC)	C/C++ (LOC)	Total (LOC)	C/C++ Parsing (secs)	C/C++ Metrics (secs)	C/C++ DP (secs)	Java Parsing (secs)	Java Metrics (secs)	Java DP (secs)	Throughput (kLOC/min)
1	0	30,590	30,590	3.98	0.25	5.69	0.08	0.13	0.25	176.82
2	0	64,816	64,816	5.74	0.22	2.82	0.06	0.22	0.12	423.63
3	208,853	1,260,818	1,469,671	77.14	3.55	92.43	9.44	31.05	48.02	337.04
4	3,759,580	11,236,372	14,995,952	922.78	268.76	2218.99	107.82	873.21	624.64	179.37
5	69,862	16,168	86,030	2.73	0.17	0.34	3.79	9.32	8.55	207.30
6	7	78,416	78,423	6.37	0.22	3.45	0.1	0.21	0.09	450.71
7	1,566,305	47,480,555	49,046,860	2766.01	901.64	11571.11	83.93	358.98	206.35	185.22
8	2,421,132	39,488	2,460,620	19.77	1.15	1.61	63.45	424.39	410.39	160.34
9	36,859	1,233,798	1,270,657	75.67	2.33	79.99	3.02	7.38	5.02	439.65
10	1,530,238	22,308,959	23,839,197	1566.42	459.4	4162.51	72.99	281.63	192.25	212.37
11	133	1,557,630	1,557,763	100.62	4.92	101.71	0.51	8.15	0.08	432.73
Total	9,592,969	85,307,610	94,900,579	5547.23	1817.56	25474.62	345.19	1,655	2,901	150.87

Eleven proprietary projects are analyzed presenting a mix of Java and C/C++ files. Table 13.2 shows the characteristics of the analyzed systems together with execution times and throughput. CLAN's clustering algorithm execution is very fast and is separately reported in Table 13.1 for all systems together.

The system architecture for the experiments is based on a virtual machine with Ubuntu Server 11.10, 1 CPU, 3GB of RAM and 200 GB of Hard Drive Disk space and relies on Apache/MySQL/PHP, Java 7 and CLAN. Processing times are reported in Table 13.2.

The "SW Similarities Portal" is the web interface available to developers to upload their code, manage their projects and visualize the results. It is written in PHP and uses a MySQL database to store project information and status. The "Project Manager Daemon" is a Java program that handles basic operations on the source code such as extraction, unsupported file removal, and backup. It also manages the CLAN processes on the projects, calls the scripts to parse the source code, extracts the metrics, and runs the visualization process. The daemon also relies on running the MySQL database to store the similarity results to update projects status.

DP alignment is executed using the implementation of an algorithm similar to that described in [68]. Clone pairs are reported as similar, if the similarity coefficient, described below, was greater or equal to 0.8. For performance reasons, alignment is not executed if one of the two fragments under comparison was larger than 200 LOC. Different size thresholds and different algorithms may be deployed for aligning larger fragments, if required.

In the following figures and diagrams, cluster size shows the cardinality of the set of mutually similar fragments.

Many clusters are only composed of two fragments, but larger clusters also exist. Average size indicates how many opportunities for re-factoring, bug fix propagation, etc. are available on average when a fragment is duplicated. Fragment size distribution shows the size of fragments larger than 6 LOCs involved in a similarity relation.

Fine grained analysis using DP matching allows the classification of clones as identical (type 1) or parametric (type 2) or different (type 3). For performance reasons, since *DPmatching* has a quadratic component, fine grained similarity has only been performed on fragments between size 7 and 200 LOC and of type similarity between 0.8 and 1.0 computed using the Jaccard coefficient between sequences of tokens. For larger files a faster but sub-optimal alignement algorithm based on *Unix diff* are used.

Identical clones are identified by a 100% image similarity, since token types and token images are identical. Parametric clones are fragments that have an identical sequence of token types, but less than 100% image similarity (due to identifiers, constants, etc.) They are good candidates for re-factoring.

13.1.2 C/C++ Results

The maximum cluster size is 8,761 and corresponds to small fragments such as constructors, destructors, initialization of variables, and so on. The average cluster size is 7.18 over 29,380 clusters. 181,756 similar fragments are detected with an average size of 82.47 LOC. 162,415 fragments show a type similarity higher than 0.8, are smaller than 200 LOC, and have an average size of 35.72 LOC. 109,837 fragments, out of these 162,415 highly similar fragments, also have a type similarity equal to 1.0 41,709 fragments, out of the 109,837 fragments with type similarity equal to 1.0, also have an image similarity equal to 1.0 and are identical clones. Their average size is 75.26 LOC. Parametric or type-2 clones have a type similarity equal to 1.0, but an image similarity different from 1.0. These are smaller, their average size is 20.74 LOC, but they are greater in number for a total of 68,128 fragments. The remaining 52,578 highly similar fragments are of type-3 or other higher clone types.

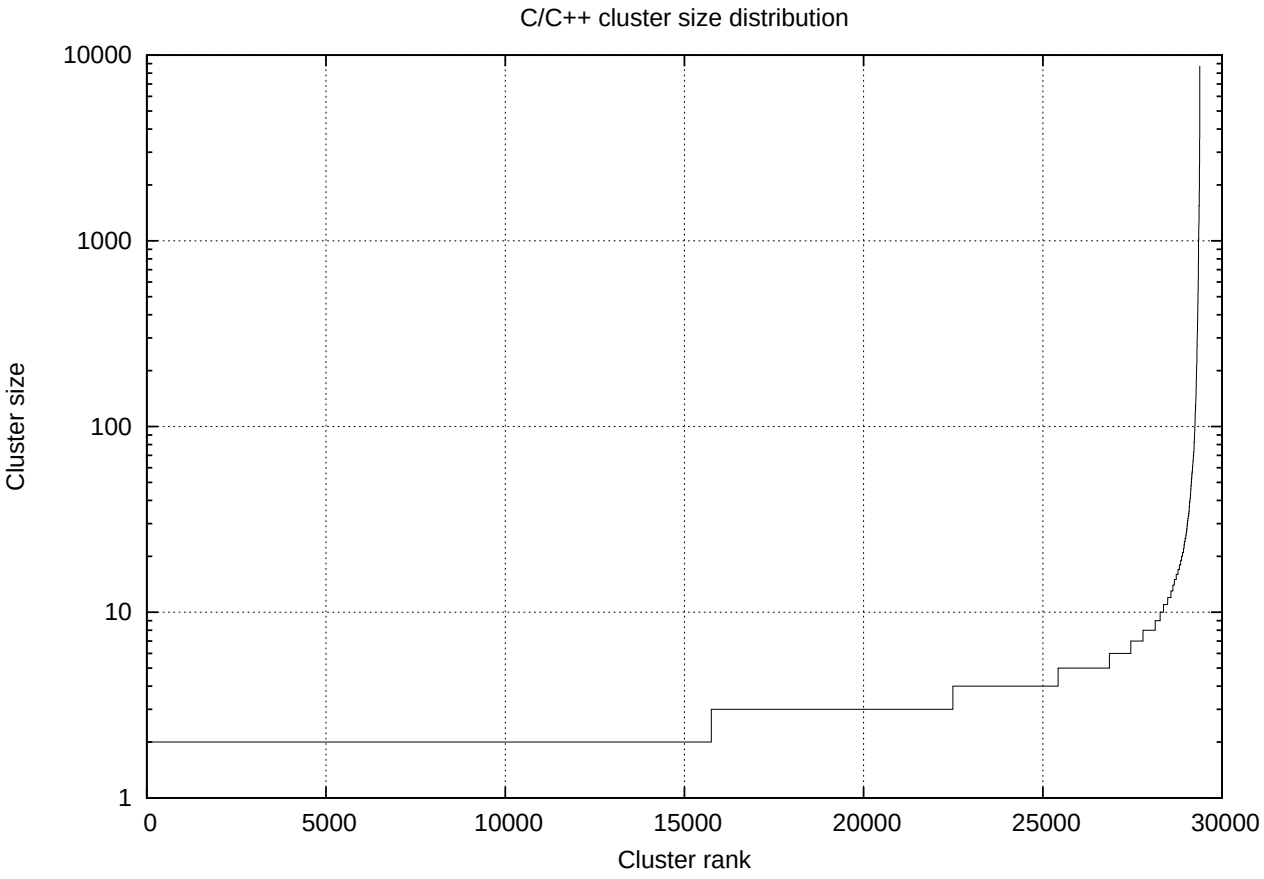


Figure 13.1 Cluster size distribution

Figure 13.1 shows the distribution of cluster sizes with respect to cluster size ranking starting from the smallest one.

