e-Informatica Software Engineering Journal
Volume 20, Issue 1, pages: 1–21
DOI: 10.37190/e-Inf260105

BibTeX

Index-Based Type-3 Clone Detection

Zdenek Tronicek*
*Corresponding author: `zdenek.tronicek@oneonta.edu`

Article info

Keywords:
clone detection
code clones
near-miss clones

Submitted: 14 Dec. 2025
Revised: 16 Feb. 2026
Accepted: 20 Apr. 2026
Available online: 23 Apr. 2026

Abstract

Context: Clone detection is a common task in software engineering. Type-3 clones are fragments of code that can be slightly different in structure.
Objective: The article presents a new algorithm for Type-3 clone detection, its open-source implementation called *DrDupLex3*, and novel open-source tools that can be used for the automated assessment of Type-3 clones and to prepare training sets for machine learning-based clone detectors.
Method: The algorithm for Type-3 clone detection builds upon the index of source code used in *DrDupLex*, the most accurate Type-2 clone detector to date.
Results: A comparison with three state-of-the-art clone detectors (*NiCad*, *CloneWorks*, and *SourcererCC*) shows that *DrDupLex3* is able to outperform them in precision, recall, and running time. It reported no false positives and found all clones reported by *NiCad*, *CloneWorks*, and *SourcererCC*.
Conclusions: The presented clone detector outperforms three state-of-the-art competitors in a scenario that can be easily repeated because it is based on tools for automated assessment of Type-3 clones.

1. Introduction

Code clones are a well-established research area in software engineering with applications in software maintenance, refactoring, vulnerability detection, program comprehension, evolution analysis, and plagiarism detection. One fundamental problem in this area is the clone detection problem, which is to identify all code clones in a given code base. Despite the simplicity of its definition, the problem is nontrivial and has been attracting the attention of researchers for more than two decades. This is partially because it is not a single problem, but rather a suite of problems, which differ in the definition of the clone. In this article, a clone is a set of code fragments that are similar to each other. Clones can be reported either as clone pairs (two code fragments) or clone classes (two or more code fragments). We typically distinguish four categories of clones [1–3]:
– Type-1 (exact) clones are code fragments that are identical or differ only in white spaces and comments.
– Type-2 (renamed) clones are fragments of code that are either Type-1 clones or differ in identifiers and literals. If identifiers are systematically renamed, the clone is called parameterized.

© 2026 The Authors. Published by Wrocław University of Science and Technology Publishing House.
This is an open access article under the CC BY-SA 4.0 license international.

- Type-3 (near-miss) clones are fragments of code that are either Type-2 clones or differ in code structure. They appear in the code base when we copy a code fragment and add, modify, or remove statements.
- Type-4 (semantic) clones are fragments of code that are semantically equivalent, i.e., they provide the same functionality. There is no requirement for syntactic similarity.

The first three types of clones are related because Type-2 clones involve Type-1 clones, and Type-3 clones involve Type-2 clones; however, Type-4 clones are different because for two code fragments to be a Type-4 clone, the code fragments do not need to be syntactically similar. This is why algorithms for identifying Type-4 clones are typically very different from algorithms for identifying Type-1, Type-2, and Type-3 clones.

Another parameter of clone detection is code granularity. Common values of granularity are “statements”, “methods” (or “functions”), and “files”. While other granularities are also possible, most application areas require one of these three.

There are dozens of approaches to clone detection described in the literature, and they can be classified by various criteria. One common approach is to classify them based on how they compare code fragments [4–7]:

- Text-based clone detectors compare the textual representation of code [8, 9].
- Token-based clone detectors translate code to a sequence of tokens and compare sequences of tokens [10, 11].
- Tree-based clone detectors represent code as a tree, such as an abstract syntax tree, and compare trees or their representation [12].
- Graph-based clone detectors translate code to a graph, such as a Program Dependence Graph, and compare graphs [13–15].
- Metrics-based clone detectors use code metrics to detect clones [16].
- Hybrid clone detection techniques combine two or more approaches mentioned above [17].

Commonly, clone detectors in the same class of this classification share the same traits. For example, token-based clone detectors typically scale better than graph-based ones because building a graph code representation requires more resources than tokenization.

Like many other areas of software engineering, clone detection was also affected by development in the area of artificial intelligence [18]. Multiple researchers have explored various machine learning-based tools capable of classifying code fragments as clones or nonclones. Although these tools are called “clone detectors”, they do not detect clones like traditional clone detectors. They only answer the question “Are these two code fragments a clone?” and sometimes provide a similarity score. In practice, they can hardly replace traditional clone detectors, which detect all Type-3 clones that are within the specified similarity.

This article deals with Type-3 clone detection. Two code fragments are considered a Type-3 clone if their similarity is above some specified threshold. There are various ways in which the similarity of code fragments can be measured, and it is common that the algorithm used for computing similarity is bound to the clone detection method. For example, text-based clone detectors typically compute the longest common subsequence of sequences of lines, and *Deckard* (see section Related Work for details) computes the distance of characteristic vectors in Euclidean space. The diversity of algorithms used for computing code similarity in clone detectors makes a comparison of clone detectors challenging because it is not always clear how to transform one similarity into the other. In addition, code fragments that are similar in one similarity can be very dissimilar in a different similarity. One approach to overcome these difficulties is to divide clone detectors into categories based on how they compute code similarity and compare them only with

clone detectors in the same category. This article introduces a category of "clone detectors that measure code similarity by the edit distance of token sequences" and presents the best clone detector in this category. The edit distance (also known as the Levenshtein distance) is the minimum number of operations substitution, insertion, and deletion required to transform one sequence of tokens into the other. This distance is particularly intuitive for code fragments that are not very distant, such as when they differ only in a few operators. On the other hand, for code fragments that differ in whole lines, the edit distance that allows block operations may be a better choice.

The rest of this article is organized as follows: Section 2 presents the motivation for this research, Section 3 summarizes the scientific contribution of this article, Section 4 describes a novel algorithm for Type-3 clone detection, Section 5 introduces an open-source clone detector called *DrDupLex3*, Section 6 discusses comparison of *DrDupLex3* with state-of-the-art competitors, Section 7 introduces open-source tools for automated evaluation of clone detectors, Section 8 outlines conducted experiments, Section 9 addresses threats to validity, Section 10 states the main limitations of the presented approach, Section 11 briefly mentions the most important related work, and Section 12 concludes.

2. Motivation

The primary motivation for research on code clones stems from their applications in software engineering. Code clone detection facilitates various tasks critical to software quality and maintainability, including:

- Refactoring [19, 20]: By identifying code clones, developers can refactor them into reusable functions or methods, reducing code duplication and improving code maintainability.
- Code maintenance [21, 22]: Detecting clones helps ensure consistent updates across all code instances, minimizing the risk of introducing bugs due to inadvertent modifications in one location but not others.
- Vulnerable code detection [23–26]: Clone detection algorithms play a crucial role in identifying potentially vulnerable code in various areas, such as cryptocurrencies [27,28], Android applications [29–31], and malware [32].
- License compliance [33]: Identifying inter-project code clones can help developers avoid potential license infringement issues, particularly when dealing with slightly modified code snippets.

While solutions to these tasks typically involve Type-2 clone detection, they can often benefit from Type-3 clone detection, too.

The diversity in applications highlights the need for recognizing clone detection as a multifaceted problem with multiple solutions, each offering different results. Categorizing clone detectors enables researchers to conduct more scientific comparisons and facilitates developers in selecting the most appropriate approach for a specific task.

