Automated Type-3 Clone Oracle Using Levenshtein Metric

Thierry Lavoie
Ecole Polytechnique de Montreal
P.O. Box 6079 Downtown Station
Montreal,Quebec
thierry-m.lavoie@polymtl.ca

Ettore Merlo
Ecole Polytechnique de Montreal
P.O. Box 6079 Downtown Station
Montreal,Quebec
ettore.merlo@polymtl.ca

ABSTRACT

Clone detection techniques quality and performance evaluation require a system along with its clone oracle, that is a reference database of all accepted clones in the investigated system. Many challenges, including finding an adequate clone definition and scalability to industrial size systems, must be overcome to create good oracles. This paper presents an original method to construct clone oracles based on the Levenshtein metric. Although other oracles exist, this is the largest known oracle for type-3 clones that was created by an automated process on massive data sets. The method behind the creation of the oracle as well as actual oracles characteristics are presented. Discussion of the results in relation to other ways of building oracles is also provided along with future research possibilities.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Algorithm

Keywords

Software clones, clone detection, type-3 clones, clone benchmark

1. INTRODUCTION

Developing new clone detection techniques requires quality assessment as well as a way to compare results with other existing techniques. Quality evaluation is usually done according to two criteria. The first one is the efficiency of resource usage, such as execution time and memory consumption, and the second one is quality of results, that is often measured using precision and recall. Efficiency of resource usage is relatively easy to measure and compare, but results quality evaluation is much more complex to achieve. The most common way to measure precision and recall is by mean of comparison with an oracle, that, for clone detection methods, is a database of all accepted clones contained in a system.

IWSC'11, May 23, 2011, Waikiki, Honolulu, HI, USA

Many state-of-the-art techniques are effective but approximate, since they may report false-positives and false-negatives. Measuring their precision only requires validating the reported candidates to sort the true-positives and the false-positives, but measuring recall requires a predetermined set of clones considered correct from which we deduce the false-negatives.

The construction of a good oracle must deal with a number of still open problems. First, a definition of what is a true clone must be provided. Since many definitions of a clone are accepted, this problem must be addressed by providing a reasonable definition at the cost of some limitation of the oracle applicability. Second, a good oracle must deal with big enough systems to have some practical interest and thus is more or less depending on an exhaustive search for clones. However, for systems in the order of millions of lines of code, exhaustive brute-force searches is computationally very expensive. Third, a large oracle could be produced by an automated mean, although *a posteriori* human validation may be considered. Current oracles are produced by human validation of clone candidates, by artificial mean, or by statistical sampling. Our objective is to reduce the computation time until it becomes feasible and acceptable to produce an automated oracle for large systems.

In this paper, we present a method to build an oracle for type-3 clones as well as the resulting oracle data for two open source systems, namely Tomcat and Eclipse. The oracle was automatically computed on a small parallel cluster using the Levenshtein metric as a similarity criterion. It contains all clone pairs in a system for which their Levenshtein distance is below a threshold that has to be selected at the moment of its construction. It can be used to assess both precision and recall of other clone detection techniques, since it contains all clone pairs, together with their Levenshtein distance, within the specified threshold. The motivation for introducing a threshold in the oracle stems from the need of somehow pruning the distance computation of those fragments pairs whose distance is over the threshold value. If the oracle threshold is chosen large enough, disregarded clone pairs show lower similarity, do not represent relevant clones for oracle purposes, and can be safely considered as not accepted clones. Of course, oracles can be built for even larger threshold values, if needed and if the computational cost is acceptable. Thus, the proposed method addresses the three previously mentioned open problems.

The rest of the paper is organized as follow: section 2 presents a brief literature review; section 3 gives a detailed explanation of the methodology used to build the oracle; section 4, the features and characteristics of the oracle; section 5, a discussion of the oracle, 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 [8]. Other methods for type-1 and type-2 include metrics-based clone detection as in [23], suffix tree-based clone detection as in [13], and string matching clone detection as in [12].