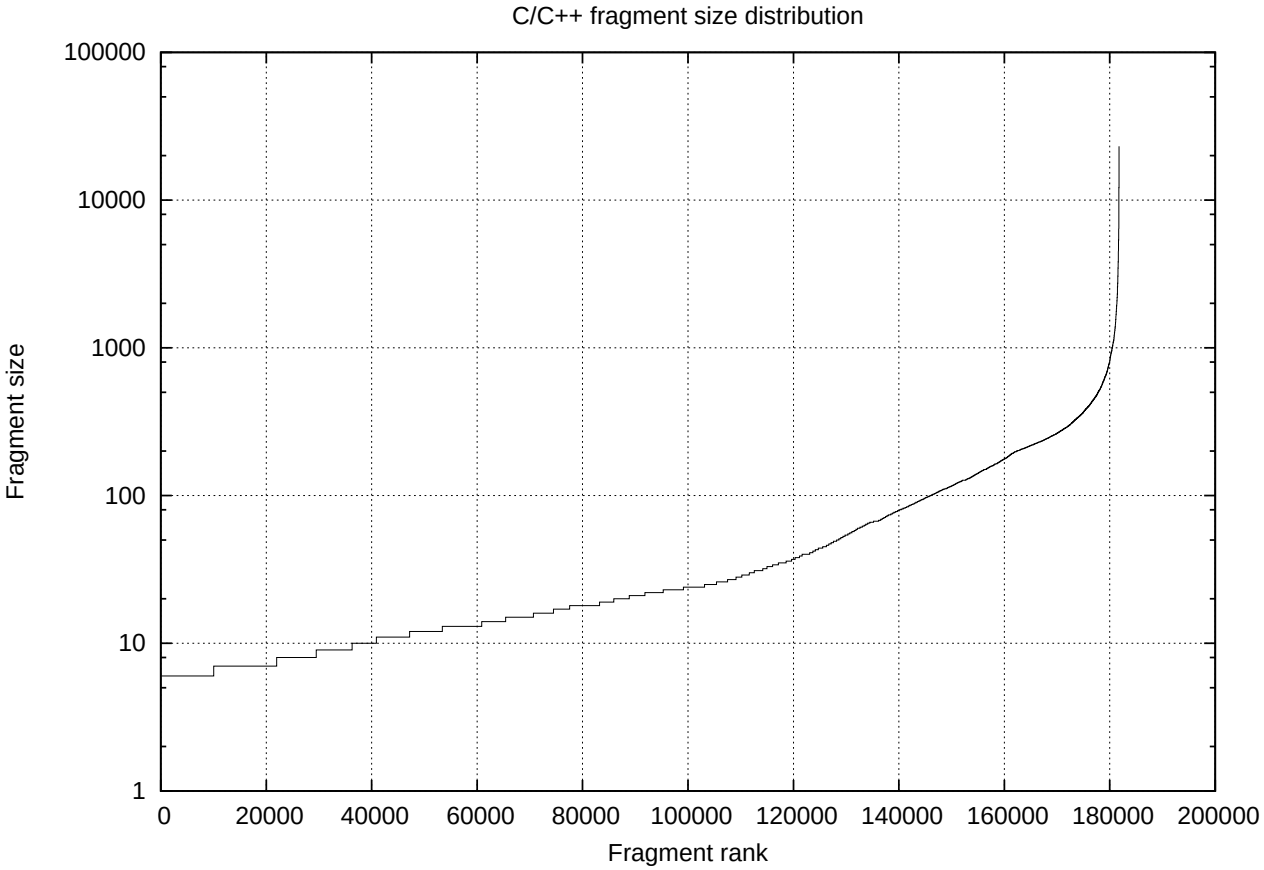


Figure 13.2 Fragment size distribution

Figure 13.2 indicates the fragment size distribution of fragments involved in a clone relation. Sizes are measured in LOCs. Many fragments are approximately tens or hundreds of LOCs, but there are some similar fragments that are tens of thousands of LOCs. Fragments are ranked in order of increasing size.

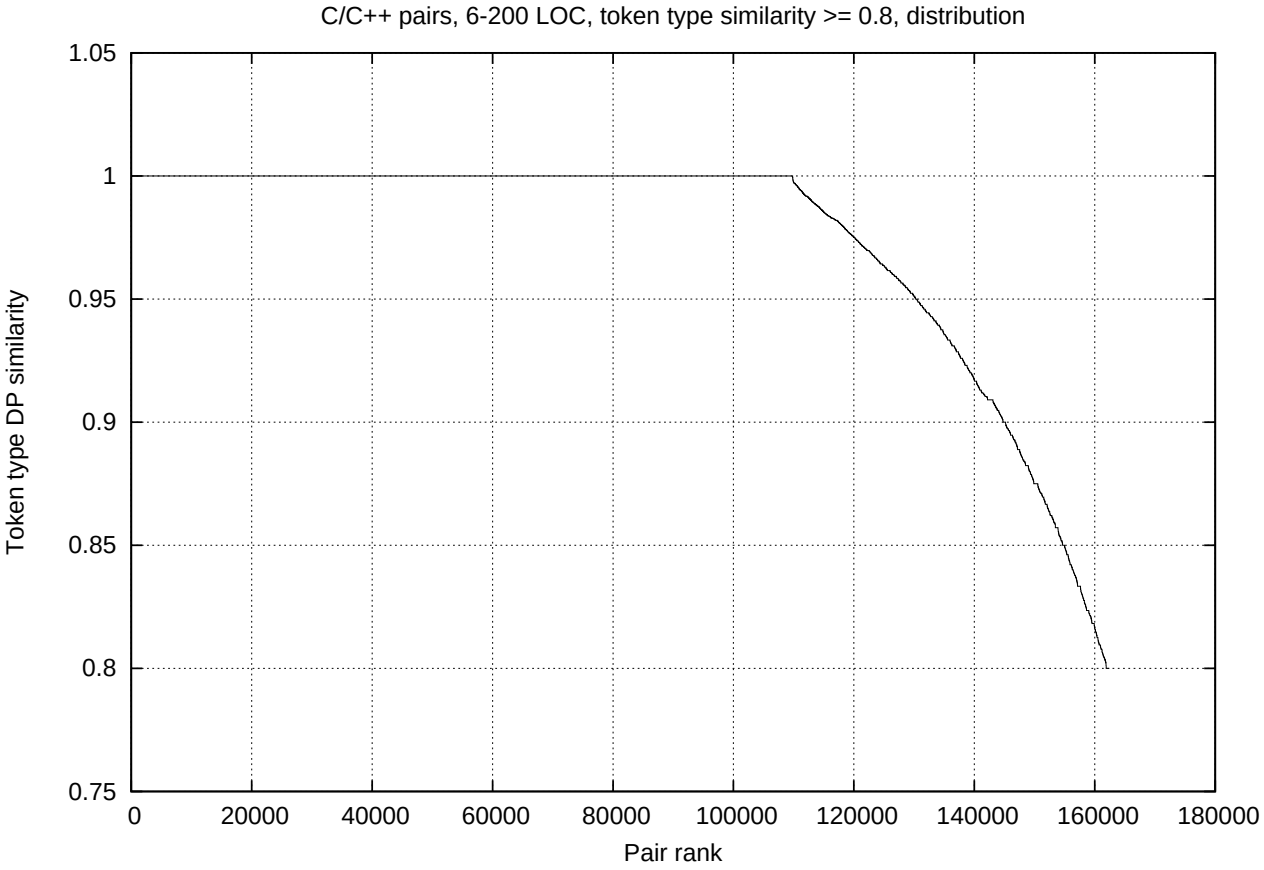


Figure 13.3 Similarity distribution

Figure 13.3 reports the distribution of token type similarity between fragment pairs in clusters.

Fine grained *DP* similarity is performed on 162,415 fragments. Pairs are ranked by descending degree of similarity, from the completely similar pairs with a similarity equal to 1.0 to the less similar pairs with a similarity equal to 0.8. In Figure 13.3, as already mentioned, 109,837 fragments out of the 162,415 fragments have similarity equal to 1.0 (types 1 and 2). The remaining 52,578 type similar fragments are of type 3 or other higher clone types.

Figure 13.4 gives more details about the distribution of totally type similar pairs, by showing the distribution of their image similarity. Pairs are ranked from the most similar images to the least similar ones, while preserving total type similarity. Image similarity ranges from 100% down to about 10%. From the 109,837 totally type similar fragments, 41,709 are

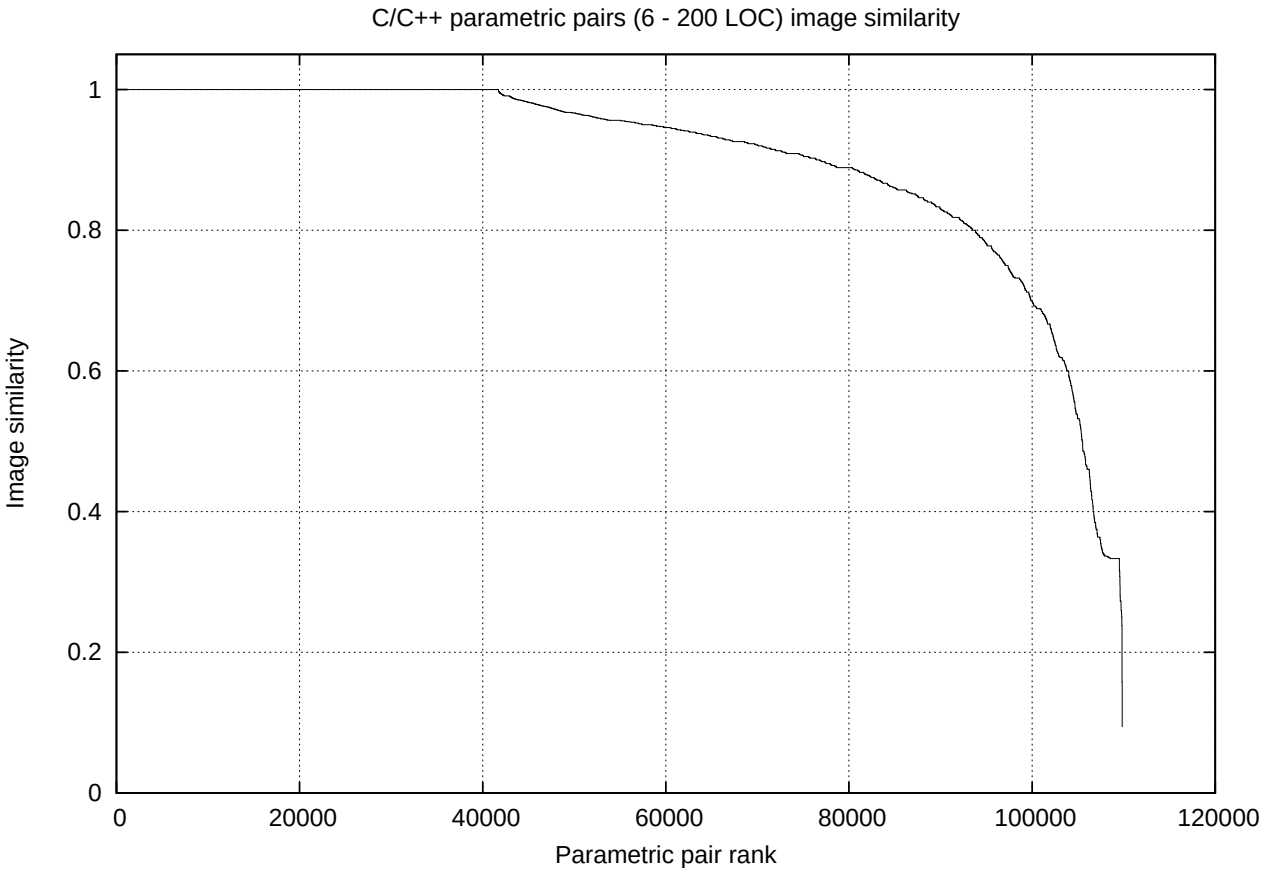


Figure 13.4 Parametric similarity distribution

also totally image similar and are identical fragments (type 1 clones) in different files. The remaining 68,128 fragments are parametric clone pairs (type-2 clones), identical by type with less than 100% image similarity (varying by degrees).

