| | |
|---|---|
| **Titre:** Title: | Levenshtein Edit Distance-Based Type III Clone Detection Using Metric Trees |
| **Auteurs:** Authors: | Thierry M. Lavoie et Ettore Merlo |
| **Date:** | 2011 |
| **Type:** | Rapport / Report |
| **Référence:** Citation: | Lavoie, Thierry M. et Merlo, Ettore (2011). Levenshtein Edit Distance-Based Type III Clone Detection Using Metric Trees. Rapport technique. EPM-RT-2011-01. |

| | |
|---|---|
| **URL de PolyPublie:** PolyPublie URL: | http://publications.polymtl.ca/2638/ |
| **Version:** | Version officielle de l'éditeur / Published version<br>Non révisé par les pairs / Unrefereed |
| **Conditions d'utilisation:** Terms of Use: | Autre / Other |

| | |
|---|---|
| **Maison d'édition:** Publisher: | École Polytechnique de Montréal |
| **URL officiel:** Official URL: | http://publications.polymtl.ca/2638/ |
| **Mention légale:** Legal notice: | Tous droits réservés / All rights reserved |

EPM–RT–2011-01

# LEVENSHTEIN EDIT DISTANCE-BASED TYPE III CLONE DETECTION USING METRIC TREES

Thierry M. Lavoie, Ettore Merlo
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Février 2011

Poly

ÉCOLE
POLYTECHNIQUE
MONTRÉAL

EPM-RT-2011-01

# LEVENSHTEIN EDIT DISTANCE-BASED TYPE III CLONE DETECTION USING METRIC TREES

Thierry M. Lavoie, Ettore Merlo
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

Février 2011

EPM-RT-2011-01
*Levenshtein Edit Distance-Based Type III Clone Detection Using Metric Trees*
par : Thierry M. Lavoie, Ettore Merlo
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

# Levenshtein Edit Distance-Based Type III Clone Detection Using Metric Trees

Thierry Lavoie, Ettore Merlo

Department of Computer and Software Engineering, École Polytechnique de Montréal,
P.O. Box 6079, Downtown Station, Montreal, Quebec, H3C 3A7, Canada
{thierry-m.lavoie, ettore.merlo}@polymtl.ca

## Abstract

*This paper presents an original technique for clone detection with metric trees using Levenshtein distance as the metric def ned between two code fragments. This approach achieves a faster empirical performance. The resulting clones may be found with varying thresholds allowing type 3 clone detection. Experimental results of metric trees performance as well as clone detection statistics on an open source system are presented and give promising perspectives.*

## 1 Introduction

Clone detection is a f eld concerned with f nding similar patterns in source code as well as interpreting and using them in design, testing and other software engineering problems, such as ones presented in [2, 5, 8]. Classif cation of clones is usually done in three categories, although it is not unheard to use more, as in [12]. Type 1 deals with identical fragments of code, whereas type 2 deals with parametric fragments. For those types, different eff cient methods with varying precision and recall are known; the detection problem is well addressed in the literature.

However, dealing with type 3 clones which are similar fragments, i.e., they can differ in their content according to some threshold or some similarity measure, is still a diff cult problem and some issues still need to be addressed. Current clone detection methods can f nd some type 3 clones, but with great restrictions of thresholds and high computational cost. We propose an original clone detection method based on metric trees to reduce the search space and reduce the computational cost on any search done with a metric. Since the Levenshtein metric is the optimal mathematical edit distance between strings, we propose to combine metric trees, as introduced by [4], with the Levenshtein metric to compute clones with varying thresholds on that metric. The use of the Levenshtein metric is motivated by the intuitive induced interpretation of the results since it is related to the human behaviour behind text edition. Thus, we compute type 3 clones of a system based on the Levenshtein metric. Moreover, we achieve better execution time performance than the naive pairwise comparison of all fragments.

The rest of the paper is organized as follow: section 2 presents a brief literature review; section 3 gives a detailed explanation of the algorithm; section 4, the experimental setup and results; section 5, a discussion of the results, and a conclusion.

## 2 Literature Review

Clone detection state of the art includes different techniques. For type 1 and type 2, AST-based detection has been introduced by [3]. Other methods for type 1 and type 2 include metrics-based clone detection as in [10], suff x tree-based clone detection as in [7], and string matching clone detection as in [6]. For a detailed survey of clone detection techniques, a good portrait is provided in [11].