Other approaches for clone analysis have been presented in [5, 6, 14, 15, 18, 19, 22, 26]. Empirical studies and evaluation of clone detection approaches can be found in [3, 20]. Scalability of clone detection approaches has been addressed in [7, 16], while clones and software evolution have been investigated in [4, 11].

For a detailed survey of clone detection techniques, a good portrait is provided in [27].

In the following sections, a discussion about some issues in building oracles is provided.

2.1 Type-1 and Type-2 Clones

Producing an oracle for type-1 clones is a task well described in the literature and it involves only identical clones. A simple hash table using the candidates content as keys and the candidates identifiers as values will produce a good-quality automated oracle even for systems of very large size. This approach has been investigated by [17]. However, even for type-1 clones oracle, there's the need of clearly stating the underlying clone definition. Since it is not a technical problem, it's still possible to produce interesting oracles for type-1 clones. Surveys and evaluations provided in [9] and [29] include forms of type-1 clone oracles.

Producing an oracle for type-2 clones is trickier since it requires dealing with parametric clones. Although [9] and [29] also include forms of type-2 clones oracle, they do not provide the oracle for complete systems. Bellon *et al.* in [9] performed oracle comparison on a statistical basis, while Cordy *et al.* used an artificial oracle made of 16 cloning cases. The same is true regarding their approach to type-3 clones oracle. Since our clone definition deals with type-2 and type-3 clones and since type-3 clones are a superset of type-1 and 2 clones, we decided to address the type-3 clones oracle problem. Thus, we provide a more exhaustive oracle than the one used in [9] and [29]. The rest of this section reviews type-3 clones literature.

2.2 Type-3 Clones

Regarding type-3 clone detection, Tiarks et al. have produced a study of the current state of the art in [31]. As reported, type-3 clones fall in two following subcategories:

- **A structure-substituted clone** is a copied fragment where some program structures have been substituted.
- **A modified 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 751 clone pairs, 90% of which were composed of fragments both smaller than 65 LOCs.

An interesting evaluation of many clone detection techniques has been published in [29]. This study defined 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 find 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 finding all variants of type-3 clones. Also, this survey showed the importance of hand-made oracles. Another paper by Cordy et al., [30], provides a good automatic comparison framework for clone detection tools.

Type-3 clones were briefly addressed in [9], but following the lack of an adequate definition, the authors decided to let their validation be human-made. Moreover, the oracle was built on a random sample which was hand-made validated among candidates produced by clone detection tools. Therefore, the oracle space was biased by the evaluated tools. Even though this does not discredit the study, it clearly shows the difficulty of providing a good oracle independent of all clone detection methods satisfying a reasonable definition. Another important point of this oracle is the time required to produce it. Bellon took 77 hours (3.2 days or 10.3 normal workdays) to oracle a sample of 6 518 candidates (out of 325 935) which is 2% of the total candidates. A complete oracle would require 3 850 hours (160 days) of man work at this rate of evaluation and even if this task was accomplished, it would still be subject to the bias of the tools used. Nevertheless, the hand-made oracle produced by Bellon along with is survey is still a landmark in clone detection.

The oracle methodology we propose in this paper has many distinctive features compared to those presented above. First, it is completely automated. Second, the methodology is applicable on systems in the order of MLOCs without use of statistical properties. Third, since the produced oracles contain all clone pairs within a given threshold, oracles can be used to assess precision and recall of other clone detection techniques. These features make the oracles, constructed with the presented methodology, the largest ones with respect to an exact mathematical objective criterion. Finally, to make the proposed methodology independent of known clone detection techniques and solely dependent on the target system, we based the oracle on the Levenshtein metric, as reported in [31]. Merlo in [23], Cordy in [28], and Mende and Koschke in [25] and [24] also support the use of Levenshtein-based clone distance computation for post-processing filtering purposes. Although these mentioned techniques achieve the Levenshtein precision by filtering out irrelevant clone pairs, their intrinsic recall figures cannot be increased by filtering alone. The proposed oracle construction, in contrast, allows for both precision and recall assessment of other clone detection techniques with respect to Levenshtein metric. Clone pairs based on the Levenshtein metric are reasonable and depend only on the lexical properties of their code fragments, namely text strings or tokens, and not depend on the AST or other syntactic features. For many languages, clone analysis should be easier if based only on strings or tokens analysis. The next section explains the method to build oracles.