3. Contribution

The scientific contribution of this article is a novel algorithm for detecting Type-3 clones and a suite of open-source tools for an automated evaluation of Type-3 clone detectors. The algorithm identifies all Type-3 clones within the codebase; because this comprehensive

detection is significantly more demanding than simple pairwise classification, machine learning-based clone classifiers are not direct competitors to this approach.

The algorithm leverages the index of the source code employed in *DrDupLex* [34], the most accurate Type-2 clone detector to date. This index, a trie data structure built on sequences of tokens, facilitates the identification of Type-1 and Type-2 clones and the elimination of candidates that are not within the specified edit distance. Unlike previous approaches, the index can be easily updated upon a change in source code, which makes the presented solution suitable for reporting clones "online", i.e., upon each change in source code. The implementation of the algorithm achieves 100% precision and 100% relative recall (no known clone was missed). The presented evaluation is automated and can be easily reproduced.

4. Algorithm

Although the algorithm description utilizes Java programming language terminology, the underlying concept applies to other programming languages as well. The description references parsing for syntactic unit identification; however, full parsing is not required. Any parsing capable of detecting syntactic unit boundaries is sufficient. The algorithm can operate on diverse syntactic units, such as methods or statements, and their combinations. It consists of five steps (see also Figure 1):

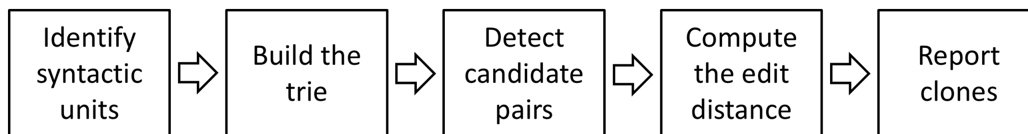

Figure 1. The clone detection has five steps: 1. Identify syntactic units and translate them into sequences of tokens; 2. Build the trie on sequences of tokens; 3. Group the sequences of tokens by length and identify pairs of sequences that can be within the specified distance; 4. Compute the edit distance for each candidate pair; 5. Report the pairs that are within the specified distance

1. Identify syntactic units (this typically involves parsing) and translate them into sequences of tokens; for example, the statement "b = a + 1" is translated into the sequence "ID = ID + INT".
2. Build a trie on these sequences (see Figure 2 for an example); this is the same trie as is used to detect Type-2 clones in *DrDupLex*; Type-2 clones have the same sequences of tokens and thus they end in the same node of the trie.

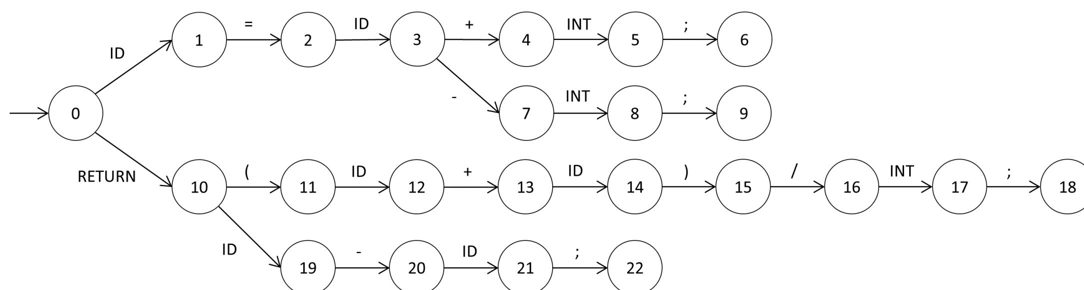

Figure 2. The trie for statements "b = a + 1 ;", "c = a – 1 ;", "y = x + 2 ;", "return (a + b) / 2 ;", "return a – b ;", and "return x – y ;"

3. Group the sequences of tokens by length and identify pairs of sequences that can be within the specified distance; for example, sequences of lengths 20 and 23 can be within the distance 3, but sequences of lengths 20 and 24 cannot be within that distance because 4 operations are needed just to make the lengths of the sequences equal.
4. Compute the edit distance for each candidate pair; the edit distance is computed between two sequences of tokens; *DrDupLex3* uses the algorithm "dynamic programming" [35], which is terminated as soon as it is clear that the sequences are not a clone; multiple threads are employed to compute the distances concurrently.
5. Report pairs that are within the specified distance.

The trie constructed in step 2 facilitates the identification of syntactic units for which the edit distance needs to be computed. As an example, let's consider the trie in Figure 2 and the maximum distance 1. Since "b = a + 1 ;" and "y = x + 2 ;" are a Type-2 clone, as well as "return a – b ;" and "return x – y ;", we get only four sequences of tokens:
– ID = ID + INT ;
– ID = ID – INT ;
– RETURN (ID + ID) / INT ;
– RETURN ID – ID ;

They form six pairs for which we need to compute the edit distance. When we group the sequences by length, we get one sequence with 5 tokens, two sequences with 6 tokens, and one sequence with 9 tokens. Pairs whose length difference exceeds the maximum distance can be eliminated because the length difference is the lower bound for the edit distance. So, we need to compute the edit distance between three pairs only:
– "ID = ID + INT ;" and "ID = ID – INT ;"
– "ID = ID + INT ;" and "RETURN ID – ID ;"
– "ID = ID – INT ;" and "RETURN ID – ID ;"

This is substantially less than the 10 pairs required if we omit to identify Type-2 clones and only cluster the methods by length.

The presented trie may seem similar to the suffix tree used in *CCFinder* [10] (see Related Work for details); however, there are two significant differences: (i) The suffix tree in *CCFinder* is built on the file level (the files are translated into sequences of tokens and the suffix tree is built on the concatenated sequences of tokens), which means that *CCFinder* does not know where syntactic units begin and end. This is probably why it does not report Type-3 clones. (ii) The trie can be modified upon a change in the source code. We can delete an original sequence of tokens and add a new sequence of tokens in time linear in the length of this sequence (see [34] for details). In contrast, a suffix tree requires rebuilding from scratch. This makes the presented algorithm suitable for reporting clones upon each change in source code. For example, an Integrated Development Environment can report clones while the developer writes code.

5. Implementation

The algorithm is implemented for the Java programming language in an open-source tool called *DrDupLex3*[1]. Like *DrDupLex*, it leverages JavaParser[2] to parse source code and builds the index, which is either plain or compressed, either in memory or on secondary storage,

[1]https://github.com/tronicek/DrDupLex3
[2]https://javaparser.org

and either method-level or statement-level. *DrDupLex3* uses the naïve algorithm and blind renaming (see [34] for explanation of these terms). The edit distance between two token sequences $a = a_1a_2 \dots a_m$ and $b = b_1b_2 \dots b_n$ is computed by "dynamic programming", which utilizes a matrix D defined as follows: $D_{i,0} = i$, for $0 \le i \le m$, $D_{0,j} = j$, for $1 \le j \le n$, and

$$D_{i,j} = \begin{cases} D_{i-1,j-1} & \text{if } a_i = b_j, \\ 1 + min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) & \text{otherwise, for } 1 \le i \le m \text{ and } 1 \le j \le n. \end{cases}$$

When the matrix is calculated, the value of $D_{m,n}$ is the edit distance between a and b.

