NIL: Large-Scale Detection of Large-Variance Clones

Tasuku Nakagawa
Osaka University
Suita, Osaka, Japan
t-nakagw@ist.osaka-u.ac.jp

Yoshiki Higo
Osaka University
Suita, Osaka, Japan
higo@ist.osaka-u.ac.jp

Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
kusumoto@ist.osaka-u.ac.jp

ABSTRACT

A code clone (in short, clone) is a code fragment that is identical or similar to other code fragments in source code. Clones generated by a large number of changes to copy-and-pasted code fragments are called large-variance (modifications are scattered) or large-gap (modifications are in one place) clones. It is difficult for general clone detection techniques to detect such clones and thus specialized techniques are necessary. In addition, with the rapid growth of software development, scalable clone detectors that can detect clones in large codebases are required. However, there are no existing techniques for quickly detecting large-variance or large-gap clones in large codebases. In this paper, we propose a scalable clone detection technique that can detect large-variance clones from large codebases and describe its implementation, called NIL. NIL is a token-based clone detector that efficiently identifies clone candidates using an N-gram representation of token sequences and an inverted index. Then, NIL verifies the clone candidates by measuring their similarity based on the longest common subsequence between their token sequences. We evaluate NIL in terms of large-variance clone detection accuracy, general Type-1, Type-2, and Type-3 clone detection accuracy, and scalability. Our experimental results show that NIL has higher accuracy in terms of large-variance clone detection, equivalent accuracy in terms of general clone detection, and the shortest execution time for inputs of various sizes (1–250 MLOC) compared to existing state-of-the-art tools.

KEYWORDS

Clone Detection, Large-Variance Clone, Scalability

ACM Reference Format:
Tasuku Nakagawa, Yoshiki Higo, and Shinji Kusumoto. 2021. NIL: Large-Scale Detection of Large-Variance Clones. In *Proceedings of The 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A code clone (in short, clone) is a code fragment that is identical or similar to other code fragments in source code. Clones are generated by copying, pasting, and modifying code fragments for reuse [16, 27]. Clones are a major problem in software maintenance because they lead to bug propagation. Therefore, clone detection techniques, which automatically detect clones in the target codebase, are essential. Many clone detection techniques have been proposed [1, 20, 36] and applied in applications, such as refactoring [6, 21, 35], debugging [3, 12, 19], and mining software repositories [11, 14, 22].

It is important for clone detection techniques to detect clones that have been heavily edited. A clone generated by inserting or deleting a large number of statements in one place in a copy-and-pasted code fragment is called a large-gap clone [37]. Such clones are common in software development and should thus be detected along with general clones. Wang et al. pointed out that it is difficult for existing clone detectors to detect large-gap clones; they proposed a technique for detecting such clones and presented its implementation, called CCAligner [37]. Wu et al. pointed out that CCAligner targets only clones in which statement insertion or deletion is made in a single place and cannot detect clones in which modifications are scattered [40]. They called the latter type of clone large-variance clones and proposed LVMapper, a clone detector for large-variance clones.

It is also important for clone detection techniques to be scalable. Highly scalable clone detectors are required for analyzing large-scale projects or source files in an inter-project repository. Many scalable clone detectors have been proposed [18, 29, 34]. To achieve scalable clone detection, SourcererCC [29] and CloneWorks [34] use heuristics to reduce the number of code block comparisons needed to detect clones, and SAGA [18] uses a GPU to parallelize its clone detection process.

However, clone detectors that can detect clones with a large number of edits fail for large inputs [37] or require a long time to detect clones [40]. Scalable clone detectors target only identical or strongly similar clones (near-miss clones). They are incapable of detecting large-variance clones, in which many statements have been inserted or deleted. Therefore, the scalable detection of large-variance clones is challenging.

In this paper, we propose a scalable technique for detecting large-variance clones and describe its implementation, called NIL[1], which uses an N-gram representation, an inverted index, and the longest common subsequence (LCS). NIL is a token-based clone detector. One of the features of large-variance clones is that the order of many tokens is preserved (i.e., the common subsequence between token sequences of large-variance clones is long). Hence, to detect large-variance clones, NIL measures the similarity between the token sequences of two code fragments based on the LCS. In addition, large-variance clones share many consecutive tokens. Hence, for scalable clone detection, NIL uses an N-gram representation of token sequences and an inverted index to reduce the number of

[1]A clone detector using N-gram, Inverted index, and LCS.

ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```java
protected int run(Commandline cmd) {
  try {
    Execute exe = new Execute(new LogStreamHandler(this,
      Project.MSG_INFO, Project.MSG_WARN);
    exe.setAntRun(getProject());
    exe.setWorkingDirectory(getProject().getBaseDir());
    exe.setCommandline(cmd.getCommandline);
    exe.setVMLauncher(false);
    return exe.execute();
  } catch (java.io.IOException e) {
    throw new BuildException(e, getLocation());
  }
}
```

(a) Clone A

```java
protected int run(Commandline cmd) {
  try {
    Execute exe = new Execute(new LogStreamHandler(this,
      Project.MSG_INFO, Project.MSG_WARN);
    if (serverPath != null) {
      String[] env = exe.getEnvironment();
      if (env == null) {
        env = new String[0];
      }
      String[] newEnv = new String[env.length + 1];
      System.arrayCopy(env, 0, newEnv, 0, env.length);
      newEnv[env.length] = "SSDIR=" + serverPath;
      exe.setEnvironment(newEnv);
    }
    exe.setAntRun(getProject());
    exe.setWorkingDirectory(getProject().getBaseDir());
    exe.setCommandline(cmd.getCommandline);
    exe.setVMLauncher(false);
    return exe.execute();
  } catch (java.io.IOException e) {
    throw new BuildException(e, getLocation());
  }
}
```

(b) Clone B

Figure 1: Example of large-gap clones

code block comparisons needed to detect clones. First, NIL transforms code blocks extracted from source files into token sequences and creates an inverted index from the N-gram representation of the token sequences. Next, it identifies the clone candidates for each code block using the code block and the inverted index. Finally, it verifies the clone candidates by measuring the similarity between the code block and the clone candidates.

We evaluate NIL's (1) large-variance clone detection accuracy, (2) general Type-1, Type-2, and Type-3 clone detection accuracy, and (3) scalability. We compared NIL with existing state-of-the-art tools, namely LVMapper [40], CCAligner [37], SourcererCC [29], and NiCad [4]. The experimental results show that NIL has high precision of 87% in large-variance clone detection. It also has high recall of 100%, as determined in our evaluation of large-variance clone detection using a mutation technique. In general clone detection, the accuracy of NIL is equivalent to that of the existing tools. In addition, we confirmed that NIL has high scalability; it can detect clones faster than the existing tools for large inputs (codebases with 250 MLOC).

The main contributions of this paper are as follows.

(1) We proposed a scalable technique for detecting large-variance clones. The proposed technique identifies clone candidates efficiently by using an N-gram representation of token sequences and an inverted index and verifies clone candidates precisely by measuring the similarity between token sequences based on the LCS.
(2) We implemented the proposed technique as a tool, called NIL. The executable file is available at https://github.com/kusumotolab/NIL.
(3) We evaluated the usefulness of NIL through three experiments. The results show that NIL has high large-variance clone detection accuracy, high scalability, and equivalent general clone detection accuracy compared to that of existing state-of-the-art tools. We have published our experiment data to facilitate replication studies.

The remainder of this paper is organized as follows. Section 2 describes the definition of general clones, large-gap clones, and large-variance clones. Section 3 describes NIL in detail. Section 4 gives an overview of our evaluation and presents the results. Section 5 describes threats to validity. Section 6 reviews related studies. Section 7 concludes this paper with a discussion and suggestions for future work.

2 PRELIMINARIES

2.1 Definition

A code fragment is a consecutive segment of source code. It can be represented by the tuple ($file\_name$, $start\_line$, end_line). A code block is a code fragment within braces. This study treats a function, which is a code block, as a clone detection unit, as done in previous studies [29, 37, 40]. Clones are code fragments identical or similar to other code fragments in source code. A pair of similar code fragments is called a clone pair. Clones are classified based on the degree of the similarity between them as follows.

Type-1 is an exact copy without modifications (except for white space and comments).
Type-2 is a syntactically identical copy; only variable, types, or function identifiers are different.
Type-3 is a copy with further modifications; statements have been changed, added, or removed.

The minimum length of clones is the minimum number of lines that a code fragment must be to be treated as a clone. It is often set to six lines or 50 tokens [1].