3. METHOD

Our goal is to produce an oracle defined as the set of all the fragments pairs of a system for which their Levenshtein distance between fragments is below a specific threshold. The naive approach would perform a pairwise comparison of all the fragments and report those under the selected threshold. However, this can easily be shown to be unpractical. Thus, since it is necessary to reduce the search space, we chose to use metric trees as presented in the foundation paper [10] of Ciaccia. A brief overview of their applicability to clone detection is presented below.

Clone detection always rely on the notion of distance between code fragments. In a set N of code fragments, computation of

the distances between all fragments pairs requires $\binom{|N|}{2}$ distance computations. Thus, if there was a way to constraint the distance to gain some space structuring properties, it could lead to computational gain.

The key point in the technique is the use of a distance that satisfies the metric axioms in a topological space. Let X be a topological space and δ a function $\delta : X \times X \rightarrow \mathbb{R}$ called a distance. The distance δ 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 δ exists for a topological space X, then X is a metric space. In many cases, distances satisfy the first 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 coefficient and the Dice coefficient are useful distances for set similarity comparisons, but they fail to satisfy the triangle inequality. However, other common distances like the Jaccard coefficient 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. Using the previous definitions, let N the set of all code fragments in a system be our topological space with the Levenshtein metric. Then, N is the desired metric space for clone detection.

Now, using the metric space N, we can build the data structure proposed in [10]: the metric tree. This structure supports two important primitives: $insert(f)$ and $rangeQuery(f, \epsilon)$. The $insert(f)$ primitive takes a fragment and inserts it in the tree. The other primitive $rangeQuery(f, \epsilon)$ takes a fragment f and a real number ϵ as parameters and returns the set of all the fragments f' in the tree for which $\delta(f, f') \leq \epsilon$, as follows:

$$rangeQuery(f, radius) = \{f^{'} \mid \delta(f, f^{'}) \leq \epsilon\} \tag{5}$$

Fragments f' are closer than the threshold $radius$ to f and are therefore type-3 clones of f. A key point, to be made at the moment, is to state that, since metric trees are used to test only the most relevant pairs of fragments and to prune the search space of distance computation, $rangeQuery$ returns the set of fragments f' without explicitly testing every other pair of fragments in the system involving f.

An outline of the $insert$ primitive is presented in Figure 1.

Line 2 of the algorithm starts by initializing a variable target with the node n_0 assumed to be the root of the tree. 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 loop first two steps at line 5 and 6 computes the distance of f with the node two already assigned fragments, called the pivots. From line 7 to 17, f distance 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 continue to search nodes until it finds a node for which at least one pivot is undefined. Line 17 to 22 then checks which of the pivot is undefined and setup the node accordingly with the new fragment.

The $rangeQuery$ primitive proceeds in a similar way to search

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

Figure 1: Insertion algorithm in metric trees

the regions, but with different criteria that are described in Table 1. In this table, y and x represent the current node pivots, f is the fragment, δ is the metric, ϵ is the query $radius$ and d is the distance between the two pivots, namely $d = \delta(p_1, p2)$.

The $rangeQuery$ is performed recursively in each selected region, starting with the root. This procedure is shown in Figure 2. Line 2 initializes the result to the empty set. Lines 3-6 test the distance between the query and the pivots with the desired $radius$. If it is less or equal, the corresponding pivot is added to the result set. Lines 7-14 test the predicate for each region. If true, $rangeQuery$ is performed on the corresponding region. All four regions may be visited for each node, although it is more likely that at least one region will be pruned. Thus, the query may be performed by doing less comparisons than a brute-force algorithm.

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	$\delta(x, f) + \epsilon \geq d \wedge \delta(y, f) + \epsilon \geq d$

Table 1: Criteria for regions selection in the rangeQuery primitive.