The run and output of *DrDupLex3* are controlled by a configuration file, which can contain the following parameters:

- *type* specifies the type of clones; accepted values are "2" (only Type-2 clones), "3" (only Type-3 clones that are not Type-2), and "2+3" (all Type-3 clones).
- *maxDistance* specifies the maximum edit distance between two code fragments; for example, if *maxDistance* is 3, only the code fragments with an edit distance of 3 or less are reported as a Type-3 clone.
- *level* specifies the granularity of the index; accepted values are "method" and "statement".
- *compressed* specifies whether the index is compressed or not; accepted values are "true" and "false".
- *persistent* specifies whether the index is built in main memory or on secondary storage; accepted values are "true" and "false". The persistent index is implemented using memory-mapped files and can be much larger than the nonpersistent index because it never needs to be fully loaded in memory.
- *minSize* specifies the minimum number of lines; for example, if *minSize* is 5, the code fragment must have at least 5 lines to be reported.
- *ignoreUnaryAtLiterals* specifies how the unary plus and minus are treated; accepted values are "true" and "false". If true, "return 1" and "return -1" are considered a Type-2 clone.
- *ignoreAnnotations* specifies whether annotations in code are taken into account; accepted values are "true" and "false".
- *treatNullAsLiteral* specifies how "null" is treated; accepted values are "true" and "false". If true, "return null" and "return 0" are considered a Type-2 clone.
- *treatSuperThisAsIdentifier* specifies whether "super" and "this" are treated as identifiers; accepted values are "true" and "false". If true, "this()" and "print()" are considered a Type-2 clone, as well as "this.size = 1" and "p.length = 2".
- *threads* specifies the number of concurrent threads employed to compute edit distances between code fragments.
- *batchFileSize* specifies how many files are processed before the index in memory is merged with the persistent index. A higher value means better throughput, but also higher memory usage.

Parameters *ignoreUnaryAtLiterals*, *ignoreAnnotations*, *treatNullAsLiteral*, and *treatSuperThisAsIdentifier* were added to facilitate comparing with other tools. For example, if a tool ignores annotations, we can configure *DrDupLex3* to ignore them as well.

6. Evaluation

Search algorithms are typically evaluated in terms of precision and recall. Precision is defined as the fraction of retrieved occurrences that are relevant:

$$precision = \frac{|\{relevant\} \cap \{retrieved\}|}{|\{retrieved\}|}$$

Recall is defined as the fraction of relevant occurrences that are retrieved:

$$recall = \frac{|\{relevant\} \cap \{retrieved\}|}{|\{relevant\}|}$$

Two types of errors can occur: false positives and false negatives. A false positive arises when an identified code fragment is not a clone, and a false negative occurs when a genuine clone remains undetected. Both false positives and false negatives are undesirable and should be eliminated.

I considered the following clone detectors for comparing with *DrDupLex3* (see section Related Work for details): *CCAligner*, *CloneWorks*, *Code2Img*, *Deckard*, *NiCad*, *NIL*, *Siamese*, *SourcererCC*, *StoneDetector*, *TACC*, and *Tamer*. However, none of them compute similarity in the same way as *DrDupLex3*:

- *CCAligner* [36] computes asymmetric Dice similarity between code windows;
- *CloneWorks* [37] computes Jaccard similarity between code fragments (but does not report it);
- *Code2Img* [38] computes Jaccard similarity of vectors that encode images created based on the adjacency matrix of the normalized abstract syntax tree;
- *Deckard* [39] computes similarity using the Euclidean distance of vectors;
- *NiCad* [40] computes similarity from the longest common subsequence of two sequences of code lines;
- *NIL* [41] computes similarity from the longest common subsequence of two token sequences;
- *Siamese* [42] computes similarity based on n-grams;
- *SourcererCC* [43] computes Jaccard similarity between code fragments (but does not report it);
- *StoneDetector* [44] computes similarity from the textual representations of paths in dominator trees;
- *TACC* [45] computes Jaccard similarity of vectors;
- *Tamer* [46] computes similarity based on the corresponding subtrees in abstract syntax trees.

How can we compare *DrDupLex3* with a tool that computes a different similarity? Let's say we select a benchmark project and let the tool identify all clones with a specified similarity threshold. Then we let *DrDupLex3* identify all clones with the equivalent edit distance and compare the outputs. This can work if we can transform the tool's similarity into the edit distance. Unfortunately, no such transformation is available for the similarities mentioned above. Another approach can be based on estimates. Let's say tool A identifies all clones with a similarity threshold s, and from s we are able to compute d such that all these clones have the edit distance at most d. Then we let *DrDupLex3* identify all clones with the maximum distance d and check that it reported all clones found by tool A. And then the opposite way: We let *DrDupLex3* identify all clones within a specified distance and

then check that tool A found them all. This can work if we are able to estimate the edit distance based on a given similarity threshold.

As an example, let's consider the similarity measured by the percentage of identical lines. Let ed be the edit distance of two code fragments, $lines$ the number of lines of the larger fragment, and $lcssim$ the percentage of identical lines (i.e., $lcssim$ is the ratio of the number of identical lines and $lines$), and let's try to estimate ed for a given value of $lcssim$. If there is at least one different line ($lcssim < 100\%$), ed can be any value ≥ 1 because a line can contain any number of changes. Now let's estimate $lcssim$ for a given value of ed. If code changes are distributed so that there is no more than one change per line (provided that $ed \leq lines$), $lcssim$ is minimal, and if all code changes are on one line, $lcssim$ is maximal. Thus

$$\frac{max(0, lines - ed)}{lines} \leq lcssim \leq \frac{lines - 1}{lines}.$$

This can be used to make estimates in some particular cases. For example, if the edit distance of two code fragments is 1, only one line can be different, and if $lines$ is at least 5, the similarity of the code fragments is at least $4/5 = 80\%$.

As another example, let's consider the Jaccard similarity, which calculates similarity between two sequences of tokens as the ratio of the number of shared tokens and the number of tokens in the longer sequence. For example, the Jaccard similarity of expressions "x + y" and "x + y + z" is 3/5 and of expressions "x – 1" and "1 – x" is 1. Let f_1, f_2 be two sequences of tokens, ed the edit distance of f_1 and f_2, $jsim$ the Jaccard similarity of f_1 and f_2, $shared$ the number of shared tokens between f_1 and f_2, and $maxlen$ the maximum of $|f_1|$ and $|f_2|$, and let's estimate ed for a given value of $jsim$. From the equation $jsim = shared/maxlen$ and the estimate $maxlen - shared \leq ed \leq maxlen$, we get

$$maxlen(1 - jsim) \leq ed \leq maxlen.$$

Now let's estimate $jsim$ for a given ed. For $ed = 1$, $jsim = (maxlen - 1)/maxlen$, and for $ed \geq 2$,

$$\frac{maxlen - ed}{maxlen} \leq jsim \leq 1.$$

The upper bound is tight, for example, when all changes are substitutions, such as for sequences $t_1\ t_2\ t_3\ t_4\ t_5$ and $t_5\ t_4\ t_3\ t_2\ t_1$ (we substitute t_5 for t_1, t_4 for t_2, and so on).

NIL may look like a good candidate for comparing because it works with tokens; however, it does not do any lexical analysis and uses sequences of input characters as tokens. For example, a statement "a = (int) (b + 0.5);" is converted into a sequence "a = (int) (b + 0.5) ;" with no information that "a", "int", and "b" are identifiers and "0.5" is a double literal. A consequence is that *NIL* does not recognize Type-2 clones that are not Type-1 clones. It reports only Type-1 clones and a specific subset of Type-3 clones (those created by inserting statements into a Type-1 clone). In other words, *NIL* detects different types of clones than *DrDupLex3* and so a direct comparison of these tools would be uninformative.