13.1.3 JAVA Results

Many clusters are only composed of two fragments, but there are larger clusters. The maximum cluster size is 613, and the average size is 3.75 over 29.866 clusters. 82,426 similar fragments are detected with an average size 136.75 LOC. 68,907 fragments show a type similarity higher than 0.8, are smaller than 200 LOC, and have an average size of 48.28. 61,297 fragments, out of these 68,907 highly similar fragments, have a type similarity equal to 1.0 and they are considered identical or parametric clones, that is of type 1 or 2. 49,109 fragments, out of these 61,297 highly similar fragments, also have image similarity equal to 1.0, have an average size of 54.82 LOC, and are identical clones. Parametric clones have type similarity equal to 1.0, but image similarity different from 1.0, are smaller (average size is 31.34 LOC) , and fewer in number (12,188 fragments). The remaining 7,610 highly similar fragments are of type 3 or other higher clone types.

Figure 13.5 shows the distribution of cluster sizes.

Figure 13.6 indicates the distribution of fragment size involved in a clone relation.

Figure 13.7 reports the distribution of token type similarity. Fine grained similarity analysis are performed on 68,907 fragments. In Figure 13.7, 61,297 fragments, out of the 68,907 similar fragments, have a type similarity equal to 1.0. The remaining 7,610 type similar fragments are of type 3 or higher.

Figure 13.8 gives more details about the distribution of totally type similar pairs.

From the 61,297 totally type similar fragments, 49,109 are also totally image similar and these are identical fragments. The remaining 12,188 fragments are parametric clone pairs.

Java clustering shows that java fragments seem to be more type similar and image similar, than those found in C/C++.

13.1.4 Discussion and Lessons Learned

Part of this discussion was published in the related paper and was co-authored with Ettore Merlo and Pascal Potvin. An extended version is presented here, with part of the original material reproduced in this section with their explicit consent.

The development of this project allowed us to learn about transferring clone detection into an industrial setting from a research laboratory perspective. The clone detection environment has been conceived for research purposes that may be somewhat different from industrial

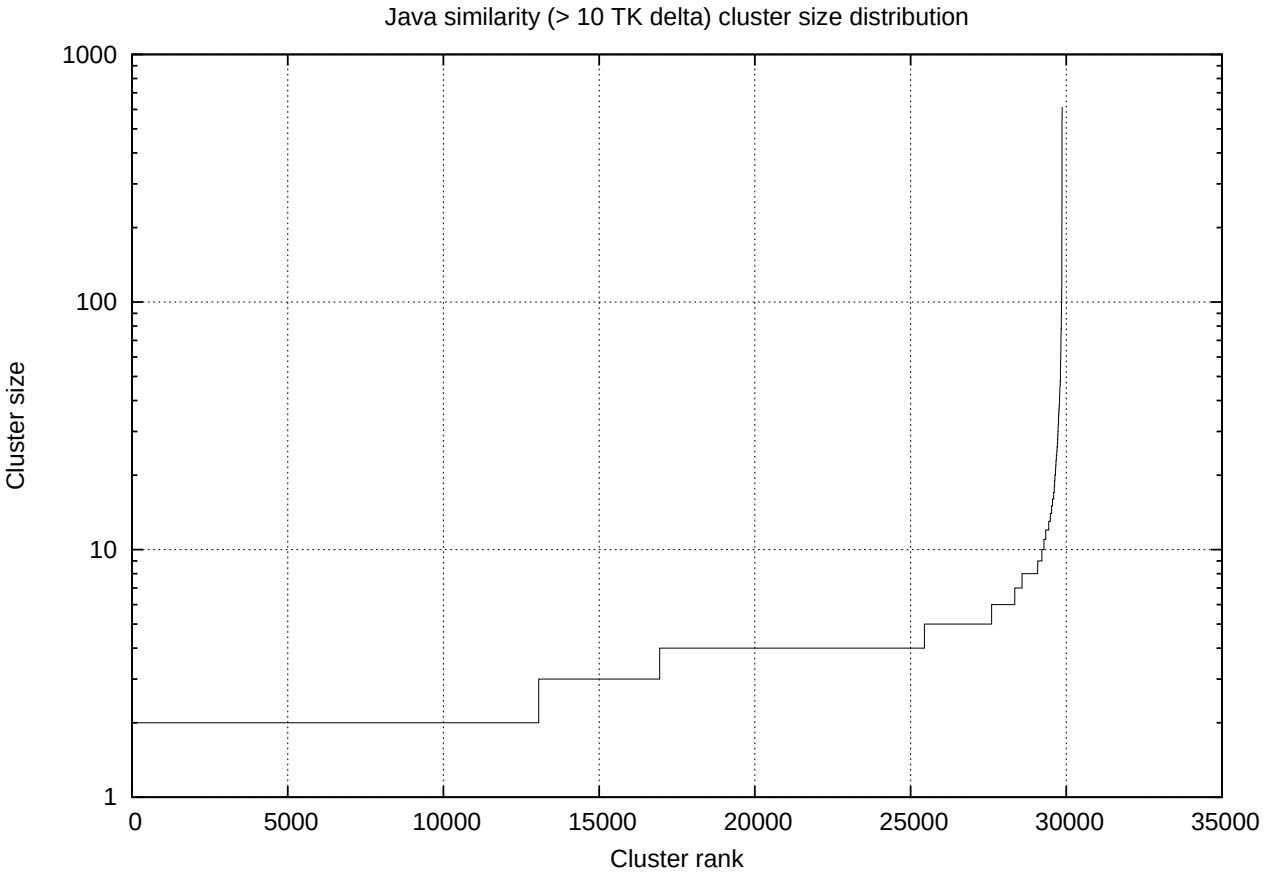


Figure 13.5 Cluster size distribution

production objectives. The process of clone detection is research oriented and reflects research requirements that may necessitate change for an industrial application. Objectives for a clone detection process from a research point of view may not be the same as the industrial objectives for a clone detection process.

13.1.5 Industrial perspectives

In order to use the *CLAN* clone detection tool within the Ericsson industrial setting some adaptation was required involving the interaction with the tool and the presentation of the results to the users. Most of the Ericsson effort went into these areas because early on it was decided to access the tool via a web interface. The source code was uploaded to a web server via the web interface, then processed by the *CLAN* tool, whose outputs were displayed back on the web interface as they became available.

Although the web interface development might seem simple, Ericsson had to invest close to two years/person to make the interface user-friendly and to display the results efficiently for the task. Understandably these were areas where very few developments had been made