For complete details about the two primitives, the reader is encouraged to read [10]. The detailed algorithm to compute the oracle is presented in Figure 3. In this code, tree is assumed to be a metric tree with the $insert(f)$ and the $rangeQuery(f, radius)$ primitives. Two steps can be distinguished in this algorithm. The first one, from lines 3 to 4, build the tree by inserting all fragments from fragments set N. The second one, from lines 6 to 8, perform a $rangeQuery$ on every fragments in N and store the result in the variable $clones$. Line 7 computes the queries' $radius$. Finally, line 9 returns the found clones.

Building the oracle requires to find all code fragment pairs whose Levenshtein distance is less or equal to a specific threshold by using the $rangeQuery(f, radius)$ primitive. Let all code fragments of a system be the strings of their corresponding tokens' types as produced by the lexer of the chosen language. The use of tokens

```
1  rangeQuery(node, f, ε)
2      result = ∅
3      if δ(node.x, f) ≤ ε
4          result = result ∪ node.x
5      if δ(node.y, f) ≤ ε
6          result = result ∪ node.y
7      if δ(node.x, f) < ε + node.d ∧ δ(node.y, f) < ε + node.d
8          result = result ∪ rangeQuery(node.region1, f, ε)
9      if δ(node.x, f) + ε ≥ node.d ∧ δ(node.y, f) < ε + node.d
10         result = result ∪ rangeQuery(node.region2, f, ε)
11     if δ(node.x, f) < ε + node.d ∧ δ(node.y, f) + ε ≥ node.d
12         result = result ∪ rangeQuery(node.region3, f, ε)
13     if δ(node.x, f) + ε ≥ node.d ∧ δ(node.y, f) + ε ≥ node.d
14         result = result ∪ rangeQuery(node.region4, f, ε)
15     return result
```

Figure 2: Range-query algorithm in metric trees

instead of strings is supported by [19]. Let a, b be two strings and $len(a), len(b)$ be the length of those strings. Then the threshold ϵ for pair (a, b) is defined as:

$$\epsilon = distance * max(len(a), len(b))) \quad (6)$$

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 clone criterion may be phrased as: the pair (a, b) is in a clone relation *iff* the Levenshtein 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.

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 $radius$ must be size-sensitive.

Other specifications of thresholds would lead to different oracles for different clone definitions. However, the chosen definition of a clone and the chosen threshold seem consistent and should lead to a good-quality oracle.