Other problems exist with the clone detectors mentioned above: *CCAligner* hangs up and does not provide any output, *Deckard* works only on small code bases, *Code2Img*, *StoneDetector*, and *Tamer* produce an output that does not contain enough information for identifying code fragments, *Siamese* compiles and runs but does not work, and *TACC* does not compile.

Based on these observations, I decided to continue with *NiCad* (version 6.2), *CloneWorks* (version 0.3), and *SourcererCC* (version on GitHub with the latest commit on May 17, 2022) and show they do not report some of the clones reported by *DrDupLex3*. In addition to differences in similarity, the tools differ in how the size of clones is determined: *NiCad* reports pretty-printed lines after removing comments, *CloneWorks* and *SourcererCC* report no lines, and *DrDupLex3* reports real lines.

Since *DrDupLex* can outperform *NiCad*, *CloneWorks*, and *SourcererCC* in precision and recall when searching for Type-2 clones [34] and *DrDupLex3* reports the same Type-2 clones as *DrDupLex*, I focused on Type-3 clones that are not Type-2 clones. *NiCad* uses the longest common subsequence algorithm to compute similarity and *DrDupLex3* computes the edit distance between sequences of tokens, which may result in significant differences. As an example, Figure 3 and 4 show code fragments that are very similar for *DrDupLex3* and only moderately similar for *NiCad*. In Figure 3, *NiCad* considers the return statements different and disregards the fact that they are different in only one character. In Figure 4, *NiCad* does not recognize "int" as an identifier, which results in differences in the first lines of code. Since the second lines are also different and the lines "}" are not counted, the similarity of code fragments is either 25% (if *NiCad* runs with parameter abstract = none) or 50% (if *NiCad* runs with parameter abstract=literal).

A convenient approach to comparing with *NiCad* would be to modify *DrDupLex3* so that it reports the same similarity as *NiCad*; however, this is not easy to achieve because of how *NiCad* preprocesses source code before computing the longest common subsequence: it removes comments and formats source code by a pretty printer that is sometimes hard to reproduce (see Figure 5).

CloneWorks and *SourcererCC* use Jaccard similarity to measure similarity of code fragments, which is sometimes very different from similarity based on edit distance. As an example, Figure 6 shows code fragments that are a Type-2 clone for *CloneWorks* and *SourcererCC* and Type-3 clone for *DrDupLex3*.

```
double root1(int a, int b, int c) {
return(-b+Math.sqrt(b*b-4*a*c))/(2*a);
}
```

```
double root2(int a, int b, int c) {
return(-b-Math.sqrt(b*b-4*a*c))/(2*a);
}
```

Figure 3. Two methods that are assessed very differently by *DrDupLex3* and *NiCad*. For *DrDupLex3*, the methods differ in one token, and so it reports a clone with edit distance 1. For *NiCad*, the methods share two lines, but the lines "}" are not counted, and so it reports a clone with similarity 50%

```
public String getBaseTypeName() {
  debugCodeCall("getBaseTypeName");
  checkClosed();
  return "NULL";
}
```

```
public int getMaxFieldSize() {
  debugCodeCall("getMaxFieldSize", 10);
  checkClosed();
  return 0;
}
```

Figure 4. Two methods that are assessed very differently by *DrDupLex3* and *NiCad*. For *DrDupLex3*, the methods differ in two tokens, and so *DrDupLex3* reports a clone with edit distance 2. For *NiCad*, the first lines are different because it does not recognize "int" as an identifier, and the fourth lines are either different or equal, depending on the "abstract" parameter. Thus, *NiCad* reports a clone with similarity either 25% (with abstract=none) or 50% (with abstract=literal)

```
public int gcd(int a, int b) {
    while (a != b)
        if (a < b)
            b -= a;
        else
            a -= b;
    return a;
}
```

```
public int gcd (int a, int b) {
    while (a != b) if (a < b) b -= a;
    else a -= b;
    return a;
}
```

Figure 5. A fragment of code (on the left) and its formatting by *NiCad* (on the right)

```
int poly1(int x) {
  return x * x - x - 1;
}
```

```
int poly2(int x) {
  return 1 - x - x * x;
}
```

Figure 6. Two methods that are assessed very differently by *DrDupLex3* and *CloneWorks* and *SourcererCC*. *DrDupLex3* reports a clone with edit distance 4. *CloneWorks* and *SourcererCC* use Jaccard similarity and report a clone with similarity 1

7. Tools

The evaluation was done on BigCloneBench [47] (the reduced version of IJaDataset 2.0), a well-established benchmark for clone detection. BigCloneBench consists of a large-scale source code repository and a clone database; however, only the source code repository is utilized in this evaluation. The rationale for excluding the clone database is that it has been shown to contain less than 3.4% of the actual Type-1 and Type-2 clones [34]. Furthermore, BigCloneBench's Type-3 clones use *NiCad* similarity.

I designed and implemented the following tools in open-source projects *EvalTool*[3] and *CloneDistance*[4] (they all work with an input XML file that contains clones):

– *FilterDistance* selects clones based on the XML attribute *distance*; the minimum and maximum distance must be specified.
– *FilterLines* selects clones based on the number of real lines of code; the minimum and maximum number of lines must be specified.
– *FilterSimilarity* selects clones based on the XML attribute *similarity*; the minimum and maximum similarity must be specified.
– *EditDistance* computes the edit distance of code fragments and stores it in the XML attribute *distance*.

I also modified the open-source project *CloneChecker*[5] [34] so that it can check Type-3 clones. When it runs with parameter type=3, it classifies clones as Type-1, Type-2, Type-3, "not a clone", or invalid. Classification "invalid" is used when the code fragments do not parse. If the code fragments parse but their distance differs from the value in the XML attribute *distance*, the clone is classified as "not a clone". In addition to these tools, the evaluation leverages the tools that were already described in [34]:

– *Counter* counts clones in the input XML file.
– *Diff* subtracts two XML files with clones and reports those in the first file and not in the second file.

[3] https://github.com/tronicek/EvalTool
[4] https://github.com/tronicek/CloneDistance
[5] https://github.com/tronicek/CloneChecker

- *Separator* separates clones in the input XML file into clone pairs. For example, a clone of A, B, and C is separated into clone pairs {A, B}, {B, C}, and {A, C}.
- *Sourcer* extracts code fragments from a code base and inserts them into the input XML file. The code fragments can then be checked visually or by *CloneChecker*.

8. Experiments

To evaluate precision, I ran *NiCad* with parameters rename = blind, abstract = literal (to treat all literals equal), threshold = 0.01, and minsize = 20, added code fragments to the output XML file (by *Sourcer*), computed distances between code fragments (by *EditDistance*), and classified the clones (by *CloneChecker*). Then I repeated that for *CloneWorks* with parameters rename-blind, abstract literal (to treat all literals equal), min-similarity 0.99, and min-lines 20. Next, I ran *SourcererCC* with parameters MIN_-TOKENS = 40 and threshold = 9.9, filtered clones shorter than 20 lines (by *FilterLines*), added code fragments to the output XML file, computed distances between code fragments, and classified the clones. Finally, I ran *DrDupLex3* with parameters type = 2 + 3, level = method, maxDistance = 1, and minSize = 20, separated clones into clone pairs (by *Separator*), added code fragments to the output XML file, and classified the clones (see Figure 7 and Table 1).