```
protected String getPrompt(InputRequest request) {
  String prompt = request.getPrompt();
  if (request instanceof MultipleChoiceInputRequest) {
    StringBuffer sb = new StringBuffer(prompt);
    sb.append("(");
    Enumeration e = ((MultipleChoiceInputRequest) request)
        .getChoices().elements();
    boolean first = true;
    while (e.hasMoreElements()) {
      if (!first) {
        sb.append(",");
      }
      sb.append(e.nextElement());
      first = false;
    }
    sb.append(")");
    prompt = sb.toString();
  }
  return prompt;
}
```

(a) Clone A

```
protected String getPrompt(InputRequest request) {
  String prompt = request.getPrompt();
  String def = request.getDefaultValue();
  if (request instanceOf MultipleInputChoiceRequest) {
    StringBuilder sb = new StringBuilder(prompt).append("(");
    boolean first = true;
    for (String next : ((MultipleInputChoiceRequest) request)
      .getChoices()) {
      if (!first) {
        sb.append(",");
      }
      if (next.equals(def)) {
        sb.append('|');
      }
      sb.append(next);
      if (next.equals(def)) {
        ab.append('|');
      }
      first = false;
    }
    sb.append(")");
    return sb.toString();
  }
  else if (def != null) {
    return prompt + "[" + def + "]";
  }
  else {
    return prompt;
  }
}
```

(b) Clone B

Figure 2: Example of large-variance clones

2.2 Large-gap clone

A large-gap clone is a clone generated by inserting or deleting a large number of statements in one place in a copy-and-pasted code fragment. Figure 1 shows an example of large-gap clones. In this example, a 10-line if-statement is inserted into Clone A (lines 4–13 of Clone B). Wang et al. pointed out that existing clone detectors are incapable of large-gap clone detection because most target to the detection of near-miss clones [37]. Wang et al. defined large-gap clone as follows. Consider two code blocks c_1 and c_2 with LOC values of L_1 and L_2, respectively, where $L_1 \leq L_2$. Let $\lambda = L_i/L_j$ (i.e., λ is the ratio of the code lengths of two code blocks). If c_1 and c_2 are Type-3 clones and the corresponding $\lambda \leq 0.7$, then these clones are large-gap clones. The clone pair shown in Figure 1 fits the definition of large-gap clones because the ratio of the code lengths of Clone A and Clone B is $12/22 \simeq 0.55 < 0.7$.

Wang et al. proposed CCAligner [37], a large-gap clone detector. CCAligner detects clones using a code window (a code fragment composed of k consecutive lines in a code block). First, CCAligner transforms code blocks into code windows. Then, it identifies clone candidates as pairs of code blocks that share at least one code window with considering e edit distance. Finally, it verifies clone candidates based on their similarity, which is calculated as follows:

$$sim(c_1, c_2) = \frac{|W_{c_1} \cap W_{c_2}|}{min(|W_{c_1}|, |W_{c_2}|)}$$

where c_1 and c_2 are two code blocks, and W_{c_1} and W_{c_2} are the corresponding sets of code windows, respectively.

2.3 Large-variance clone