Although metric trees can reduce the search space and the number of distance computations required to find the desired candidates, using metric trees alone is not enough to compute oracles for systems with size above a few hundred KLOCs, at the time being. This conclusion is drawn from our failed attempt to compute the oracle of Eclipse (as presented in section 4) with the presented technique. We allowed 2 weeks for the computation to complete on a single core, but the oracle was far from completion after this time. From this try, we estimated the total required time to be between 4 to 6 weeks. For this reason, we decided to use parallel computation for the range-queries. With the tree built for a system and written down in an .xml file, it is possible to distribute the computation of the range-queries with minimum effort. It is necessary and sufficient to compute a fraction of the oracle to have the tree and a list of queries. If multiple cores are given the trees and independent lists of queries for which the union is all the queries required by the oracle, then the oracle will be computed on a parallel architecture and gain a constant factor acceleration. Since dual and quad cores processors are mainstream nowadays, it is easy to have 8,16,32 or even more cores to compute the oracle. Other parallel solutions such as GPUs may also potentially be used, as reported in [21].

```
1  oracleClones(N, distance)
2      tree = new MetricTree
3      forall f ∈ N :
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 3: Clone oracle algorithm

4. FEATURES AND CHARACTERISTICS

Oracle computation has been performed on two open-source Java systems, namely Tomcat 5.5 and Eclipse 3.3. Tomcat [2] 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. Eclipse [1] is a widely used IDE for many common programming languages. Computation for Tomcat's oracle has been executed on an Intel Core 2 Duo, 2.16 GHz clock, 4 GB RAM, under Linux Fedora 13. For Eclipse's oracle, a cluster of 32 Opteron, 2.00 GHz clock, 5 GB RAM, under Adelie Linux was used. In both cases, software used has been compiled with g++ 4.4.4.

The size, number of fragments and some interesting statistics on fragments size are presented in Table 2. The number of LOCs reported in that figure excludes comments in source file. 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 difficulty in computing the distance.

Fragments correspond to syntactic blocks, methods, and classes. Blocks can be nested, but don't otherwise intersect or overlap. Statement sequences within the same block are not further split into sub-sequences, at the moment. If required, sub-sequences within a block could be easily built from chains of statements and used as candidate fragments.

In Table 2 and in all reported figures and experiments, fragments refer to blocks larger than 70 tokens, which is a threshold that has been chosen as a lower bound of significance 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.

The reported characteristics of the oracles in Table 3 are computed by $rangeQuery$ on a $radius$ corresponding to a threshold ϵ using a $distance$ equal to 0.3 in equation 3.

This value seems a reasonable upper bound for the $distance$ threshold. However, the reported oracles may easily be shrunk to oracles valid for a $distance$ less than 0.3. An entry in an oracle is a tuple $< f_1, f_2, \delta >$ where f_1 and f_2 are the fragment in the

System	Tomcat	Eclipse
Version	5.5	3.3
LOC	130K	1.3M
Fragments	5084	129 258
Av. Length of fragments in tokens	341.29	403.64
Max. Length of fragments in tokens	19999	128168

Table 2: System features

clone relation and δ is the distance between them. To compute an oracle for a $distance$ less than 0.3, it suffices to linearly scan all tuples and selecting only those with the desired threshold. Hence, oracles for smaller values of $distance$ may be easily obtained from our results.

Keeping in mind that Tomcat's oracle was computed with a single core and Eclipse's oracle was computed on a 32 cores cluster, we conclude from Table 3 that computing Eclipse's oracle is many thousand times more complex than computing Tomcat's one even though their LOC size is within a factor of 10. Two factors may explain this huge difference: the great difference in the number of fragments and the difference in the average length of fragments. The definite cause behind the practical performance difference has not been investigated in this paper and is left to further investigation.

Figure 4 shows the frequency distribution of clones pairs with respect to the distance between clone fragments. The distance has been computed as described in section 3. Tomcat's distribution looks bell-curved with a maximum in the class $[0.20, 0.25)$. Eclipse distribution is increasing monotonically. In both cases, the distribution suggests type-3 clones may form a bigger set than the union of type-1 and 2 clones sets. This justifies the interest one may have in studying type-3 clones. However, we did not investigate the distributions of separate types of clones and this is left to further research.

An example of the clone found in Tomcat's oracle is shown in Figure 5. The length of the fragments are respectively 177 for 5a and 145 for 5b. The reported Levenshtein distance is 34 or equivalently 0.192 according to our oracle's definition. This example exhibits many of the characteristics of a type-3 clone. First, there's an added prefix in the longer fragment. Second, there's an added suffix in the longer fragment. Finally, most of the code in the smaller fragment have been embedded in a while loop in the larger fragment. These differences have been correctly identified by the alignment as shown in the figure.

5. DISCUSSION

The two resulting oracles presented in section 4 are complete oracles under the Levenshtein metric. This means that the sets of clone pairs found satisfying the condition of being closer than chosen distance threshold would be the same as the ones produced by the pairwise comparison on the same condition. These are the first oracles produced automatically on large systems under a well-accepted similarity criterion. Considering that the Levenshtein metric is good for matching sequences of tokens, then the oracles are of good quality for type-3 clone detection assessment.

With respect to hand-made oracles cite in section 2, our method has the main advantage of being able to deal with large volume in reasonable time. For example, the oracle produced by Bellon took 77 hours of work or roughly 10 days of regular work (assuming men-work of 7.5 hours a day). In that time, some 6 500 candidates were classified. Our oracle for Eclipse, despite needing intensive computation on a small cluster, classified over 8 billions candidates under the Levenshtein metric in 6.2 day (assuming machine-work of 24 hours a day). This is a tremendous gain over the hand-made method. Moreover, our technique doesn't rely on proposed clone candidates from existing tools, artificial clones nor injected clones. Thus, it has a greater degree of independence.