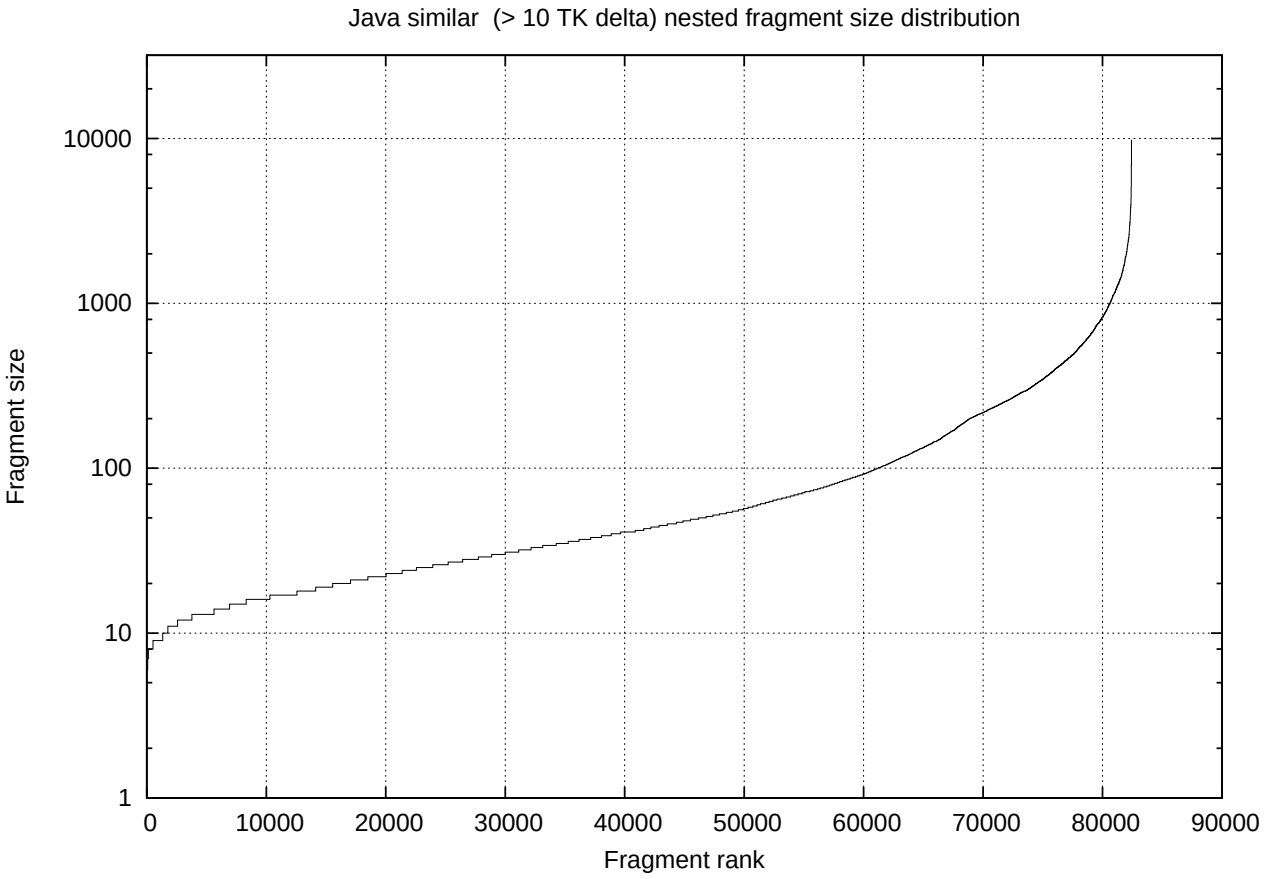


Figure 13.6 Fragment size distribution

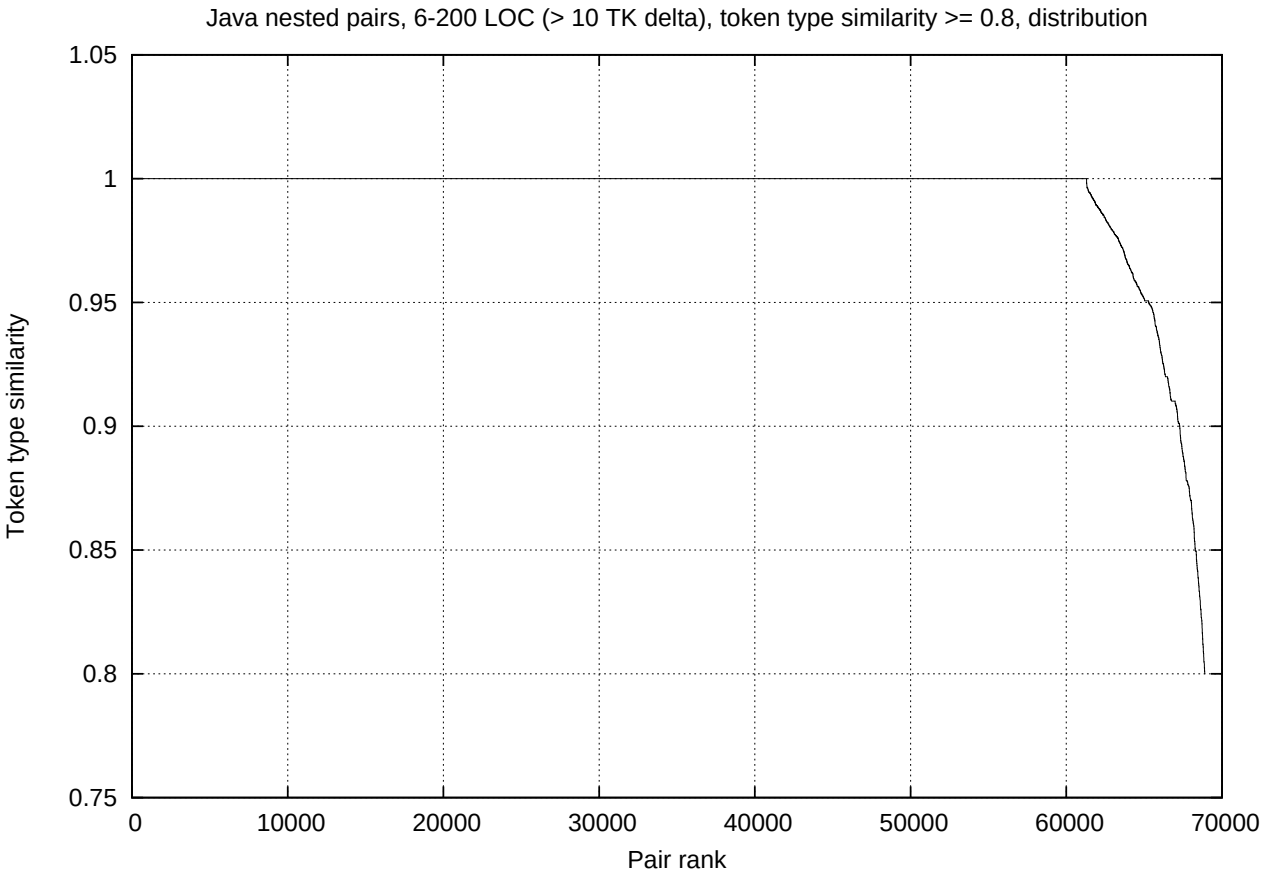


Figure 13.7 Similarity distribution

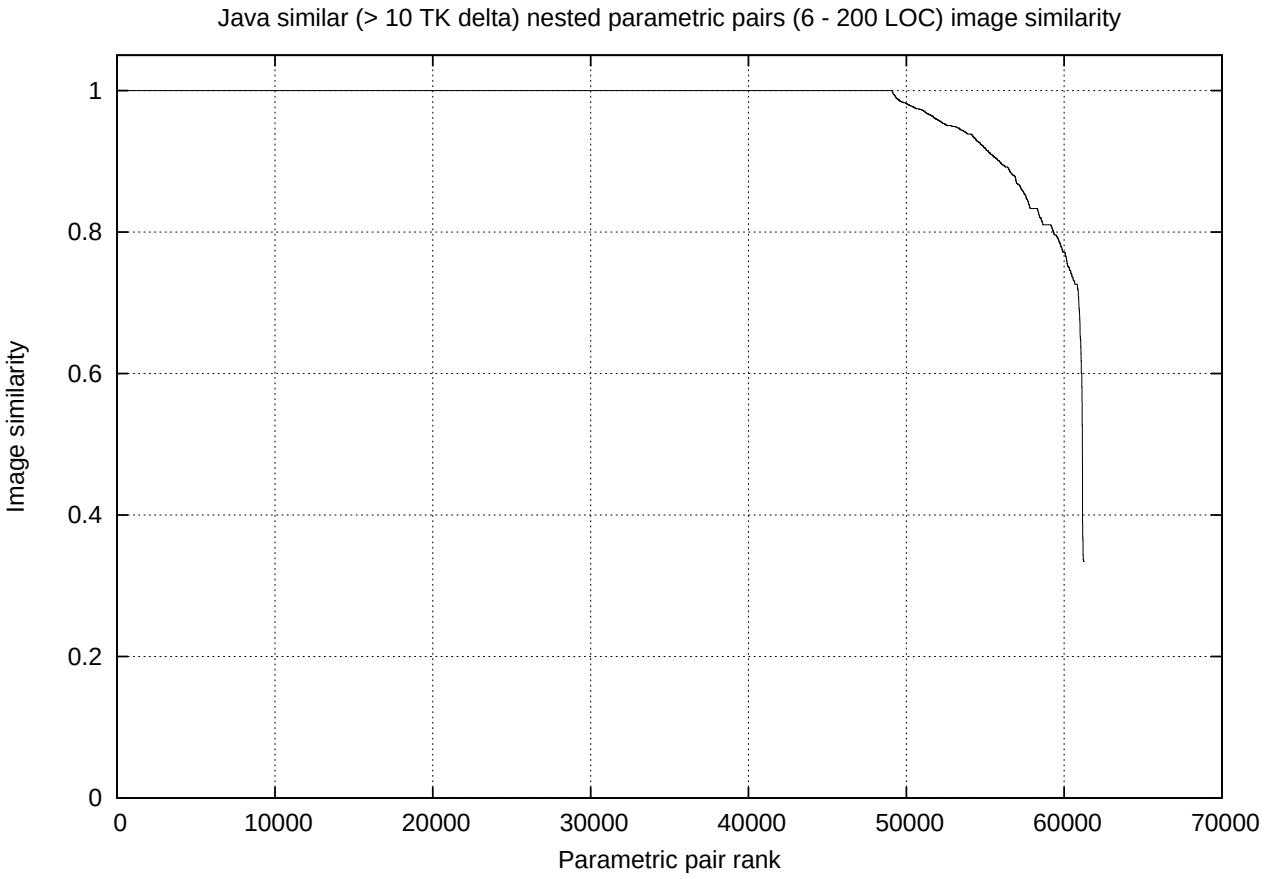


Figure 13.8 Parametric similarity distribution

for the academic version of *CLAN* where it was used as an experimental tool. However when such a tool is ported to an industrial setting where the same experiment is conducted repeatedly with some fixed parameters, usability must be maximized.

Based on that, it is of the utmost importance that the tool presented to the developers for clone detection be as simple as possible to use, to extract clone information, and to take up as little as their time as possible.

With the current sample of systems analyzed we noticed a number of those systems sharing different versions of the Linux kernel. We also found a number of systems using various versions of OpenLDAP as well as many other 3PP and open-source libraries.

System designers and software architects may take greater interest in identifying complete trees in the structure presenting a high ratio of software clones with other trees in the same or different systems developed by the company.

Using the Tree view, we were able to detect the reuse of a number of Linux distributions and a number of LDAP stacks.

Discovering common parts between the systems was not a trivial task. In the first iteration of the deployed environment for the Software Clone detection tool, the only visualization and analysis tools available were the overall tree view and the side-by-side comparison of source code fragments, presenting any similarities.

Software designers and programmers are probably more interested in the current side-by-side view. However, after some initial discussions with them, it is obvious that showing the context of the code is important.

Developers are only mildly interested in software clones because they generally have a limited circle of influence on the code they own and can modify or change. Therefore, even if they know they have software similar to some other system in the company, they cannot do anything about it.

That perspective changed when we introduced the Hierarchical Bundle Edge Graph perspective. Suddenly it was possible for system designers and architects to see information that was previously extremely difficult to find and required them to dig through the source code directory structure to identify individual software similarities. The introduction of the Hierarchical Bundle Edge Graph enabled the aggregation of information and highlighted reuse which was not apparent before.

With the Hierarchical Bundle Edge Graph we were able to confirm the existence of duplication as well as some other less obvious duplication that had gone unidentified and unadressed.

We quite easily identified duplication of some cryptographic libraries, Ssh libraries, au-

thentication libraries, strings manipulation libraries, virtualization libraries, graphic libraries, synchronization libraries, and encoding libraries.

The aggregation of the information in a relevant format quickly made the software similarity analysis relevant to the designer and architect group who has the potential to suggest the merging of those commonly used libraries between different systems.

Finding the proper visualization immediately made the technique accessible within the company. The visual appeal of the graph got extremely positive comments from the first-time viewers.

On the other hand, some actions can be expected from making the clone information visible to software architects and managers who can see the benefits of limiting independently managed copies of open-source or 3PP libraries within their products.

The structure of the company implies that if a product owner sees the need for a specific 3PP or open-source library, he will proceed to the selection of the best candidate for such a library. The product developers will then integrate the library within the product and testers will make sure that the library operates as expected. The release process and deployment of the product will need to take into account the inclusion of such a library (technical as well as legal considerations).

As could be expected, many Ericsson products will require similar functionality from 3PP or open-source libraries as they are all communication products. Thus, the above mentioned process is repeated for a number of product organizations within the company.

Assuming a number of product owners go through those steps, not only will they come to the same (or equivalent) selection of a specific version of a library, but they will have to independently track the bug correction releases or new feature releases to determine the need to re-integrate that new version, and then re-integrate, test, validate, release, etc. the new versions. The cost of doing so multiple times for those different products is simply a waste which cannot be recouped.

It is obvious that it is more efficient to put the process of selecting, tracking, and testing a given component in the hands of a single team which can provide a proper isolation layer around the component and provide it uniformly to internal customers, thus ensuring consistency within the company.

It is important to designate a team within the company that would be responsible for a common release of the Linux version or a common release of OpenLDAP.

Failing to do so could lead to potential inconsistencies between systems. This means that each system owner has to spend time to maintain the proper version of those common parts individually (including analysis of the required version, integration, testing, etc.), thereby duplicating the required effort within the company.

In any sizeable company development silos emerge. The software development follows a structure which reflects the organization itself. Working beyond those silos is remarkably difficult and time consuming.