A large-variance clone is a clone generated by inserting or deleting many statements in various places in a copy-and-pasted code fragment. Figure 2 shows an example of large-variance clones. In this example, statements have been inserted into and deleted from various places in Clone A to create Clone B. Wu et al. pointed out that CCAligner targets the detection of large-gap clones, making it incapable of large-variance clone detection [40]. Wu et al. defined large-variance clones as clones whose code length ratio is less than 0.7. This means that large-gap clones are a special case of large-variance clones. They proposed LVMapper, a large-variance clone detector. Figure 3 shows that the relation among clone types and the clone types targeted by CCAligner, LVMapper, and NIL. CCAligner targets large-gap clones, whereas LVMapper and NIL target large-variance clones, which include large-gap clones.

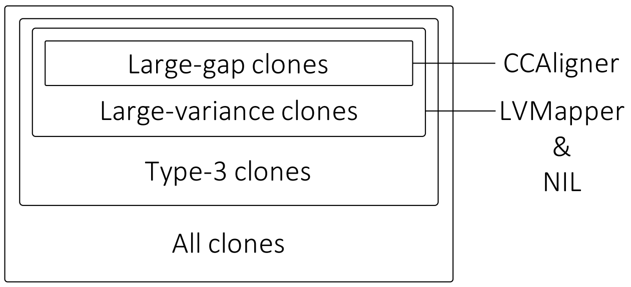

Figure 3: Relation among clone types and target clone types for several tools

LVMapper detects clones using code windows, just like CCAligner. Its clone detection has three phases, namely the locating, filtering, and verifying phases. In the locating phase, LVMapper identifies pairs of code blocks that share at least one code window as clone

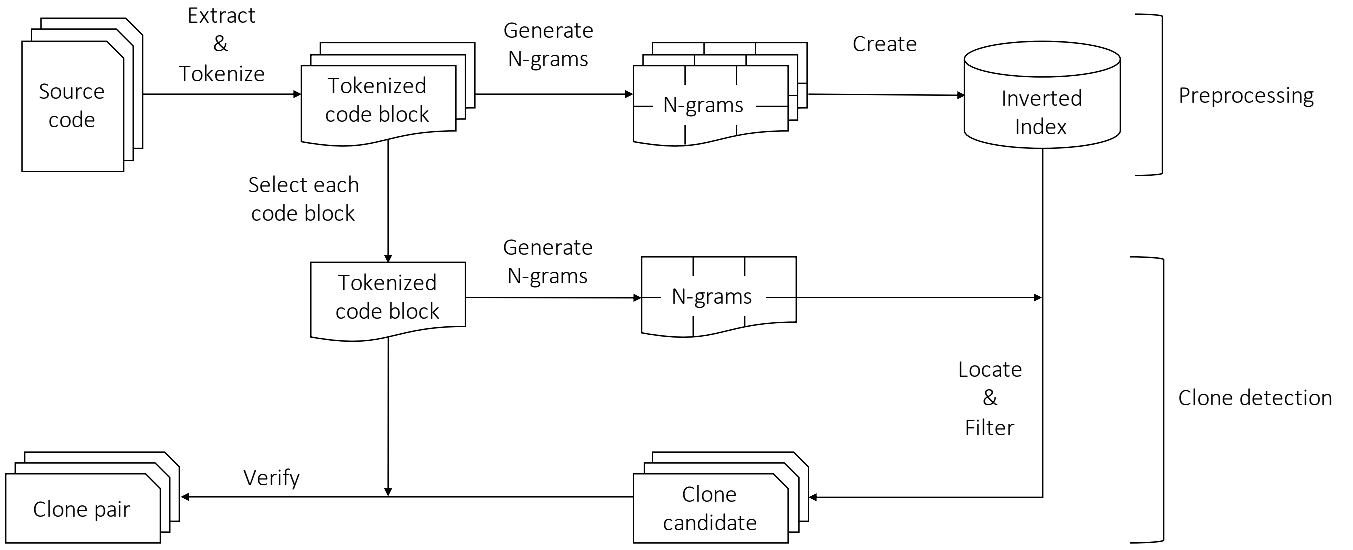

Figure 4: Overview of NIL

candidates. Then, in the filtering phase, it calculates the proportions of common code windows for each clone candidate and removes clone candidates whose proportions are lower than filtering threshold θ. Finally, in the verifying phase, it verifies each clone candidate based on similarity measured using a common subsequence of lines between each clone candidate's code block pair.

3 APPROACH

Figure 4 shows an overview of the proposed technique. The input is a set of source code files, and the output is the clone pairs in the source code. In the proposed technique, large-variance clones are detected based on the similarity between token sequences based on the LCS, taking advantage of the fact that the order of many tokens in a large-variance clone pair is preserved. In addition, large-variance clones share many consecutive tokens. Hence, to achieve scalable large-variance clone detection, the proposed technique reduces the number of code block comparisons using an N-gram representation of token sequences and an inverted index. The proposed technique transforms code blocks in source code into token sequences in the *Preprocessing* phase and detects clones by comparing the token sequences in the *Clone detection* phase. In this study, we implemented the proposed technique as a tool, called NIL. NIL is written in the Kotlin language and currently targets only Java source code. The following subsections describe the *Preprocessing* and *Clone detection* phases.

3.1 Preprocessing

In the *Preprocessing* phase, NIL extracts code blocks from the target source code and transforms them into token sequences. NIL does not perform lexical analysis but simply divide each code block's text based on symbols (e.g., "+", "−", or braces), white spaces, or newlines, as done by SourcererCC. For example, when the code block shown in figure 2(a) is transformed into the token sequence, `protected`, `String`, `getPrompt`, `InputRequest`, `request`, ...

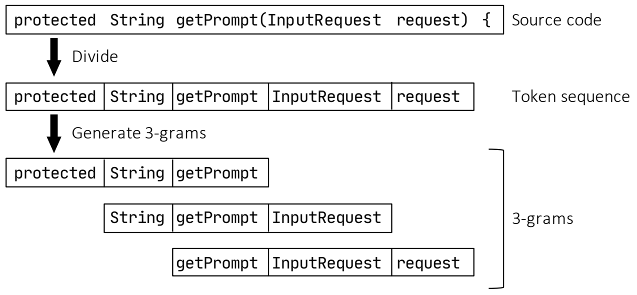
protected String getPrompt(InputRequest request) {
Source code
Divide
protected | String | getPrompt | InputRequest | request
Token sequence
Generate 3-grams
protected | String | getPrompt
String | getPrompt | InputRequest
getPrompt | InputRequest | request
3-grams

Figure 5: Example of generating 3-grams