The oracles from both system are relatively small compare to the total number of fragments pairs. In Tomcat, there are 12 920 986 fragments pairs from which 2933 are clones with respect to our cloning criterion for a cloning rate of 0.02%. In Eclipse, from 8 353 750 653 fragments pairs there are 316 728 clones according to our cloning criterion for a cloning rate of 0.004%. These numbers show a very low rate of true positive in the search spaces and partly explain why the use of metric trees for oracle computation allows good space reduction,. The low proximity rate among fragments lead to high space pruning in metric trees.

As of the date this paper is written, the computation of Levenshtein based oracles for systems over 200 KLOCs seems to take more than one week on a single desktop computer. However, mainstream desktop computers now have four or eight cores. Since the proposed computation doesn't produce interdependent results, splitting of the computation may be hand-made and even ported to different physical machines with minimal effort. Since we showed the computation can be done on 32 cores with expected reasonable computation time, between four to eight mainstream computers would be enough to reproduce our results. Thus, the required computational power is only a small limiting factor.

The provided oracles rely on the Levenshtein metric and the assumption that it produces good quality clones. However, humans may question some clones found with the Levenshtein metric depending of the underlying clone definition. We also noticed that this is especially true for clones with higher distances, while closer clones under Levenshtein distance seem more consistent with respect to human validation. This suggests that hand-made classification is relevant even with the possibility of automated oracles. In fact, hand-made classification may be complementary to these oracles, in some cases.

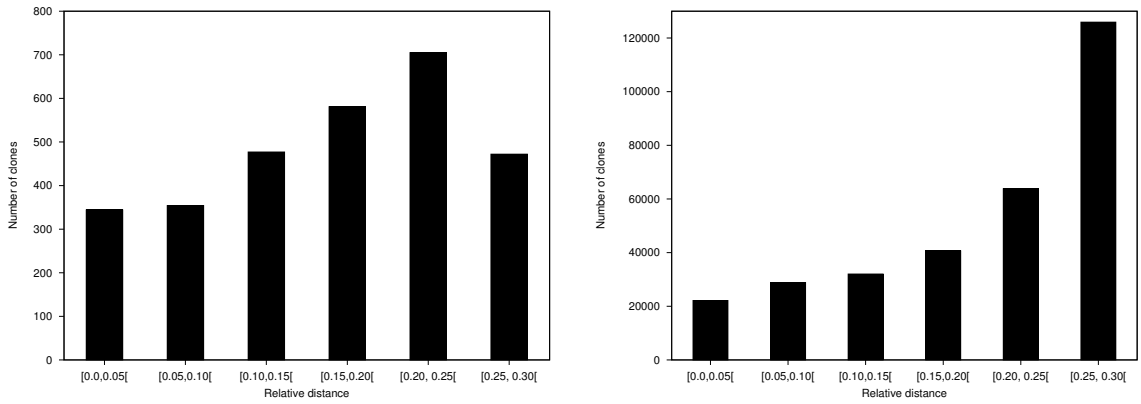
There are some threats to the validity of these results. We cannot certify the correctness of the software used to compute the oracle. Since we cannot compare our results to the pairwise comparison to validate our results, confidence in the oracles must be drawn from their characteristics that seem plausible. Also, the choice of the Levenshtein metric limit our oracles to tokens-based features. Even though syntactically similar clones always show similar tokens type representation and would be part of the oracle, the oracles may report clones that do not share enough syntactic features to be considered clones from a syntax point of view. Thus, the oracles may contain some false-positive according to other definitions of clones. Nevertheless, our oracles cannot have false-negatives for syntactic clones because of the previously mentioned link between tokens type based and syntactic based clones. Therefore, it should be a reasonable tool of comparison even for those types of clones. A concrete limit of this technique to build oracles would be dealing with semantic clones, but since they are not classified as type-3, it doesn't mitigate our claim of providing a good oracle for type-3 clones. Still, this points out the need to find other ways of computing oracles for clones of higher type than type-3.

Another threat is the suitability of the Levenshtein metric. Even though the Levenshtein metric is the optimal mathematical alignment of token sequences, more empirical studies on its practical relevance should be performed.

Other threats to validity may concern the nature and the number of systems experimented on. The presented results show a clear sensibility to the systems characteristics, therefore perfor-

Systems	Reported pairs	Time (h.)	Max. length of clones in tokens	Avg. length of clones in tokens
Tomcat	2933	2.67	6534	181.99
Eclipse	316 728	148.80	115 141	222.31

Table 3: Total number of clone pairs (a,b) with Levenshtein distance no more than 0.3*max(len(a),len(b)) along with execution time



9**(a)** Tomcat

9**(b)** Eclipse

Figure 4: Distribution of clones with respect to their relative distance

mance cannot be easily generalized to different systems. Moreover, bigger systems must be investigated to verify further scalability of the method. All code fragments were written in the Java programming language. Since different languages may have different cloning characteristics, the method may not apply as well as in this case. However, the proposed approach is tokens-based and uses only lexical features. Since writing lexical analyzers is simpler than writing syntactic analyzers and string alignment is a language independent operation, our technique should easily be applied to many imperative programming and scripting languages.

6. CONCLUSION

We have shown a method to produce oracles for clone detection systems with size up to some MLOCs without relying on statistical properties. The technique is fully automated. The resulting oracles are of good quality for type-3 clones comparison since it is based on the Levenshtein metric and allows for precision and recall assessment. The oracles for Java systems Tomcat and Eclipse were produced and some features of these oracles were presented.

Future research will attempt to improve the oracle's computation time either by using alternatives to metric trees or investigating different parallel architectures such as super clusters and GPU. The objective is to reduce the computation time until it becomes feasible and acceptable to produce oracles for large systems.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Eclipse. http://www.eclipse.org.

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

[3] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. In *International Symposium on Empirical Software Engineering*, 2005.

[4] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing clone evolution in the linux kernel. *Information and Software Technology*, pages 755–765, 2002.

[5] B. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering - IEEE Computer Society Press*, 2007.

[6] 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.

[7] H. Basit, S. Pugliesi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2007.

[8] 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.

[9] 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.

[10] 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.

[11] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society Press, 2007.

[12] S. Ducasse, O. Nierstrasz, and M. Rieger. On 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        }
```