Regarding type 3 clone detection, Tiarks et al. have produced a study of the current state of the art in [13]. As def ned in this paper, type 3 clones fall in two subcategories:

- **A structure-substituted clone** is a copied fragment where some program structures have been substituted.
- **A modif ed clone** is a copied fragment where code has been deleted or added or both.

Of course, some type 3 clones may fall into both categories. Tiarks et al. support the use of the Levenshtein distance to compute a clone oracle and gave some results of the distribution of the clones using that distance. However, the results were gathered from a small sample of clone candidates and the candidates were very small code fragments.

An interesting evaluation of many clone detection techniques was done in [12]. This study def ned 16 cloning scenarios and assessed the performance of many known detection methods with respect to these 16 cases from which 9 cases were type 3 clones. According to that study, graph-based clone detection has the best potential to f nd type-3 clones. Metrics-based, tokens-based and text-based methods may also handle type-3 clones. Although they can detect type-3 clones, AST-based methods seem to be much less effective. Nevertheless, this survey shows that no known method performs well on f nding all variants of type 3 clones.

In this paper, following the support of the Levenshtein distance from Tiarks et al., we build a clone detection method using the combination of metric trees and the Levenshtein distance. Since all 16 cases presented in [12] have small Levenshtein distance, the technique can easily f nd all clones in these categories and thus should perform better than the current state-of-the-art techniques. The next section presents the new algorithm.

## 3 Metric tree-based clone detection

First, it is necessary to def ne a metric. Let $X$ be a topological space and $\delta$ a function $\delta : X \times X \rightarrow \mathbb{R}$ called a distance. The distance $\delta$ is a metric if and only if the following properties hold for $x, y, z \in X$:

$$\delta(x, y) \geq 0 \; (non \; negativity) \tag{1}$$

$$\delta(x, y) = \delta(y, x) \; (symmetry) \tag{2}$$

$$\delta(x, y) = 0 \Leftrightarrow x = y \tag{3}$$

$$\delta(x, y) + \delta(y, z) \geq \delta(x, z) \; (triangle \; inequality) \tag{4}$$

If a metric $\delta$ exists for a topological space X, then X is a metric space. In many cases, distances satisfy the f rst three axioms of a metric, but many don't satisfy the fourth, the triangle inequality, which is the key to interesting space partitioning. Many common distances are not proper metrics. For example, the overlap coeff cient and the Dice coeff - cient are useful distances for set similarity comparisons, but they fail to satisfy the triangle inequality. However, other common distances like the Jaccard coeff cient and the Levenshtein distance do satisfy the triangle inequality and are proper metrics. For this reason, the Levenshtein distance will be referred to as the Levenshtein metric in this paper. Since the construction of metric trees relies on the triangle inequality, it is hard to use approximation of the metric because metric embeddings for approximation usually fail to respect the triangle inequality.

Now, suppose we have a metric $\delta$ on space X. We want to def ne a data structure that separates the search space to increase the performance on similarity queries, namely range queries. Let M be a tree and N the set of nodes $\{n_0, n_1, ..., n_k\}$ in M. Every $n_i \in N$ is def ned as the 4-tuple $t =< x, y, d, c[4] >$, where $x, y \in X$, $d = \delta(x, y)$, and $c[i]$ is the node's ith child. We call x and y the pivots of node $n_i$. According to the preceding def nitions, we can now def ne an algorithm $insert(f)$ which take an arbitrary $f \in X$ as a parameter and insert this element into tree M as the pivot x or y of some node $n_i$. The complete def nition of the $insert(f)$ algorithm may be found in [4].

```
$detectClones(S, distance)$
    $tree = new \; MetricTree$
    $forall \; f \in S:$
        $tree.insert(f)$
    $clones = \emptyset$
    $forall \; f \in S$
        $radius = distance * len(f)$
        $clones = clones \bigcup tree.rangeQuery(f, radius)$
    $return \; clones$
```

Figure 1: Clone detection algorithm

With the metric-tree built for clone candidates under the Levenshtein metric $\delta_l$, it is easy to perform a range-query for fragment $f$ with radius $r$ to f nd all fragments $f'$ for which $\delta_l(f, f') \leq r$. For a complete description of the range-query algorithm as well as some possible optimizations on the tree building operation, see [4].