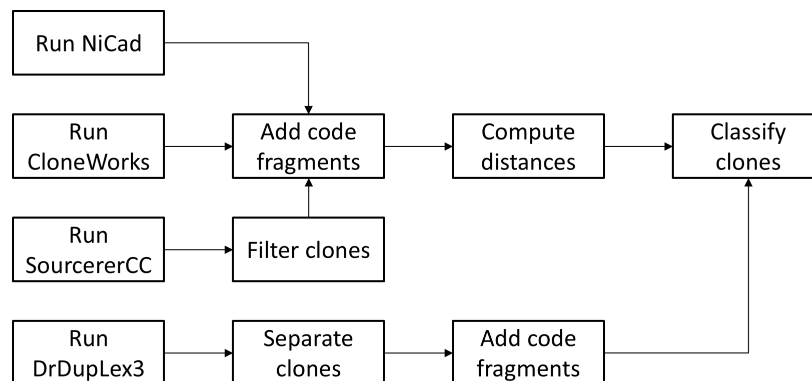

Figure 7. The flowchart of how precision was evaluated. There is a path for each clone detector. For example, the path for *NiCad* is "Run *NiCad*, Add code fragments, Compute distances, Classify clones". The following tools were used: *Sourcer* to add code fragments, *EditDistance* to compute distances between code fragments, *CloneChecker* to classify clones, *FilterLines* to filter clones, and *Separator* to separate clones into clone pairs

To evaluate Type-3 precision, I ran *NiCad* with parameters rename=blind, abstract=literal, threshold=0.05, and minsize=20, filtered out Type-2 clones (by *FilterSimilarity*), added code fragments to the output XML file, computed distances between code fragments, and classified the clones. The clones classified as Type-2 were incorrectly reported with a similarity of less than 100. Then I ran *DrDupLex3* with parameters type = 3, level = method, maxDistance = 4, and minSize = 20, added code fragments to the output XML file, and classified the clones (see Figure 8 and Table 2).

To evaluate recall, I ran *NiCad* with parameters rename = blind, abstract = literal, threshold = 0.10, and minsize = 20, added code fragments to the output XML file, computed distances between code fragments (this eliminated invalid clones), and filtered out clones

Table 1. The clone pairs reported in BigCloneBench by *NiCad* (rename = blind, abstract = literal, threshold = 0.01, minsize = 20), *CloneWorks* (rename-blind, abstract literal, min-similarity 0.99, min-lines 20), *SourcererCC* (MIN_TOKENS = 40, threshold = 9.9), and *DrDupLex3* (type = 2 + 3, level = method, maxDistance = 1, minSize = 20) and their classification by *CloneChecker*. The precision is computed based on this classification. *NiCad* seemingly reported more Type-1 clones than *DrDupLex3* because it ignores annotations and uses pretty-printed lines. For instance, two methods that differ only in annotations are reported as a Type-1 clone by *NiCad*. *SourcererCC* reported more Type-3 clones than the other tools, but only 409 of them had distance = 1

Tool	Reported	Type 1	Type 2	Type 3	Invalid	Precision
NiCad	258,845	226,503	31,836	435	71	99.97%
CloneWorks	258,870	224,216	30,949	3,633	72	99.97%
SourcererCC	235,737	213,847	2,142	7,200	12,548	94.68%
DrDupLex3	267,653	224,496	38,367	4,790	0	100%

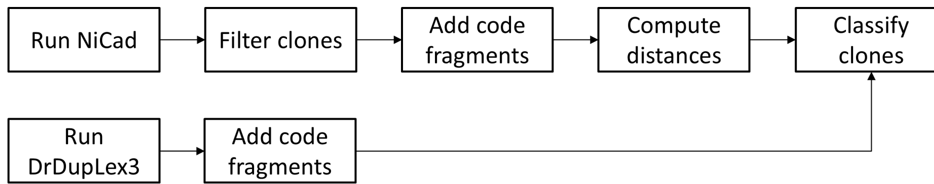

Figure 8. The flowchart of how Type-3 precision was evaluated. There is a path for each clone detector. The path for *NiCad* is "Run *NiCad*, Filter clones, Add code fragments, Compute distances, Classify clones" and the path for *DrDupLex3* is "Run *DrDupLex3*, Add code fragments, Classify clones". The following tools were used: *FilterSimilarity* to filter clones, *Sourcer* to add code fragments, *EditDistance* to compute distances between code fragments, and *CloneChecker* to classify clones

Table 2. The clone pairs reported as Type-3 clone pairs in BigCloneBench by *NiCad* (rename=blind, abstract=literal, threshold = 0.05, minsize = 20) and *DrDupLex3* (type = 3, level = method, maxDistance = 4, minSize = 20) and their classification by *CloneChecker*. Type-2 clones were filtered out by *FilterSimilarity*. *CloneWorks* and *SourcererCC* are not involved because they do not report similarity

Tool	Reported	Type 1	Type 2	Type 3	Invalid	Type-3 Precision
NiCad	20,032	0	284	19,744	4	98.56%
DrDupLex3	23,659	0	0	23,659	0	100%

with a distance greater than 1. Then I ran *DrDupLex3* with parameters type = 2 + 3, level = method, maxDistance = 1, minSize = 6, and ignoreAnnotations = true (because *NiCad* ignores annotations), subtracted the output from *NiCad*'s output (by *Diff*), and got 13 clones. All 13 clones were methods with less than 15 lines of code that were pretty-printed on more than 20 lines. Then I ran *DrDupLex3* with parameters type = 2 + 3, level = method, maxDistance = 1, and minSize = 40, separated the clones into pairs, ran *NiCad* with parameters rename = blind, abstract = literal, threshold = 0.20, and minsize = 10, subtracted the output from *DrDupLex3*'s output, added code fragments to the output XML file, and classified the clones (see Figure 9 and 10).

Then I ran *CloneWorks* with parameters rename-blind, abstract literal, min-similarity 0.90, and min-lines 20, added code fragments to the output XML file, computed distances between code fragments (this eliminated invalid clones), filtered out clones with a distance greater than 1, subtracted the output of *DrDupLex3* with parameters type = 2 + 3,

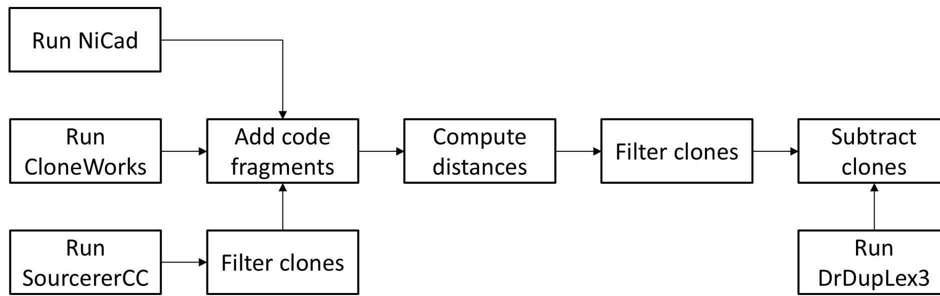

Figure 9. The flowchart describing how the output of *DrDupLex3* was subtracted from the output of *NiCad*, *CloneWorks*, and *SourcererCC* (the results are in Table 3 and 4). Each clone detector has its own path. For example, the path for *NiCad* is "Run *NiCad*, Add code fragments, Compute distances, Filter clones, Subtract clones detected by *DrDupLex3*". The following tools were used: *Sourcer* to add code fragments, *EditDistance* to compute distances between code fragments, *FilterDistance* to filter clones, and *Diff* to subtract clones

level = method, maxDistance = 1, minSize = 6, and ignoreAnnotations = true (because *CloneWorks* ignores annotations), and got an empty XML file. Then I ran *CloneWorks* with parameters rename-blind, abstract literal, min-similarity 0.90, and min-lines 10, subtracted the output from the output of *DrDupLex3* with parameters type = 2 + 3, level = method, maxDistance = 1, and minSize = 40, added code fragments to the output XML file, and classified the clones.