In any large company, making changes which could affect the company structure takes some time. The awareness of the software clone existence is now a given and discussions about some of the highly redundant libraries have begun possibly leading to bringing them under the control of a single entity within the company.

However, this will require organizational changes, since a number of product organizations have control over the activities related to the integration of those libraries and need to relinquish that control to some other organization.

As mentioned earlier, one open question was to investigate the impact of clone information on developers and on the development process. At posteriori it appears that clone information does not lead to any actions from the developer. After interviewing many developers who participated in testing the clone detection technology implemented at Ericsson, it becomes evident that besides the intellectual interest of discovering the existence of clones within the boundary of their code, developers have low empowerment or time to actually take actions based on the clone information.

Refactoring tools are available to developers within their *IDE* of choice (mainly Eclipse is used within Ericsson, an officially certified distribution being maintained for the developer community). Also, with the introduction of the clone detection tool, clone information is now readily available to developers.

However, with the prominent development model based on a form of agile development comprised of short sprints, the developers have little time to spare with tools which can be perceived as peripheral, for which the benefit is perceived as a minor quality improvement.

It should also be noted that the clones detected within the company software are mainly those that spread over a number of organizational boundaries. Very few clones are within the context of what is under the responsibility of a single developer. Moreover those few clones are generally of low interest for the positive impact refactoring would bring to them.

Thus, because of the project governance model, lack of time to implement minor quality changes and the fact that the most interesting changes would spread over a number of organizational boundaries, no developer has taken any direct action based on the clone information.

Detected clones are simply outside the maintainability horizon of the developers. It appears that actions on clones requires an organizational change.

This is quite different from the dominant open-source software development culture with few organizational barriers, where developers may more easily take actions based on clone information.

At present, developers cannot be expected to take action on the clone information. The solution requires organizational changes in order to consolidate the process of selecting, releasing, and packaging commonly used 3PP or open-source components. Discussions have started but changes take time and can only be achieved with a proper visualization aid for the architects and managers so that they can see the duplication and wasted effort brought by the existence of such software clones.

One last consideration relates to the choice of technology for the deployment of the clone detection tool. The developers' platform is usually a multi-purpose laptop that they use not only for software development but also for other day-to-day tasks. Laptops reboot regularly and their disk space is generally limited as network storage is prioritized. Since the processing of a large system can take more than eight hours, the clone detection should be centrally processed, allowing at the same time inter-product clone detection in a context where the development effort is spread throughout the world.

Also, because it is more economical to use virtual servers than dedicated ones, we decided to deploy the clone detection tool on a virtualized web server within the company to provide uniformity and save money.

13.1.6 Academic perspectives

Transferring a clone detection environment and process highlights differences between a research and an industrial perspective. From an algorithmic point of view, *CLAN* is a highly customizable and configurable program. At run-time, several matching or visualization parameters can be set, including metrics, thresholds, filtering and visualization parameters.

Issues arise from a highly customizable environment, since interfering configurations are difficult to manage consistently. Although *CLAN* has been tested, many combinations may result in run-time errors that users find hard to solve by making appropriate changes in configurations. Industrial usage focuses on one or several configurations corresponding to recommended or usual practices. Consequently, part of the transfer involved identifying those configurations that best correspond to industrial needs. Proper parameters would then be frozen as fixed configurations at run-time.

This consolidation effort also aims towards a better rationalization of output. In research, we would tend to collect a lot of potentially useful intermediate results and store them for future use. However, in industrial applications, only the necessary information for current processing is stored, allowing for a leaner ,more efficient *I/O*, and disk access time.

Development, optimization, and test phases were frozen for a specific Linux version with a 64-bit architecture. These phases present dependencies with respect to development tools and configurations, such as the compiler version, library versions, and so on. In order to freeze

the development, a virtualization approach has been followed. A *CLAN* virtual machine with all the appropriate tools, libraries, and binaries has been built. Distribution of the *CLAN* tool across the organization can be performed by installing multiple copies of the same virtual environment.

13.2 Security Discordant Clones

The figures and tables in this section were published in the original paper [41], and are reproduced here with the explicit permission of the lead author, Francois Gauthier, to whom we give many thanks.

The intuition behind security concordant clones is actually very simple: we hypothesized that code fragments that have syntactic similarities should be protected with similar security privileges. The tricky part is to statically determine the privileges of all code fragments. We will not give all the details of the required analysis, but we will summarize its core steps:

- Manually identify security patterns in an application
- Generate the control flow graph (CFG) and the call graph of the application
- Assume that all statements initially have no privilege
- Do a fixed point interprocedural static flow analysis of the propagation of the patterns
- Associate all fragments with the privileges of their statements.

The reader is invited to refer to [41] and many of its references for a complete discourse on security pattern traversals and propagations in a static analysis framework. What is relevant to this work is that at the end of this analysis, we get all the privileges of a fragment. We then do a clone analysis of the system, but we restrict the results to the clone pairs and clone clusters for which the fragments do not share the same set of privileges. We report those fragments as security discordant clones. We then manually review those files to confirm or infer the presence of a security flaw.

We used Moodle and Joomla! as target systems. It is important to note that the main reason behind such a small benchmark is the difficulty in finding applications in PHP that have stereotyped and well-defined security patterns.

The total computation times are reported in table 13.3. The total computation time for the clones is rather high compared to previously reported execution time, but this experiment used a distance threshold of 0.3, which is relatively high and lead to a massive increase in the

Table 13.3 Computation times for each step of the analysis together with numbers of privileges and PHP lines of code.

Application	Computation times			Privileges	PHP LOC
	Clones	SPT	Security-discordant clone clusters		
Joomla! 2.5.4	38m27s	4m26s	27s	81	266,458
Moodle 2.3.2	10h12m24	2h43m36s	7m26s	307	1,245,417

number of results. Since the clone detection algorithm is output sensitive, the high threshold accounts for the increase in the computation time.

Figure 13.9 presents the ratio of files and LOCs that were manually inspected to the total number of files and LOCs. For, Joomla!, the investigation was on 8% of the files and 4% of the total number of LOCs, while for Moodle, we investigated of 12% of the files and 2% of the total number of LOC. In general, the investigated files were small in Moodle, despite being widespread in the whole system.

The results were then categorized according to the following scheme:

1. **Semantic inconsistency**: Clusters that fall in this category reveal fragments that are protected by privileges which are semantically unrelated to the action the fragments perform. Semantic inconsistency is related to CWE-285: Improper Authorization.

2. **Shaky logic**: The fragments that compose these clusters differ from a protection perspective but the logic behind the enforced privileges is not clear. Shaky logic cases are related to CWE-637: Unnecessary Complexity in Protection Mechanism and CWE-863: Incorrect Authorization (15th).

3. **Weak encapsulation**: Weak encapsulation characterizes fragments that are "encapsulated" in privileged code. Weak encapsulation is most closely related to CWE-749: Exposed Dangerous Method or Function and less specifically to CWE-863: Incorrect Authorization (15th).

4. **Least privilege violation**: Least privilege violations occur when fragments are protected by privileges that are too elevated for the task at hand. Least privilege violations are related to CWE-250: Execution with Unnecessary Privileges (11th).

5. **Missing privilege**: This category encompasses clusters where some fragments miss a privilege check. Missing privileges are related to CWE-862: Missing Authorization (6th).

Figures 13.10, 13.11(a), 13.11(b) show the distribution of the clusters according to these categories. Figures 13.11(a) and 13.11(b) also report the legitimate inconsistencies and the utility functions, which are irrelevant from a security point of view. It appears that for Moodle, the vast majority of the clusters fall under these two categories, while for Joomla!, a little less than 50% of the clusters fall under these categories. This means that the technique suffers from a lot of false-positives. In the end, we confirmed that 16% of the results from Moodle and 54% of the results from Joomla were true positives. No explicit reason is given to explain the high disparity of the accuracy of the technique applied to the two systems and further investigation should be done to identify the reason.

Interestingly, the clusters in both systems are distributed in a completely different fashion. For example, around 25% of the inconsistent clone clusters of Joomla! are a least privilege violation, but Moodle has none. Also, weak-encapsulation is dominant in Moodle, but the pattern appears the least in Joomla!.

13.2.1 Identified Security flaws

Investigation of security-discordant clone clusters revealed 4, previously unknown, security flaws in the systems under investigation as well as 5 previously known flaws. Table 13.4 summarizes the numbers of novel and known access control flaws that were detected by our approach.

Two novel access control flaws were identified in Joomla!. The first flaw (CVE-2013-3056) allowed remote authenticated users to bypass intended privilege requirements and delete the private messages of arbitrary users. Indeed, no privilege check asserted that the message to be deleted belonged to the user performing the deletion.

Table 13.4 Numbers of novel and known access control flaws that were revealed by security-discordant clone clusters.

Application	Novel flaws	Known flaws	
		Detected	Missed
Joomla! 2.5.4	2	1	0
Moodle 2.3.2	2	4	2

The second security flaw in Joomla! (CVE-2013-3057) allowed remote authenticated users to bypass intended privilege requirements and list the privileges of arbitrary users. In this case, security-discordant clone clusters revealed that debug pages were performing operations that are privileged elsewhere in the system. Further investigation confirmed that these debug pages could leak sensitive information to unprivileged users.

Two novel access control flaws were also identified in Moodle. The first flaw (MDL-39570)[1] concerns a missing privilege check that allows remote authenticated users to bypass intended privilege requirements and access reports they should be denied access to.

The second flaw in Moodle (MDL-39628)[2] allows remote authenticated users to bypass intended privilege requirements and access a chat room without the mod/chat:chat privilege. Contrary to previous flaws, this one is of the semantic inconsistency type as the privilege check that is performed (verify if the user is a guest) is semantically unrelated to the action it protects (enter a chat room).

This work was seminal in clone detection for PHP. Using clones to detect errors and bugs is not a new idea, recent work by [83], [115] and [56] extensively used clones to detect inconsistencies in programs and suggest potential bugs. In all cases, the clone detector leverages information from a defective piece of code to detect latent bugs. The main limitation of these approaches lies in the fact that a defective chunk of code is required *a priori*. Our approach works without such knowledge as it relies on inferred security specifications and privilege granting patterns extracted from the source code. From a security point of view, our approach has the advantage of detecting security flaws *before* a "day zero attack".

A similar work was presented in [42], but used LDA instead of clone detection to identify similar code fragments. The study presented in that paper was restricted to Moodle only, and it identified only 2 flaws, whereas the clone detection based method identified 6 of them. The data being limited to a single system are of course not statiscally significant, but the results in this section may still be interpreted as an evidence that there exists some cases in some problems where clone detection achieve better results than LDA. However, while the number of identified flaws was higher with clone detection, one flaw identified with LDA wasn't caught by the clone approach. That case would be interesting to investigate to understand some fundamental differences between LDA and clone detection.