Suppose we have a system $S = \{f | f \; is \; a \; code \; fragment \; of \; S\}$. Let $tree$ be the metric tree built with all $f \in S$ under metric $\delta_l$. Also, let $len(f)$ be the length of $f$. Finding all clone fragments with respect to $f$ is equivalent to searching for all fragments $f'$ for which $d_l(f, f') \leq r$ for a chosen radius r. This is the exact def nition of a range query in metric trees; therefore, f nding all clones to fragment $f$ is equivalent to a range-query on $tree$ around $f$ with radius $r$. All these steps are presented in the algorithm of f gure 1. The method $rangeQuery$ takes a fragment $f$ as f rst parameter and the radius as second parameter; it returns the set of all clones to $f$. In this algorithm, the radius was chosen to be a function of $len(f)$ and a constant parameter $distance$. As discussed in the next section, this was the chosen radius def nition for our experiments. However, there are no restriction to the def nition of the radius as long as $r \in \mathbb{R}$.

## 4 Experimental setup and preliminary results

The main goal of the experiments was to f nd all code fragment pairs whose Levenshtein distance is less or equal to a specif c threshold. From now on, let all code fragments of a system be the strings of their corresponding tokens as produced by the lexer of the chosen language. The use of tokens instead of strings is supported by [9]. Let

$a, b$ be two strings and $len(a), len(b)$ be the length of those strings. Then the threshold $\epsilon$ for pair $(a, b)$ is def ned as $\epsilon = d * max(len(a), len(b)))$ where $d \in (0, 1)$ is the coeff cient of desired maximum distance. Another meaningful interpretation of $d$ may be stated as: the desired similarity between fragments is $1 - d$. The cloning criteria may be phrased as: the pair $(a, b)$ is in a cloning relation $iff$ the Levenshtein distance between a and b is smaller or equal to threshold $\epsilon$ where $\epsilon$ is the maximum length of a and b times the distance coeff cient (or times one minus the similarity coeff cient). Thus, there are two important points: the queries' radius is proportional to the length of the fragments, and the experiments' varying parameter is the distance coeff cient and not directly the Levenshtein metric. Metric trees do not prohibit the use of f xed thresholds instead of proportional ones, but to recover more signif cant clones for larger fragments, it was more natural to specify the thresholds as a function of the size.

In fact, it is easy to build fragments for which f xed thresholds would falsely report them as clones or non-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 f rst 10 000 tokens, but at the end of $f'$ there is an appendage of 3 000 tokens. If threshold $\epsilon$ 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 must be size-sensitive.

For type 3 clone detection, $d$ must vary. For this reason, $d$ was chosen in $\{0.05, 0.15, 0.30\}$ to assess the eff ciency of the method. We believe the chosen values of $d$ to be reasonable. However, the exact def nition of an acceptable threshold is not specif ed in the literature. Therefore, more experiments should be done to determine an adequate interval to let $d$ vary in.

Experiments have been performed on an open-source Java system, Tomcat, and have been executed on an Intel Core 2 Duo, 2.16 GHz clock, 4 GB RAM, under Linux Fedora 13. Code has been compiled with g++ 4.4.4. Tomcat [1] is an implementation of the Java Servlet and the Java Server Pages technologies and is widely used to power different kinds of web-based systems.

The size, number of fragments and some interesting statistics on fragments size are presented in table 1. In this table and in all reported f gures and experiments, fragments refer to blocks larger than 70 tokens, which is a threshold that has been chosen as a lower bound of signif cance for blocks size. The literature suggests setting this lower bound at 7-10 LOCs. Since our size is expressed in number of tokens, we need to convert the generally accepted bound; 70 comes from the previous authors' experience and is roughly equivalent to 7-10 LOCs. Also, blocks refer to logical blocks, functions and classes. Since the Levenshtein metric computation is quadratic on the size of the fragments, this table reports different statistics on the size of the fragments to assess the expected diff culty in computing the distance.

| System | Tomcat |
|---|---:|
| Version | 5.5 |
| LOC | 130K |
| Fragments | 5084 |
| Av. Length of fragments | 341.29 |
| Max. Length of fragments | 19999 |

Table 1: System features

Preliminary results of the technique are presented in table 2. The total number of reported clones is very small compare to the total number of candidates, which is 14 311 250. The total number of candidates is the number of all possible fragment pairs. The execution time increases as the distance coeff cient increases.