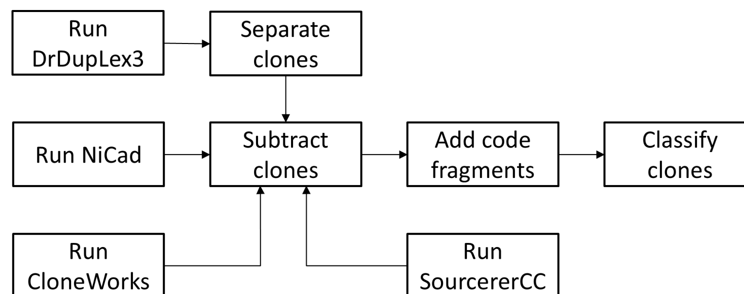

Figure 10. The flowchart describing how the output of *NiCad*, *CloneWorks*, and *SourcererCC* was subtracted from the output of *DrDupLex3* (the results are in Table 3 and 4). Each clone detector has its own path. For example, the path for *NiCad* is "Run *NiCad*, Subtract clones from the output of *DrDupLex3*, Add code fragments, Classify clones". The following tools were used: *Separator* to separate clones into clone pairs, *Diff* to subtract clones, *Sourcer* to add code fragments, and *CloneChecker* to classify clones

Then I ran *SourcererCC* with parameters MIN_TOKENS = 40 and threshold = 9, filtered out clones shorter than 20 lines, added code fragments to the output XML file, computed distances between code fragments (this eliminated invalid clones), filtered out clones with a distance greater than 1, subtracted the output of *DrDupLex3* with parameters type = 2 + 3, level = method, maxDistance = 1, minSize = 6, and ignoreAnnotations = true (because *SourcererCC* ignores annotations), and got an empty XML file. Then I ran *SourcererCC* with parameters MIN_TOKENS = 40 and threshold = 8, subtracted the output from the output of *DrDupLex3* with parameters type = 2 + 3, level = method, maxDistance = 1, and minSize = 40, added code fragments to the output XML file, and classified the clones.

Table 3. The output of tool B was subtracted from the output of tool A. The output of *NiCad*, *CloneWorks*, and *SourcererCC* in the column "Tool A" was filtered by *FilterDistance* and *FilterLines*. The column "Difference" is the number of clones with distance = 1 reported by tool A and missed by tool B. These values were used to calculate recall in Table 4

Tool A	Tool B	Difference
NiCad, rename=blind, abstract=literal, threshold=0.10, minsize=20 *FilterDistance*	*DrDupLex3*, type=2+3, level=method, maxDistance=1, minSize=6, ignoreAnnotations=true	13
DrDupLex3, type=2+3, level=method, maxDistance=1, minSize=40	*NiCad*, rename=blind, abstract=literal, threshold=0.20, minsize=10	101
CloneWorks, rename-blind, abstract literal, min-similarity 0.90, min-lines 20, *FilterDistance*	*DrDupLex3*, type=2+3, level=method, maxDistance=1, minSize=6, ignoreAnnotations=true	0
DrDupLex3, type=2+3, level=method, maxDistance=1, minSize=40	*CloneWorks*, rename-blind, abstract literal, min-similarity 0.90, min-lines 10	159
SourcererCC, MIN_TOKENS=40, threshold=9, *FilterLines*, *FilterDistance*	*DrDupLex3*, type=2+3, level=method, maxDistance=1, minSize=6, ignoreAnnotations=true	0
DrDupLex3, type=2+3, level=method, maxDistance=1, minSize=40	*SourcererCC*, MIN_TOKENS=40, threshold=8	8,435

Table 4. The clone pairs reported in BigCloneBench by *DrDupLex3* (type=2+3, level=method, maxDistance=1, minSize=40) and missed by *NiCad* (rename=blind, abstract=literal, threshold=0.20, minsize=10), *CloneWorks* (rename-blind, abstract literal, min-similarity 0.80, min-lines 10), and *SourcererCC* (MIN_TOKENS=40, threshold=8) (see Table 3) and their classification by *CloneChecker*. The values in the columns "Recall" and "Type-3 Recall" are relative recalls calculated as the ratio of clones reported by the tool to clones reported by *DrDupLex3* (it reported 67,688 clones: 61,340 Type-1 clones, 3,379 Type-2 clones, and 2,969 Type-3 clones)

Tool	Missed	Type 1	Type 2	Type 3	Recall	Type-3 Recall
NiCad	101	88	11	2	99.85%	99.93%
CloneWorks	159	119	37	3	99.77%	99.90%
SourcererCC	8,435	5,623	133	2,679	87.54%	9.77%

To compare the running time and consumed memory, I used a virtual machine with CPU Intel Xeon Silver 4310 at 2.10 GHz, 16 GB of memory, 64-bit Oracle Linux, and Oracle Java 17.0.9. I measured the running time and memory consumption by prefixing each command with "/usr/bin/time -v". Table 5 contains the elapsed (wall clock) time and the maximum resident set size as it was reported by the command ‘‘/usr/bin/time -v’’.

9. Threats to validity

The major threat to the validity of the presented evaluation is hidden in the configuration parameters. Since *NiCad* and *DrDupLex3* report different lines (*NiCad* reports

Table 5. Running time and peak memory usage of *NiCad* (rename=blind, abstract=literal, threshold=0.01, minsize=20), *CloneWorks* (rename-blind, abstract literal, min-similarity 0.99, min-lines 20), *SourcererCC* (MIN_TOKENS=75, threshold=9.9), and *DrDupLex3* (type=2+3, level=method, maxDistance=1, minSize=20) for processing BigCloneBench. The values in the columns “Time” and “Memory” are arithmetic averages of three measurements. The time is in minutes and seconds

Tool	Reported Clones	Time	Memory
NiCad	258,845	35:06	214 MB
CloneWorks	258,870	13:37	2,472 MB
SourcererCC	262,433	48:28	4,220 MB
DrDupLex3	267,653	10:12	1,669 MB

pretty-printed lines and *DrDupLex3* reports real lines), *NiCad* can miss a clone because the code fragments have fewer than the minimum number of pretty-printed lines. To mitigate this concern, I manually inspected all Type-3 clones missed by *NiCad* and verified that all of them should have been reported.

Another threat is the limited set of clone detectors used in evaluation. Although *NiCad*, *CloneWorks*, and *SourcererCC* represent state-of-the-art clone detectors, a comprehensive evaluation should involve more than three tools. Unfortunately, it turned out that out of eleven considered clone detectors, only five can be readily used, and out of these five, one tool is unable to process BigCloneBench, and one tool detects only a specific subset of Type-3 clones.

10. Limitations

The presented approach has three key limitations: (i) It requires a robust parser to identify syntactic units; (ii) The source code must be syntactically correct so that it can be parsed; (iii) It computes the edit distance for each pair of code fragments individually, which may be slow for large projects and big distances. An efficient algorithm for computing the edit distances in a set of strings would be preferable, but unfortunately, no such algorithm is available in the literature to date. Since computing edit distances can be parallelized, *DrDupLex3* employs multiple threads.

The algorithm uses the edit distance to measure code similarity. For a different code similarity, the algorithm may become less efficient. For example, for the normalized edit distance, i.e., the edit distance divided by the total number of tokens, the similarity needs to be computed for all candidate pairs because no pairs can be eliminated based on the difference in token counts. The trie is still useful, though, because it identifies Type-2 clones, which can be eliminated from candidate pairs.