[1] https://tracker.moodle.org/browse/MDL-39570
[2] https://tracker.moodle.org/browse/MDL-39628

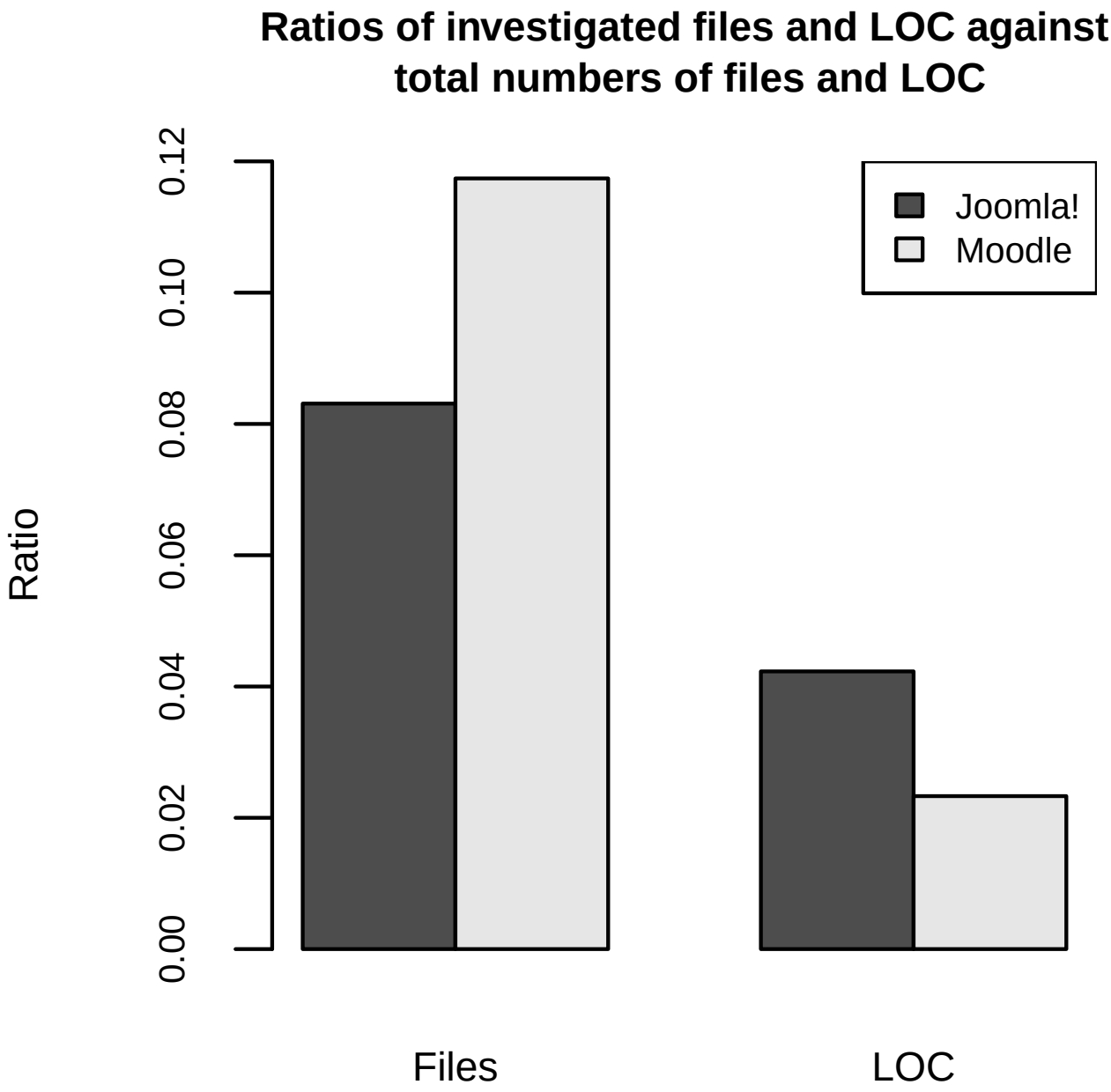


Figure 13.9 Ratios of files and PHP lines of code that were reviewed during the investigation of security-discordant clone clusters.

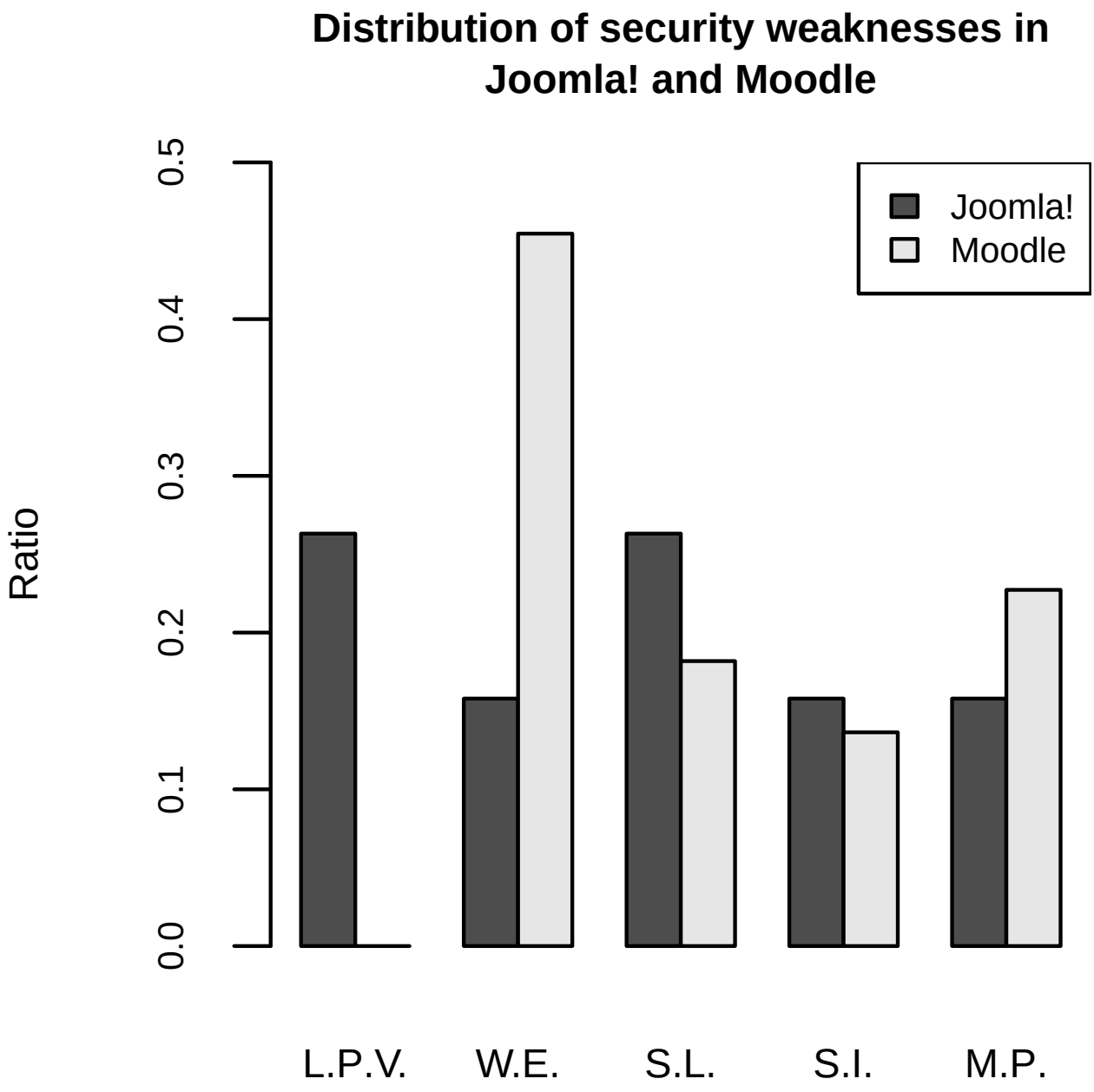
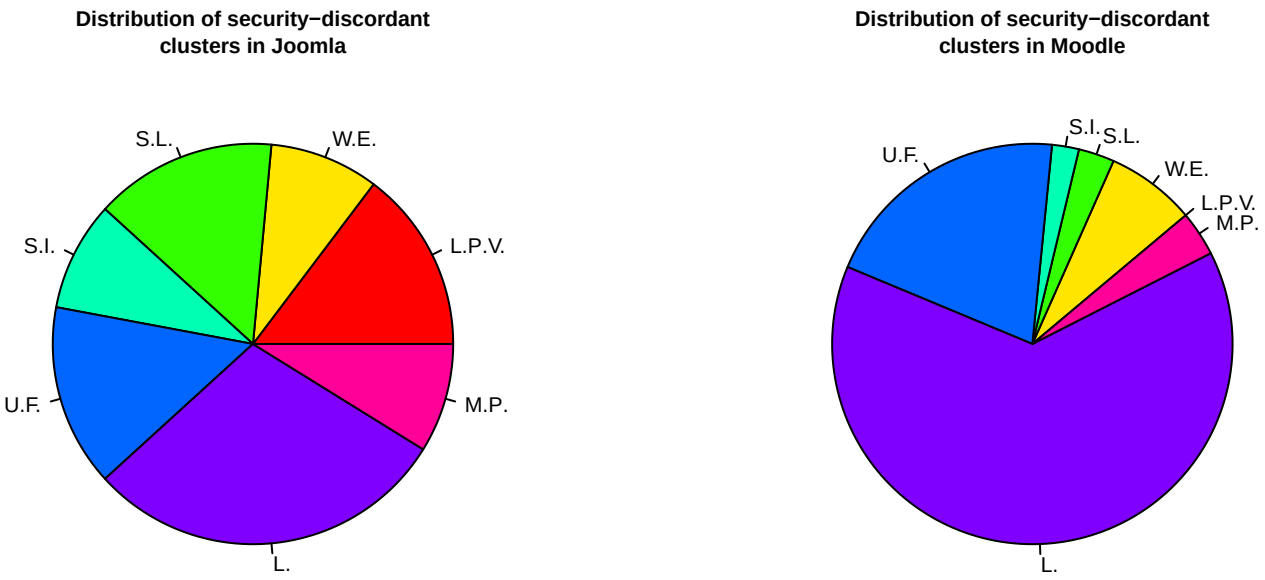


Figure 13.10 Distribution of security weaknesses in Joomla! and Moodle. Category names were abbreviated for space considerations: Least privilege violation (L.P.V.), Weak encapsulation (W.E.), Shaky logic (S.L.), Semantic inconsistency (S.I.) and Missing privilege (M.P.)