With this transformation, lexical analyzers for other languages do not need to be implemented to extend NIL. The token sequence transformation is fast because lexical analysis is not necessary. In addition, NIL has a relatively low rate of false positives because it does not normalize identifiers, such as variable and function names. However, it may not detect clones whose identifiers have been changed (i.e., Type-2 clones). We discuss the impact of the lack of identifier normalization in Section 4.

Next, NIL generates N-grams from each token sequence. An N-gram is a chunk of consecutive N tokens. Figure 5 shows an example of generating 3-grams from the code block shown in figure 2(a). In this example, three 3-grams are generated from the five tokens on the first line in the code block. Even though large-variance clones include many modifications (statement insertions and deletions), many tokens other than the statements match consecutively (i.e., many N-grams match). Therefore, using N-grams is effective for scalable large-variance clone detection.

Then, NIL creates an inverted index from the generated N-grams. An inverted index is an information retrieval technology that allows the fast retrieval of documents that contains a word given as a query [15]. It is often used in clone detection techniques [7]. NIL

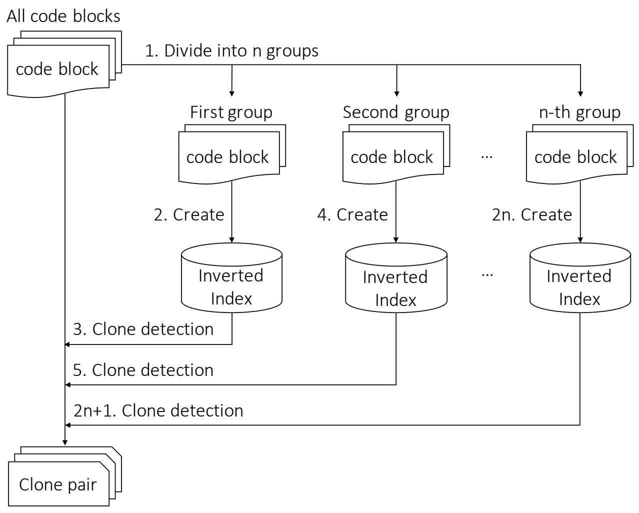

Figure 6: Concept of partial inverted indexes

uses a dictionary whose keys are the hash values of N-grams, and values are the code blocks containing the corresponding N-gram as an inverted index. All code blocks containing an N-gram can be quickly obtained by looking up the hash value of the N-gram in the inverted index. Therefore, a pair of code blocks that share an N-gram (i.e., the pair is possibly a large-variance clone pair) can be obtained quickly using the inverted index.

However, an inverted index consumes a lot of memory. Hence, creating an inverted index from all code blocks may lead to large memory consumption. To avoid this, we apply partial inverted indexes [34]. Code blocks are divided into several groups and an inverted index is created for each group (i.e., a partial inverted index) instead of creating a single inverted index for all code blocks. Figure 6 shows the concept of partial inverted indexes. First, code blocks extracted from source code are divided into *n* groups (Step 1), where *n* is set to a value such that the memory consumption of a partial inverted index is manageable. Next, an inverted index is created from one group of code blocks (Step 2). Based on the partial inverted index created in Step 2 and all code blocks, clone pairs between the code blocks in the group and all code blocks are detected in Step 3 (the clone detection process is described in the following Section 3.2). Steps 2 and 3 are performed for each group of code blocks.

3.2 Clone detection

After the *Preprocessing* phase, NIL performs *Clone detection* using the inverted index created in the *Preprocessing* phase and all code blocks. *Clone detection* is divided into three phases: location, filtration, and verification, as done by LVMapper [40]. First, NIL selects a code block from all code blocks prepared in the *Preprocessing* phase as the target code block. Then, in the location and filtration phases, NIL identifies the clone candidates of the target code block using an N-gram and the inverted index. Next, in the verification phase, NIL verifies that the target code block and the clone candidates are clone pairs by calculating the LCS. These phases are

Algorithm 1: Clone Detection