The presented implementation detects clones only in the Java source code. While parsing of other programming languages is conceptually similar (a lexer and parser are typically employed), no other languages have been tried.

11. Related work

Over the past three decades, a substantial body of research has emerged in the area of clone detection. Numerous publications have described various techniques for identifying syntactic (Type-1, Type-2, and Type-3) and semantic (Type-4) clones; however, not all syntactic clone detection techniques are capable of detecting Type-3 clones.

Baxter et al. [12] described a method for detecting exact and near-miss clones that used abstract syntax trees and hashing. The similarity of code fragments was computed from the number of shared and different nodes in abstract syntax trees.

Kamiya, Kusumoto, and Inoue [10] presented *CCFinder*, which converted source code into sequences of tokens, concatenated these sequences, built a suffix tree, and used it to identify Type-2 clones. It does not support Type-3 clones, though.

Li et al. [11] presented *CP-Miner*, the tool that uses data mining techniques to efficiently identify copy-pasted code in a large code base. It converts source code into a sequence of tokens and searches for frequent subsequences.

Jiang et al. [39] presented an approach based on characteristic vectors in Euclidean space and its implementation for Java and C called *Deckard*. It parses source code, captures the syntactic information in numerical vectors, and clusters the vectors based on their proximity. Unfortunately, the method does not scale well, and the tool is unable to process BigCloneBench.

Kraft, Bonds, and Smith [48] described an approach for detecting clones that span multiple languages and its implementation for the .NET Framework.

Roy and Cordy [40] presented *NiCad*, the accurate clone detector based on parsing and pretty printing. It formats source code, renames identifiers, and computes the longest common subsequence of sequences of lines.

Uddin et al. [49] researched the effectiveness of *simhash*, the fingerprint-based data similarity measurement technique, for clone detection.

Sajnani et al. [43] presented *SourcererCC*, the token-based clone detection tool, which uses an optimized inverted index and filtering heuristics to achieve exceptional scalability.

Svajlenko and Roy [37] presented *CloneWorks*, the fast and scalable clone detector, particularly suitable for large-scale near-miss clone detection. Unfortunately, it does not report the similarity of clones.

Saini et al. [50] focused on Type-3 clones with a similarity of less than 70% and presented *Oreo*, the tool built on machine learning, information retrieval, and software metrics, which can detect these clones. However, it is unclear how similarity is computed.

Wang et al. [36] presented *CCAligner*, the tool that uses a sliding window as a basic unit of matching and demonstrated that it was the best-performing large-gap clone detection tool at that time. Unfortunately, the project provided on GitHub does not work.

Ragkhitwetsagul and Krinke [42] introduced a novel clone detection technique that incorporates multiple code representations, query reduction, and a customized ranking function. They implemented the technique in a tool called *Siamese* and demonstrated that the tool scales to hundreds of millions of code lines. The implementation builds on *Elasticsearch*, the open-source distributed full-text search engine. Unfortunately, the project provided on GitHub does not work. I contacted the authors, but they were not able to help me.

Nakagawa, Higo, and Kusumoto [41] presented *NIL*, the clone detector that uses n-grams and an inverted index. It is particularly suitable for large-variance clones, which arise by inserting or deleting many statements in a copied fragment of code.

Schäfer, Amme, and Heinze [51] described a tool called *Stubber*, which compiles source code into bytecode without dependencies. It can be used for clone detection on the bytecode level.

Amme, Heinze, and Schäfer [44] described a new approach to clone detection based on control flow analysis and dominator trees and presented its implementation named *StoneDetector*. It parses source code and builds abstract syntax trees, transforms them into control flow graphs and dominator trees, encodes the dominator trees' paths into textual representations, and computes the similarity of these textual representations. The output

is a sequence of lines, one line for each reported clone. The line consists of eight values: folder 1, file name 1, start line 1, end line 1, folder 2, file name 2, start line 2, and end line 2. Unfortunately, the folder is not a full path to the file, but only the parent folder, which is not sufficient for identifying the file.

Wang et al. [45] compared 12 clone detection algorithms and proposed a new tool called *TACC*, which is reportedly superior to *CCAligner*, *SourcererCC*, *NiCad*, *NIL*, and *Siamese*. Unfortunately, the source code in its GitHub repository does not compile. In addition, it seems that the output of the detector is just pairs of method identifiers with no additional information, which is insufficient for identifying code fragments.

Hu et al. [46] introduced *Tamer*, a tree-based clone detector that reportedly outperforms ten state-of-the-art clone detectors. Unfortunately, its GitHub repository contains only three Java files, and it looks more like a proof of concept than a real clone detector. In addition, the sample output in the repository does not provide enough information to identify code fragments.

Wang et al. [52] presented *CCStokener*, a fast and accurate clone detection tool with the semantic token, and described an experiment in which *CCStokener* achieved the best recall in detecting moderate Type-3 clones. Unfortunately, its output does not provide enough information to identify code fragments.

Hu et al. [38] presented *Code2Img*, a tree-based code clone detector, which converts the abstract syntax trees into images, transforms each image into a one-dimensional vector, and calculates the Jaccard similarity of these vectors. The authors claim that *Code2Img* outperforms eight state-of-the-art clone detectors; however, the result is difficult to verify because the tool output consists of text files that contain only integers, which is insufficient for identifying code fragments.

Lavoie and Merlo [53] described an original method for constructing clone oracles based on the Levenshtein metric. It would be interesting to check their oracles with *CloneChecker*, but unfortunately, they do not provide it for download.

Keivanloo, Zhang, and Zou [33] proposed a threshold-free method for Type-3 clone classification. The method is based on K-means clustering.

White et al. [54] presented a deep learning-based approach to clone classification that can classify all types of clones.

Li et al. [55] described *CCLearner*, a neural-network classifier, which uses supervised learning, and trained it on a part of BigCloneBench.

Gao et al. [56] described a tree embedding technique for clone classification and its implementation called *TECCD*. The advantage of this technique is that it does not require a training set.

Several survey and comparison papers provide an overview of the state-of-the-art tools for clone detection at the time of their publication.

Rattan, Bhatia, and Singh [4] published a systematic literature review based on 213 papers selected out of 2039 papers published in 11 journals and 37 conferences and workshops.

Ragkhitwetsagul, Krinke, and Clark [57] evaluated 30 code similarity detection techniques and tools using five experimental scenarios for Java source code.

Walker, Cerny, and Song [58] published a study on open-source clone detection tools and benchmarks.

Zhang and Sakurai [26] provided a survey of clone detection from a security perspective. They describe applications of code clones in vulnerable code detection.

Zakeri-Nasrabadi et al. [59] published a systematic literature review on code similarity measurement and evaluation techniques. They investigated 80 tools working mainly on Java, C, and C++.

Kaur and Rattan [18] wrote a systematic review on using machine learning in clone classification.

12. Conclusion

The article described a new algorithm for Type-3 clone detection and its open-source implementation called *DrDupLex3*. The evaluation revealed that six out of eleven state-of-the-art clone detectors (*CCAligner*, *Code2Img*, *StoneDetector*, *Tamer*, *Siamese*, *TACC*) cannot be readily used, and one (*Deckard*) is not able to process BigCloneBench. The comparison of *DrDupLex3* with *NiCad*, *CloneWorks*, and *SourcererCC* demonstrated that *DrDupLex3* detects more clones than these tools. *DrDupLex3* builds on the index structure used in *DrDupLex*, the most accurate Type-2 clone detector to date. When the index structure is modified upon a change in source code, only the distances of modified pairs need to be computed, which makes the presented approach suitable for detecting clones "online". Furthermore, the article described the tools for automated evaluation of Type-3 clone detectors, which can facilitate the preparation of training sets for machine learning-based clone classifiers. For example, the tools can prepare a training set of clones and nonclones.