(a) Joomla! (b) Moodle

Figure 13.11 Distribution of the categories of security-discordant clone clusters in Joomla! and Moodle. Category names were abbreviated for space considerations: Least privilege violation (L.P.V.), Weak encapsulation (W.E.), Shaky logic (S.L.), Semantic inconsistency (S.I.), Missing privilege (M.P.), Utility function (U.F.) and Legitimate (L.)

CHAPTER 14

GENERAL DISCUSSION

This work is clearly separated into two themes: techniques for clone detection and application of the technology to solve different problems related to software comprehension and maintenance. This chapter discusses the different aspects and impacts on these two topics in the work presented.

14.1 Discussion on The Techniques for Clone Detection

At the beginning of this work, many experts in the field meeting at the Dagstuhl seminar on industrial clones [71] expressed their disbelief on the possibility of solving many problems pertaining to type-3 and type-4 clones. The paper presented in chapter 4 was published only few months after this seminar and in a sense provided some answers to the questions formulated during this seminar.

First, what was referred to as large-gap type-3 clones (very different type-3 clones) indeed proved themselves to be a hard task to tackle as well as a difficult set of clones to approximate properly; the reader should recall that the quality of the approximation declines with the distance between the clones. However, the proposed approximation did improve the scaling of high-quality matching from around two thousand lines of code to tens of thousands on a token basis. Namely, the size factor for which the technique can handle fragments was improved by a factor of around 1,000, as most techniques drop the bigger fragments. While it remains true that tremendously big fragments are seldom encountered, the new technique did propose an easy way to get good matching for fragments of this scale. Nevertheless, as good as the approximation is, it doesn't improve on the general accuracy expected from clone detectors. In chapter 2, we cited an adverse bound, which suggests that textual and syntactic matching may already be at the limit of their capabilities. Different paradigms would be needed to address the still open problems of unpractical time and memory overhead in type-3 and higher types of clones.

Second, the seminar put emphasis on application-focused clone detection. This proved to be one of the motivations behind the development of the nearest-neighbour clone detection paradigm, as for many applications the closest fragment is sufficient. As shown in the many applications proposed in this thesis, configuring the tool on a per-basis application instead of thriving to get a universally configured, fine-tuned clone detector has proven to be much more

relevant than the general technique itself. For example, recovering repository information worked better on images instead of token types. This was a very surprising result with respect to the literature, but not so surprising considering the problem: repository reconstruction requires the closest match possible on as many criteria as possible, and not just a simple structural match.

Finally, clone detection oriented towards a specific goal allows combined clone information with other information inferred from the software, like security properties or defect rate. Cross-referencing clones with external information was the crux of all the applications presented in this thesis. In that sense, it answers positively the second question asked at Dagstuhl: should we do application-based clone detection?

14.2 Discussion on Applications of Clone Detection

Different applications have been investigated successfully, but almost all of them raised as many questions as they provided insight. In fact, this might be the paramount problem of clone detection: what can we really get from it?

In the reconstruction of repositories and in the identification of security inconsistencies, clone detection provided a very intuitive way to solve these problems. Both problems were inherently defined as a code similarity problem: the origin of a file is very likely to be the most similar file that exists and inconsistencies are likely to be very small differences. However, in both cases, the framework provided by the problem constrained the similarities in such a way that enhanced their meaning: the similarities were interesting because of an intended purpose. In some sense, this structure provides a natural filter to much of the natural noise that occurs in clone analysis.

For the other applications, awareness of cloning was a first step and it proved to be enough, yet the problems for which the clone analysis was used did not have a complete resolution. Refactoring in both experiments done with Ericsson was at first perceived to be one of the possible courses of action, but other considerations, such as industrial culture, code ownership, etc. limited its possible application. However, it is clear that uncovering the clones and measuring the extent of the phenomenon is welcomed, albeit mitigated by the lack of direct actions taken to pursue analysis further.

In all cases, clones provided insights on how the applications were developed, or on how they would benefit from some changes. At the very least, the clones always raise interesting questions that need answers, and in the end it might be a proper role for clone detection. Questions are always related to hidden issues and not being aware of issues, whether highly critical or not, and can lead to wrong or ill-informed decisions. This work showed that

different categories of problems can be solved by first looking at the existing clones in a software. While never the solution, it did suggest a good direction in order to begin to solve the problems.

14.3 General Observations and Closing Comments

Many type-3 clone detectors rely directly or approximately on metrics related to the Levenshtein distance, including the one presented in this work. The major difference between the previously existing clone detectors and the novel method of this thesis is the choice of the approximation and the underlying data structures to accelerate the technique and remove memory limitations on big code fragments. Despite that, there exists specialized technique, like hashing, restricted to type-1 and type-2 clones that outperform most of the general type-3 clone detectors, and all performance discussions for type-3 should not take those specialized techniques into account.

As good as the existing techniques are, understanding their behaviour and creating a model of the expected clones is still a major problem that has proven itself outside the reach of scientists. Indeed, there is no known distribution function for the distribution of the edit distance in a space of strings, which could lead to a better understanding of cloning patterns and the expected frequency of cloning. Moreover, only bounds on the mean of the edit distance exist and the order of magnitude of the standard deviation is only known for binary strings. The reader is invited to consult the work in [27, 66, 81] for further details on the probability distribution of the edit distance.

This implies that, while the edit distance is prominent amongst clone detectors, we actually know little of the expected results, apart from what our intuition and our experience has brought. Thus, the lack of a complete theoretical model makes understanding clone detectors much harder. It also implies that improving the quality of type-3 clones is a very hard task, since it will either require fully understanding the edit distance or a paradigm change. However, a paradigm change is plagued by many of the limitations exposed in 2.

Despite these facts, applications of clone detectors, like the many presented in this work, are still possible. Even if applications of clone detectors would benefit from an improvement of the general quality of the clone results, their practical use is not jeopardized by the inherent false-positives and false-negatives. Thus, in the same vein as the original comments at Dagstuhl, it is likely that in the future, clone detection will be more and more tied to a specific use and no longer seen as a ubiquitous and independent field of research.

CHAPTER 15

CONCLUSION

In this work, we have dealt with two different aspects of clone detection: improving detection and applying clone detection to improve and gather insight into applications.

In chapter 3, we introduced a novel clone detection technology based on the metric tree space partitioning data structure. While allowing the use of general metric space, we chose to focus on approximations using the l_1 distance and chapter 4 showed that the proposed technique is an exemplary approximation of the Levenshtein distance. Clone detection using metric trees is not inherently incremental, but small modifications and the use of lazy deletion as shown in chapter 7 provide a practical mean to do incremental clone detection. Chosing the metric tree technique over existing techniques has the advantage of a small memory footprint and the capability to scale on very large fragments, while maintaining relatively fast running times. It also has the advantage to offer a lot of flexibility to consider different metrics all within the same framework. To complete the discussion around the detection techniques, chapter 6 offered a quick discussion of how standard distances are actually metrics and are equivalent to one another, while arguing in favor of using metrics instead of general distances.

Different applications of the proposed technique were introduced in the following chapters. Each of these applications were aimed at improving or understanding software.

In chapter 9, a detailed analysis of the evolution of Firefox was presented. The analysis also tried to link the amount and the quality of the changes between versions to the mean time between failure in those versions. A comparison of before and after a change in the release cycle was done in order to assess whether or not this change had an impact on the nature of the changes between the versions. We found that a change in release cycles lead to fewer deep changes, but that even if the changes were smaller on average, they lead to an increased failure time.

In chapter 11, we presented an application of clone detection for the TTCN-3 language, a specialized language to test telecommunication applications. We showed that this language has unique a clone distribution property, especially that is has a much higher clone rate than other standard languages like Java and C/C++. This was in part due to the fact that testers use copy and pasting as a programming paradigm. We discussed the impact of such a high cloning rate within an organization as large as Ericsson.

In chapter 12, we designed a new technique to infer and recover missing information from file repositories. Mainly, we created an algorithm to recover missing file moves. We bench-

marked the technique against many versions of large systems and showed that unlike other applications, this algorithm performed better using textual representation instead of token types. We also discussed the difference in impact on missing information in file repositories.

Finally, in chapter 13, we presented two complementary results. First, we presented clone distributions in a very large telecommunication code basis. We also discussed the impact of clone detection in a very large company. Second, we presented the results of a cross-analysis between security pattern traversal and clone detection in order to find inconsistencies in code fragments and identify security flaws. We found new security flaws acknowledged by developers. We also argued that combining static flow analysis with clone detection could lead to more detection of security flaws before ”day-zero” attacks.

Future research in clone analysis can be separated in two categories:

- **Clone Detection:** Clone detection can be improved by better understanding similarity measures, as argued in chapter 14. Research in identifying clones using features other than syntactic and lexical ones, like memory footprint, could be investigated.

- **Clone Application:** Clone application research can be pursued by trying to cross-match external information with cloning information like the work presented in 13. Further inconsistencies analysis could be explored. Future work could also include further investigation into the usability of clone detection in large industrial settings, including how to promote their use by developers.

BIBLIOGRAPHY

[1] Adempiere. http://sourceforge.net/projects/adempiere/.

[2] Jhotdraw. http://sourceforge.net/projects/jhotdraw/.

[3] Mozilla foundation. ftp.mozilla.org/pub/mozilla.org/firefox.

[4] Tomcat. http://tomcat.apache.org.

[5] Ttcn-3 official website, Nov. 2014. www.ttcn-3.org.

[6] Ttcn-3 official website, Nov. 2014. www.ttcn-3.org/index.php/about/introduction.

[7] B. Al-Batran, B. Schätz, and B. Hummel. Semantic clone detection for model-based development of embedded systems. In *Proceedings of the 14th international conference on Model driven engineering languages and systems*, MODELS'11, pages 258–272, 2011.

[8] F. Al-Omari, I. Keivanloo, C. Roy, and J. Rilling. Detecting clones across microsoft .net programming languages. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 405–414, 2012.

[9] M. Alalfi, J. Cordy, T. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for simulink models. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 295–304, 2012.

[10] A. Andoni and R. Krauthgamer. The smoothed complexity of edit distance. *ACM Transactions on Algorithms*, 8(4):44, 2012.

[11] A. Andoni and K. Onak. Approximating edit distance in near-linear time. *SIAM J. Comput.*, 41(6):1635–1648, 2012.