```
Input: $C$ is a list of tokenized code blocks $\{c_1, c_2, \ldots, c_n\}$,
    Inverted Index $I$ of $C$, $N$ for size of N-gram, $\theta$ for filtering
    threshold, $\delta$ for verifying threshold
Output: All clone pairs $CP$
1:  $CP \leftarrow \phi$;
2:  for all each $c_i$ in $C$ do
3:      // Location phase
4:      // $CC$ represents clone candidates
5:      $CC \leftarrow \phi$
6:      for $j = 1, 2, \ldots, (c_i.len - N + 1)$ do
7:          // $c_i[j]$ is $j$-th token in $c_i$'s token sequence
8:          $n\_gram = concat(c_i[j], c_i[j+1], \ldots, c_i[j+N-1])$;
9:          $key = hash(n\_gram)$;
10:         /* $get$ is a function that returns values to which a given
            key is mapped in a given dictionary */
11:         $CC = CC \cup get(I, key)$;
12:     end for
13:
14:     // Filtration phase
15:     for all each $cc_j$ in $CC$ do
16:         /* $common\_ngrams$ is a function
            that computes the number of common N-grams between
            two given code blocks */
17:         $cn = common\_ngrams(c_i, cc_j)$;
18:         $m = min(c_i.len, cc_j.len)$;
19:         $filtration\_sim = cn/(m - N + 1)$;
20:         if $filtration\_sim < \theta$ then
21:             $CC = CC \setminus \{cc_j\}$;
22:         end if
23:     end for
24:
25:     // Verification phase
26:     for all each $cc_j$ in $CC$ do
27:         /* $lcs$ is a function that computes the length of the LCS
            between token sequences of two given code blocks */
28:         $lcs\_len = lcs(c_i, cc_j)$;
29:         $verification\_sim = lcs\_len/min(c_i.len, cc_j.len)$;
30:         if $verification\_sim \geq \delta$ then
31:             $CP = CP \cup (c_i, cc_j)$;
32:         end if
33:     end for
34: end for
35: return $CP$;
```

performed for each code block to detect all clone pairs in the target source code. Algorithm 1 shows the *Clone detection* algorithm. The three phases of *Clone detection* are described in detail below.

3.2.1 Location phase. In the location phase, NIL collects the clone candidates of the target code block using the inverted index. Lines 3–11 in Algorithm 1 are the location phase.

First, NIL generates N-grams from the token sequence of the target code block. M−N+1 N-grams are generated from a token sequence with length M. Next, a hash value is calculated for each N-gram. This hash value is used as a query when looking up values in the inverted index. Finally, NIL applies the hash values to the inverted index and collects code blocks that contain the N-gram whose hash value is the same as the given hash value. The obtained code blocks are referred to as the clone candidates of the target code block.

3.2.2 Filtration phase. In the filtration phase, NIL removes code blocks that unlikely to be clones from the clone candidates collected in the location phase. Lines 14–23 in Algorithm 1 are the filtration phase. It is necessary to reduce the number of clone candidates for scalable and fast clone detection because NIL performs the LCS calculation, which is a time-consuming process, in the verification phase. NIL filters clone candidates based on a feature of large-variance clones.

As described in Section 3.1, the two code blocks of a large-variance clone pair share a certain number of N-grams. If two code blocks share few N-grams, the pair is unlikely to be a large-variance clone pair. Based on this feature, NIL calculates $filtration\_sim$, defined below, between the target code block and each clone candidate.

$$filtration\_sim(c_1, c_2) = \frac{common\_ngrams(c_1, c_2)}{min(ngrams(c_1), ngrams(c_2))}$$

$$common\_ngrams(c_1, c_2) = |ngrams(c_1) \cap ngrams(c_2)|$$

where c_1 and c_2 are two code blocks with lengths $|c_1|$ and $|c_2|$, respectively. $ngrams(c_1)$ and $ngrams(c_2)$ are the numbers of N-grams generated from code blocks c_1 and c_2, respectively. Because of the large number of statement insertions and deletions in large-variance clones, the two code blocks may have significantly different token sequence lengths. We use *min* in the denominator so that $filtration\_sim$ can be properly calculated even in such cases. NIL removes clone candidates whose $filtration\_sim$ is less than filtration threshold θ.