CRediT authorship contribution statement

Zdenek Tronicek: Conceptualization, Methodology, Software, Validation, Investigation, Writing – Original Draft Preparation, Writing – Review & Editing.

Declaration of competing interest

The author declares no competing interests.

Funding

The research has no funding sources.

References

[1] C.K. Roy and J.R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, Vol. 541, No. 115, 2007, pp. 64–68.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, 2007, pp. 577–591.

[3] C.K. Roy, J.R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, Vol. 74, No. 7, 2009, pp. 470–495.

[4] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, Vol. 55, No. 7, 2013, pp. 1165–1199.
[5] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, Vol. 137, No. 10, 2016, pp. 1–21.
[6] Q.U. Ain, W.H. Butt, M.W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, Vol. 7, 2019, pp. 86 121–86 144.
[7] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, "Code clone detection – A systematic review," *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, 2021, pp. 645–655.
[8] B.S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM Journal on Computing*, Vol. 26, No. 5, 1997, pp. 1343–1362.
[9] J.R. Cordy and C.K. Roy, "The NiCad clone detector," in *19th International Conference on Program Comprehension*. IEEE, 2011, pp. 219–220.
[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, 2002, pp. 654–670.
[11] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, 2006, pp. 176–192.
[12] I.D. 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*, 1998, pp. 368–377.
[13] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–309.
[14] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International Static Analysis Symposium*, 2001, pp. 40–56.
[15] C. Liu, C. Chen, J. Han, and P.S. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 872–881.
[16] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of International Conference on Software Maintenance*, Vol. 96, 1996, p. 244.
[17] E. Kodhai and S. Kanmani, "Method-level code clone detection through LWH (Light Weight Hybrid) approach," *Journal of Software Engineering Research and Development*, Vol. 2, 2014, pp. 1–29.
[18] M. Kaur and D. Rattan, "A systematic literature review on the use of machine learning in code clone research," *Computer Science Review*, Vol. 47, 2023, p. 100528.
[19] W. Wang and M.W. Godfrey, "Recommending clones for refactoring using design, context, and history," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 331–340.
[20] N. Tsantalis, D. Mazinanian, and G.P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, Vol. 41, No. 11, 2015, pp. 1055–1090.
[21] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *31st International Conference on Software Engineering*, 2009, pp. 485–495.
[22] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Symposium on Security and Privacy*. IEEE, 2012, pp. 48–62.
[23] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: Software vulnerability discovery using code clone verification," *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 6, 2016, pp. 1900–1917.
[24] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.
[25] B. Bowman and H.H. Huang, "VGRAPH: A robust vulnerable code clone detection system using code property triplets," in *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 53–69.

[26] H. Zhang and K. Sakurai, "A survey of software clone detection from security perspective," *IEEE Access*, Vol. 9, 2021, pp. 48 157–48 173.
[27] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the Ethereum smart contract ecosystem," in *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 2020, pp. 654–675.
[28] Q. Hum, W.J. Tan, S.Y. Tey, L. Lenus, I. Homoliak et al., "Coinwatch: A clone-based approach for detecting vulnerabilities in cryptocurrencies," in *International Conference on Blockchain (Blockchain)*. IEEE, 2020, pp. 17–25.
[29] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*. Springer, 2012, pp. 37–54.
[30] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 175–186.
[31] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, "Droidcc: A scalable clone detection approach for Android applications to detect similarity at source code level," in *42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 2018, pp. 100–105.
[32] M.R. Farhadi, B.C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 2014, pp. 78–87.
[33] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous Java repository," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 201–210.
[34] Z. Tronicek, "Indexing source code and clone detection," *Information and Software Technology*, Vol. 144, 2022, p. 106805.
[35] R.A. Wagner and M.J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, Vol. 21, No. 1, 1974, pp. 168–173.
[36] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C.K. Roy, "CCAligner: A token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1066–1077.
[37] J. Svajlenko and C.K. Roy, "Fast and flexible large-scale clone detection with CloneWorks," in *IEEE/ACM 39th International Conference on Software Engineering (Companion Volume)*, 2017, pp. 27–30.
[38] Y. Hu, Y. Fang, Y. Sun, Y. Jia, Y. Wu et al., "Code2Img: Tree-based image transformation for scalable code clone detection," *IEEE Transactions on Software Engineering*, 2023.
[39] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering*, 2007, pp. 96–105.
[40] C.K. Roy and J.R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.
[41] T. Nakagawa, Y. Higo, and S. Kusumoto, "NIL: Large-scale detection of large-variance clones," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 830–841.
[42] C. Ragkhitwetsagul and J. Krinke, "Siamese: Scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, Vol. 24, No. 4, 2019, pp. 2236–2284.
[43] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy, and C.V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[44] W. Amme, T.S. Heinze, and A. Schäfer, "You look so different: Finding structural clones and subclones in Java source code," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 70–80.

[45] Y. Wang, Y. Ye, Y. Wu, W. Zhang, Y. Xue et al., "Comparison and evaluation of clone detection techniques with different code representations," in *IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2023, pp. 332–344.

[46] T. Hu, Z. Xu, Y. Fang, Y. Wu, B. Yuan et al., "Fine-grained code clone detection with block-based splitting of abstract syntax tree," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 89–100.

[47] J. Svajlenko and C.K. Roy, "Evaluating clone detection tools with BigCloneBench," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 131–140.

[48] N.A. Kraft, B.W. Bonds, and R.K. Smith, "Cross-language clone detection," in *20th International Conference on Software Engineering & Knowledge Engineering*, 2008, pp. 54–59.

[49] M.S. Uddin, C.K. Roy, K.A. Schneider, and A. Hindle, "On the effectiveness of Simhash for detecting near-miss clones in large scale software systems," in *18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 13–22.

[50] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C.V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 354–365.

[51] A. Schäfer, W. Amme, and T.S. Heinze, "Stubber: Compiling source code into bytecode without dependencies for Java code clone detection," in *15th International Workshop on Software Clones (IWSC)*. IEEE, 2021, pp. 29–35.

[52] W. Wang, Z. Deng, Y. Xue, and Y. Xu, "CCStokener: Fast yet accurate code clone detection with semantic token," *Journal of Systems and Software*, Vol. 199, 2023, p. 111618.

[53] T. Lavoie and E. Merlo, "Automated type-3 clone oracle using Levenshtein metric," in *Proceedings of the 5th International Workshop on Software Clones*, 2011, pp. 34–40.

[54] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[55] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[56] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang et al., "TECCD: A tree embedding approach for code clone detection," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 145–156.

[57] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, Vol. 23, 2018, pp. 2464–2519.

[58] A. Walker, T. Cerny, and E. Song, "Open-source tools and benchmarks for code-clone detection: past, present, and future trends," *ACM SIGAPP Applied Computing Review*, Vol. 19, No. 4, 2020, pp. 28–39.

[59] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," *Journal of Systems and Software*, 2023, p. 111796.

Authors and affiliations

Zdenek Tronicek
e-mail: zdenek.tronicek@oneonta.edu
ORCID: https://orcid.org/0000-0002-2737-1317
State University of New York, Oneonta, NY, USA