[12] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics—Doklady*, 11(5):1209–1210, 1970.

[13] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In *ECOOP 98, SCM-8, LNCS 1439*, pages 146–157. Springer-Verlag, 1998.

[14] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *WCRE*, pages 326–336, 1999.

[15] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. International Software Metrics Symposium*, pages 292–303. IEEE Computer Society Press, 1999.

[16] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis as a basis for object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 98–107. IEEE Computer Society Press, 2000.

[17] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis as a basis for object-oriented system refactoring. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 98–107. IEEE Computer Society Press, 2000.

[18] T. Ball, J. min Kim, A. A. Porter, and H. P. Siy. Abstract if your version control system could talk...

[19] L. Barbour, H. Yuan, and Y. Zou. A technique for just-in-time clone detection in large scale systems. In *ICPC*, pages 76–79, 2010.

[20] H. Basit, S. Pugliesi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *ESECFSE*, 2007.

[21] I. Baxter, A. Yahin, l. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 368–377, 1998.

[22] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, 1998.

[23] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering - IEEE Computer Society Press*, 33(9):577–591, 2007.

[24] H. Berghel and D. Roach. An extension of ukkonen's enhanced dynamic programming asm algorithm. *ACM Trans. Inf. Syst.*, 14(1):94–106, 1996.

[25] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[26] J. Cheney, S. Chong, N. Foster, M. I. Seltzer, and S. Vansummeren. Provenance: a future history. In *OOPSLA Companion*, pages 957–964, 2009.

[27] V. Chvatal and D. Sankoff. Longest common subsequences of two random sequences. Technical report, Stanford, CA, USA, 1975.

[28] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of 23rd International Conference on Very Large Data Bases*, pages 426–435. Morgan Kaufmann Publishers, 1997.

[29] J. R. Cordy and C. K. Roy. Debcheck: Efficient checking for open source code clones in software systems. In *ICPC*, pages 217–218, 2011.

[30] J. R. Cordy and C. K. Roy. The nicad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 219–220, Washington, DC, USA, 2011. IEEE Computer Society.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[32] Y. Dang, S. Ge, R. Huang, and D. Zhang. Code clone detection experience at microsoft. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 63–64, 2011.

[33] Y. Dang, S. Ge, R. Huang, and D. Zhang. Code clone detection experience at microsoft. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 63–64, New York, NY, USA, 2011. ACM.

[34] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. Xiao: Tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 369–378, New York, NY, USA, 2012. ACM.

[35] I. Davis and M. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 242–246, 2010.

[36] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, S. Teuchert, and J. F. Girard. Clone detection in automotive model-based development. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, 2008.

[37] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 603–612, 2008.

[38] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance and Evolution: Research and Practice - Wiley InterScience*, (18):37–58, 2006.

[39] G. F., L. T., and M. E. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013.

[40] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Softw. Engg.*, 13(6):601–643, Dec. 2008.

[41] F. Gauthier, T. Lavoie, and E. Merlo. Uncovering access control weaknesses and flaws with security-discordant software clones. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 209–218. ACM, 2013.

[42] F. Gauthier and E. Merlo. Semantic smells and errors in access control models: a case study in php. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1169–1172. IEEE, 2013.

[43] P. L. Geesaman, J. R. Cordy, and A. Zouaq. Light-weight ontology alignment using best-match clone detection. In *Proc. IWSC*, pages 1–7, 2013.

[44] D. M. German, M. D. Penta, G. Antoniol, and Y. gael Gueheneuc. Code siblings: Phenotype evolution. In *In Proc. of the 3rd Intl. Workshop on Detection of Software Clones*, 2009.

[45] N. Göde and R. Koschke. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR' 2009, pages 219–228. IEEE Computer Society, 2009.

[46] N. Gode and R. Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 219–228, 2009.

[47] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *Software Engineering, IEEE Transactions on*, 31(2):166–181, 2005.

[48] M. W. Godfrey, D. M. German, J. Davies, and A. Hindle. Determining the provenance of software artifacts. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 65–66, New York, NY, USA, 2011. ACM.

[49] J. Guo and Y. Zou. Detecting clones in business applications. In *Proceedings of the Working Conference on Reverse Engineering*, 2008.

[50] M. Hackerott and A. Urquhart. An hypothesis test technique for determining a difference in sampled parts defective utilizing fisher's exact test ic production. *IEEE Transactions on Semiconductor Manufacturing*, 3(4):247–248, Nov 1990.

[51] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *WCRE' 12*, pages 357–366, 2012.

[52] F. Hermans, B. Sedee, M. Pinzger, and A. v. Deursen. Data clone detection and visualization in spreadsheets. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 292–301, Piscataway, NJ, USA, 2013. IEEE Press.

[53] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A pdg-based approach. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 3–12, 2011.

[54] W. Hines. *Probability and Statistics in Engineering*. John Wiley, 2003.

[55] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9, 2010.

[56] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2012.

[57] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 96–105, 2007.

[58] E. Juergens, F. Deissenboeck, and B. Hummel. Clone detective - a workbench for clone detection research. In *ICSE*, pages 603–606, 2009.

[59] T. Kamiya. Agec: An execution-semantic clone detection tool. In *Proc. ICPC*, pages 227–229. IEEE, 2013.

[60] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[61] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[62] I. Keivanloo. Leveraging clone detection for internet-scale source code search. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 277–280, 2012.

[63] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *MSR' 12*, pages 179–188, 2012.

[64] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, 2011.

[65] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[66] M. Kiwi, M. Loebl, and J. Matoušek. Expected length of the longest common subsequence for large alphabets. In M. Farach-Colton, editor, *LATIN 2004: Theoretical Informatics*, volume 2976 of *Lecture Notes in Computer Science*, pages 302–311. Springer Berlin Heidelberg, 2004.

[67] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, 2001.

[68] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:77–108, March 1996.

[69] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *CSMR*, pages 309–318, 2012.

[70] R. Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 2013. accepted for publication.

[71] R. Koschke, I. D. Baxter, M. Conradt, and J. R. Cordy. Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071). *Dagstuhl Reports*, 2(2):21–57, 2012.

[72] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering*, pages 253–262. IEEE Computer Society Press, 2006.

[73] R. Krauthgamer and Y. Rabani. Improved lower bounds for embeddings into l1. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 1010–1017, New York, NY, USA, 2006. ACM.

[74] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[75] J. Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309, 2001.

[76] G. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 104–113, Feb 2014.

[77] T. Lavoie, M. Eilers-Smith, and E. Merlo. Challenging cloning related problems with gpu-based algorithms. In *IWSC*, pages 25–32, 2010.

[78] T. Lavoie and E. Merlo. Automated type-3 clone oracle using levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 34–40, New York, NY, USA, 2011. ACM.

[79] T. Lavoie and E. Merlo. An accurate estimation of the levenshtein distance using metric trees and manhattan distance. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 1–7, New York, NY, USA, 2012. ACM.

[80] T. Lavoie and E. Merlo. An accurate estimation of the levenshtein distance using metric trees and manhattan distance. In *IWSC*, pages 1–7, 2012.

[81] J. Lember, H. Matzinger, et al. Standard deviation of the longest common subsequence. *The Annals of Probability*, 37(3):1192–1235, 2009.

[82] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. volume 10, pages 707–710, 1966.

[83] J. Li and M. D. Ernst. Cbcd: cloned buggy code detector. In *ICSE '12*, pages 310–320. ACM/IEEE, 2012.

[84] J. Li and M. D. Ernst. CBCD: Cloned Buggy Code Detector. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, Zürich, Switzerland, June 6–8, 2012.

[85] J. Mayrand and F. Coallier. System acquisition based on software product assessment. In *Proceedings of the International Conference on Software Engineering*, pages 210–219, Berlin, 1996.

[86] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 244–253, Monterey, CA, Nov 1996.

[87] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 364–374, 2012.

[88] E. Merlo, G. Antoniol, M. D. Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 412–416. IEEE Computer Society Press, 2004.

[89] E. Merlo and T. Lavoie. Detection of structural redundancy in clone relations. Technical Report EPM-RT-2009-05, Ecole Polytechnique of Montreal, 2009.

[90] E. Merlo, T. Lavoie, P. Potvin, and P. Busnel. Large scale multi-language clone analysis in a telecommunication industrial setting. In *Software Clones (IWSC), 2013 7th International Workshop on*, pages 69–75. IEEE, 2013.

[91] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Folding repeated instructions for improving token-based code clone detection. In *SCAM*, pages 64–73, 2012.

[92] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *Proc. ICPC*, pages 93–102. IEEE, 2013.

[93] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, Mar. 1970.

[94] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[95] J. Pate, R. Tairas, and N. Kraft. Clone Evolution: A Systematic Review. *Journal of Software Maintenance and Evolution: Research and Practice*, Sept. 2011.

[96] C. Percival. *Matching with Mismatches and Assorted Applications*.

[97] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[98] C. Roy and J. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University, November 2007.

[99] C. Roy and J. Cordy. A survey on software clone detection research. Technical Report Technical Report 2007-541, School of Computing, Queen's University, November 2007.

[100] C. Roy and J. Cordy. An empirical study of function clones in open source software. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 81–90, Oct 2008.

[101] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *International Conference on Program Comprehension*, pages 172–181. IEEE Computer Society Press, 2008.

[102] C. Roy and J. Cordy. Towards a mutation-based automatic framework for evaluating clone detection tools. pages 137–140, Washington, DC, USA, May 2008.

[103] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. 74(7):470–495, may 2009.

[104] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[105] H. Sajnani and C. Lopes. A parallel and efficient approach to large scale clone detection. In *IWSC*, 2013.

[106] S. seok Choi and S. hyuk Cha. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, pages 43–48, 2010.

[107] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, Mar. 1981.

[108] É. D. Taillard, P. Waelti, and J. Zuber. Few statistical tests for proportions comparison. *European Journal of Operational Research*, 185(3):1336 – 1350, 2008.

[109] R. Tiarks, R. Koschke, and R. Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *Workshop on Source Code Analysis and Manipulation*, pages 67–76. IEEE Computer Society Press, 2009.

[110] E. Tuzun and E. Er. A case study on applying clone technology to an industrial application framework. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 57 –61, june 2012.

[111] M. S. Uddin, C. K. Roy, and K. A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *ICPC*, pages 236–238, 2013.

[112] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[113] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100 – 118, 1985.

[114] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[115] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC '12*, pages 359–368. ACM, 2012.

[116] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Industrial application of clone change management system. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 67 –71, june 2012.

[117] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.

[118] Y. Yuan and Y. Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 286–289, New York, NY, USA, 2012. ACM.