3.2.3 Verification phase. In the verification phase, NIL checks whether the target code block and each clone candidate are a true large-variance clone pair. Lines 25–33 in Algorithm 1 are the verification phase. As mentioned in Section 3.2.2, one of the features of large-variance clones is that the common subsequence between token sequences of large-variance clones is long even if there are a large number of insertions and deletions. Therefore, NIL calculates the LCS between the target code block and each clone candidate and measures the similarity of the pair based on the length of the LCS. The similarity function $verification\_sim(c_1, c_2)$ is expressed as following

$$verification\_sim(c_1, c_2) = \frac{lcs(c_1, c_2)}{min(|c_1|, |c_2|)}$$

where c_1 and c_2 are token sequences with lengths $|c_1|$ and $|c_2|$, respectively, and $lcs(c_1, c_2)$ is the length of the LCS between c_1 and c_2. We use *min* as the denominator of the similarity function to detect large-variance clones, as done in the studies on CCAligner [37] and LVMapper [40] even if the lengths of the token sequences differ greatly.

Other clone detectors [4, 40] also use the LCS to measure similarity. However, they calculate line-based LCS, whereas NIL calculates token-based LCS. In general, the token sequence of a code block is longer than its line sequence. The time complexity of a method for LCS calculation based on dynamic programming is $O(|A| \times |B|)$, indicating a very long computation time for a large input size. To reduce time complexity, NIL uses the Hunt-Szymanski algorithm [8]. With this algorithm, NIL can calculate the LCS in $O(r \log |A| + |B| \log |B|)$, where A and B are token sequences ($|A| \le |B|$) and r is the number of pairs of common tokens between A and B.

4 EVALUATION

We evaluated NIL in terms of

- large-variance clone detection accuracy,
- general clone detection accuracy, and
- scalability.

In the following subsections, we first optimize the N-gram size based on a balance between recall and execution time. Next, we evaluate large-variance clone detection accuracy in terms of precision and recall. Then, we evaluate general clone detection using two commonly used benchmarks. Finally, we evaluate scalability by measuring execution time for various input sizes. Additionally, we compare the above results to those for four state-of-the-art tools [4, 29, 37, 40]. Table 1 shows these clone detectors and their settings. These settings were taken from the prior studies [29, 37, 40]. Note that the threshold δ for verification of LVMapper is variable and that δ takes the following values depending on the number of the lines of clone l.

$$\delta = \begin{cases} 0.7 & \text{if } 6 < l \le 10, \\ 1 - 0.03 \times l & \text{if } 10 < l \le 20, \\ 0.4 & \text{if } 20 < l \end{cases}$$

4.1 Summary

First, we summarize the results of this evaluation. We found that NIL has high precision of 87% and high recall of 100% in large-variance clone detection. These values are the highest among the tested large-variance clone detectors [37, 40]. In general Type-1, Type-2, and Type-3 clone detection, NIL's accuracy is equivalent to that of the existing clone detectors, including large-variance clone detectors, and its precision is higher than that of large-variance clone detectors. Moreover, we confirmed that NIL is the fastest at detecting clones in large codebases (1–250 MLOC) among the tested clone detectors.

Table 1: Settings for various clone detectors

Tool	Settings
LVMapper	Min length 6 lines, window size $k = 3$, filtering threshold $\theta = 0.1$, verification threshold δ is variable.
CCAligner	Min length 6 lines, window size $q = 6$, edit distance $e = 1$, min 60% similarity.
SourcererCC	Min length 6 lines, min 70% similarity.
NiCad	Min length 6 lines, max length 20,000 lines, blind renaming, identifier abstraction, min 70% similarity.

4.2 Parameter setting