9(**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         }
```

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

Figure 5: A clone example from Tomcat's oracle with distance 0.192. Alignment differences are shown in bold.

effectiveness of clone detection by string matching. *International Journal on Software Maintenance and Evolution: Research and Practice - Wiley InterScience*, (18):37–58, 2006.

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

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

[15] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. *Software Maintenance, IEEE International Conference on*, 0:1–9, 2010.

[16] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *Workshop on Source Code Analysis and Manipulation*, 2007.

[17] E. Juergens, F. Deissenboeck, and B. Hummel. Clone detective - a workbench for clone detection research. In *Proceedings of the International Conference on Software Engineering*, pages 603–606. IEEE Computer Society Press, 2009.

[18] T. Kamiya. Variation analysis of context-sharing identifiers with code clone. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*. IEEE Computer Society Press, 2008.

[19] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. volume 28, pages 654–670. IEEE Computer Society Press, 2002.

[20] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2005.

[21] T. Lavoie, M. Eilers-Smith, and E. Merlo. Challenging cloning related problems with GPU-based algorithms. In *IWSC10 Proceedings of the 4th International Workshop on Software Clones*, pages 25–32, 2010.

[22] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 107, Washington, DC, USA, 2001. IEEE Computer Society.

[23] 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.

[24] T. Mende, F. Beckerwert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *CSMR '08 Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering - IEEE Computer Society Press*, pages 163–172, 2008.

[25] T. Mende, R. Koschke, and F. Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution*, 21(2):143–169, march-april 2009.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] C. K. Roy and J. R. Cordy. A mutation / injection-based automatic framework for evaluating clone detection tools. In *ICSTW09 International Conference on Software Testing, Verification and Validation Workshops*, pages 157–166, 2009.

[31] 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.