An example of the clone found with the technique is shown in f gure 2. The length of the fragments are respectively 177 for 2a and 145 for 2b. The reported Levenshtein distance is 34 or equivalently 0.192 according to our experiment's def nition. This example exhibits many of the characteristics of a type 3 clone. First, there's an added pref x in the longer fragment. Second, there's an added suff x in the longer fragment. Finally, most of the code in the smaller fragment have been imbedded in a while loop in the larger fragment. These differences have been correctly identif ed by the alignment as shown in the f gure. It is interesting to see that the location of these two fragments are not only in different f les in the system but also at completely unrelated locations. This fact supports the benef ts of performing the analysis on the whole system and not only on subparts.

| Distance $d$ | Execution time (s.) | Clone pairs |
|---|---:|---:|
| 0.05 | 6023 | 226 |
| 0.15 | 9401 | 963 |
| 0.30 | 13163 | 2933 |

Table 2: Algorithm performance for Tomcat with varying distance $d$

## 5 Discussion

The goal of the experiment was to assess the possibility of detecting clones with the technique. Because of the

```
67                                  {
69          boolean hasCharset = false ;
71          int len = type . length ( ) ;
72          int index = type . indexOf ( ';' ) ;
73          while ( index != - 1 ) {
74             index ++ ;
75             while ( index < len && Character . isSpace ( type . charAt ( index ) ) ) {
76                index ++ ;
77             }
78             if ( index + 8 < len
79                   && type . charAt ( index ) == 'c'
80                   && type . charAt ( index + 1 ) == 'h'
81                   && type . charAt ( index + 2 ) == 'a'
82                   && type . charAt ( index + 3 ) == 'r'
83                   && type . charAt ( index + 4 ) == 's'
84                   && type . charAt ( index + 5 ) == 'e'
85                   && type . charAt ( index + 6 ) == 't'
86                   && type . charAt ( index + 7 ) == '=' ) {
87                hasCharset = true ;
88                break ;
89             }
90             index = type . indexOf ( ';' , index ) ;
91          }
93          return hasCharset ;
94       }
```

(a) org/apache/coyote/Response.java lines 474-493

```
474                     {
475          semicolonIndex = index ;
476          index ++ ;
477          while ( index < len && Character . isSpace ( type . charAt ( index ) ) ) {
478             index ++ ;
479          }
480          if ( index + 8 < len
481                && type . charAt ( index ) == 'c'
482                && type . charAt ( index + 1 ) == 'h'
483                && type . charAt ( index + 2 ) == 'a'
484                && type . charAt ( index + 3 ) == 'r'
485                && type . charAt ( index + 4 ) == 's'
486                && type . charAt ( index + 5 ) == 'e'
487                && type . charAt ( index + 6 ) == 't'
488                && type . charAt ( index + 7 ) == '=' ) {
489             hasCharset = true ;
490             break ;
491          }
492          index = type . indexOf ( ';' , index ) ;
493       }
```

(b) org/apache/tomcat/util/http/ContentType.java lines 67-94

Figure 2: A clone example from Tomcat with distance 0.192. Alignment differences are shown in bold. New lines were skipped.

Levenshtein metric's definition, we report 100% of type 1 clones. Moreover since we perform the detection on token-based strings instead of image-based strings, we report 100% of type 2 clones. Type 2 clones are defined as parametric clones for which identifiers are replaced. Therefore, these clones have an intrinsic Levenshtein distance of 0 if the distance is computed on their token-based representation and the algorithm will always report them as clones.

Apart from detecting all type 1 and type 2 clones, we report every Levenshtein-based type 3 clones in a system that would be reported by an exhaustive pairwise search, but at a lower computational cost. In other words, our method is equivalent to performing a search on all $\dfrac{n*(n-1)}{2}$ clone pair candidates of a system, where n is the total number of fragments in the system, and reporting all pairs for which their Levenshtein distance is below a specific threshold.

Using metric trees, we achieved an interesting type 3 clone detection on Tomcat, while the pairwise technique couldn't perform that computation in reasonable time. To estimate the required time for the pairwise method, we compared our algorithm to the pairwise technique on smaller samples of our target system. With a sample of 100 fragments, the pairwise approach completed its task in 181 s while our technique used 21 s, while for a sample of 500 fragments, the execution time were respectively 4080 s and 118s. Using those numbers, we project the required computation for the pairwise technique to be around 415 427 s. (roughly 5 days for the whole system). Based on that estimation, we project the total acceleration factor at around 30 times for our method. This acceleration factor is only valid for queries with similarity coefficient 0.3 or less. However, based on the results of table 2, for distance coefficient smaller than 0.3, the runtime for our technique decreases as the distance coefficient decreases. Thus, the total acceleration factor with respect to the pairwise method increases with smaller distance coefficient value. This is consistent with the results provided by [4], which supported a reduction of the search space for range-queries. However, it seems that the radius of the range-queries also have a direct influence on the size of the search space. Although no experimental evidence is provided here, this claim is reasonable. Further researches will try to measure the exact influence of the radius on the performance of the range-query.

This specific result is of great interest for clone detection, since we generally do not need to know the distance between non-clone fragments. While the pairwise method computes every pair's distance, the metric tree approach computes only the relevant distances for the desired similarity between fragments. Thus, if the desired similarity is high, the search space may be pruned significantly leading to a high computational gain. Moreover, even if the chosen similarity is high, the resulting clones are still optimally aligned by the Levenshtein metric.

The method is scalable for systems up to hundreds of KLOCs. Systems in the order of MLOCs would still retain the acceleration factor over the pairwise technique. However, those systems would require computational time over several days without an underlying parallel approach. Still, metric tree clone detection for huge systems may provide an opportunity to compute clones under the Levenshtein distance, which would be practically unfeasible with the pairwise approach (with computation time estimated to several years). Scalability will be investigated in future researches.

A drawback of this method if used on all blocks of a system (see the definition of block in section 4) is the possibility of finding self-referring clones. Self-referring clones are defined as follow: let a and b be fragments, then (a,b) is a self-referring clone $iff$ a is properly contained in b or b is properly contained in a. This can occur frequently if all logical blocks are used but rarely if only functions and classes are used. This may be explained by the very low probability that a function may be a proper clone to a class (usually, programmers do not make class from single method and vice versa). Nevertheless, self-referring clones may be f ltered easily and quickly after clone detection.

Because the tree is constructed using code fragments, the choice of using all blocks instead of restricting the blocks to functions and classes has a direct impact on the execution time. The total number of nodes searched in the tree is dependant on the total node inserted in the trees. Consequently, the total running time is dependent on the number of LOCs as well as the total number of fragments inserted and the average length of each fragment. In fact, the total number of nodes searched is dependant on the tree size as well as the average similarity between fragments, and the total time spent computing the Levenshtein distance is closely related to the average size of code fragments. Therefore, one must be careful when comparing the performance of the technique with respect to the size in LOCs (even though the preceding factors are related to the number of LOCs). Because of this presumed sensibility to system characteristics, we planned to validate the practical performances of our technique on many other systems.

## 6 Conclusion

This paper presented an original clone detection technique based on the Levenshtein metric and metric trees. Running times of the method have been compared with the pairwise comparison approach and achieved an acceleration factor of 30. Since the clones are computed using the Levenshtein metric, the results are intuitive and interesting type 3 clone candidates. Future research will include a scalability study on even larger systems along with an exhaustive experimental validation on many systems. Other possible continuations are a study of the method under different metrics and a comparison with existing type 3 clone detection techniques.

## 7 Acknowledgements

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

## References

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

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lagu, 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.

[3] 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.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An eff cient access method for similarity search in metric spaces. In *Proc. of 23rd International Conference on Very Large Data Bases*, pages 1–1, 1997.

[5] 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.

[6] 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*, 2006.

[7] N. Gode and R. Koschke. Incremental clone detection. In *European Conference on Software Maintenance and Reengineering*, 2009.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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*. IEEE Computer Society Press, 2009.

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

**École Polytechnique de Montréal**

**École affiliée à l'Université de Montréal**

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

**www.polymtl.ca**

ÉCOLE
POLYTECHNIQUE
MONTRÉAL