NIL requires three parameters, namely N for N-grams, filtration threshold θ, and verification threshold δ. We set δ to 0.7, which is often used in clone detectors [4, 29]. We set θ to 0.1, as done for LVMapper. N must be carefully selected because it has a large impact on performance (e.g., execution time and clone detection accuracy). If N is set to a too small value, the recall of clone detection will increase because more code blocks will share the same N-grams. However, because more code blocks are identified as clone candidates in the location phase, the number of comparison targets in the filtration and verification phases increases, resulting in a longer execution time. Therefore, to optimize the N, we executed NIL with $N = 1$–9 and measured the execution time and clone detection recall for each N value. We used BigCloneEval [33] to measure recall. BigCloneEval automatically measures the recall of clone detectors using BigCloneBench [32].

Table 2 shows the results for each N value. For $N < 5$, an increase in N significantly reduces the execution time without lowering recall by more than one point compared to when $N = 1$. For $N = 5$ and 6, execution time decreases but recall also decreases. For $N > 6$, execution time does not significantly decrease. Therefore, considering the balance between execution time and recall, $N = 5$ is the optimal value.

4.3 Large-variance clone detection

We evaluated the large-variance clone detection accuracy of NIL in terms of precision and recall and compared the results to those for existing large-variance and large-gap clone detectors, namely CCAligner and LVMapper.

4.3.1 Precision. Precision is the ratio of correct clones detected to all clones detected. A clone detector with higher precision provides more accurate results. In general, precision is measured via a manual validation of the clones detected by the target tool. In this study, we used Ant and Maven, used in the prior studies on CCAligner [37] and LVMapper [40] to measure precision. JDK1.2.2 and OpenNLP, also used in the above studies, were not used here because we could not find the source code for JDK1.2.2, which has been end-of-lifed and we considered OpenNLP to be unsuitable for manual validation because it is a machine learning library whose source code is difficult to read (e.g., it includes repetitive array manipulation code). We used the following procedure to measure the precision of each tool:

(1) we input the target source code into each tool,
(2) we randomly selected 100 large-variance clone pairs with more than ten lines for each tool and target system, and
(3) we manually confirmed whether the clone pairs were correct.

To remove bias in the manual validation, the detected large-variance clone pairs of a given tool were validated without knowledge of the tool used for detection. Table 3 shows the number of files and total LOC for Ant and Maven.

Table 4 shows the number of large-variance clone pairs detected by each tool and the precision for each tool[2]. The number of large-variance clones detected by NIL detected was almost the same as that of LVMapper and more than that of CCAligner. The manual validation results indicate that NIL had high average precision of 87% whereas LVMapper and CCAligner had low average precision values of about 60% and 40%, respectively. We considered this difference in precision to be due to LVMapper and CCAligner normalizing the identifiers in code blocks. After checking the large-variance clones detected by LVMapper and CCAligner, we found that code blocks with consecutive assignment statements, such as constructors, and consecutive `if`-statements were incorrectly detected as large-variance clones. In contrast, NIL detected large-variance clones more precisely because it does not perform identifier normalization.

Table 3: Target systems

System	# Files	LOC
Ant 1.10.1	895	109,073
Maven 3.5.0	698	60,471

Table 4: Large-variance clone detection results

Tool	System	# Large-variance clones	Precision (%)
NIL	Ant	354	86.0
	Maven	398	88.0
LVMapper	Ant	355	64.0
	Maven	389	60.0
CCAligner	Ant	184	43.0
	Maven	284	40.0

[2]Manual validation descriptions are available at https://zenodo.org/record/4490845

Table 2: Recall and execution time results for each N value

N	1	2	3	4	5	6	7	8	9
Type-2	97.5	97.5	97.5	97.5	96.6	96.3	96.1	95.8	93.3
Very Strongly Type-3	93.5	93.5	93.5	93.5	93.5	93.1	92.9	92.7	92.5
Strongly Type-3	68.4	68.4	68.4	68.3	67.1	65.3	64.6	63.7	62.0
Moderately Type-3	11.2	11.2	11.2	11.1	10.6	9.9	9.3	8.7	7.8
Execution time	> 24h	2h 13m 52s	21m 42s	5m 56s	2m 58s	1m 13s	1m 4s	1m 2s	57s

In addition, NIL had higher precision than that of LVMapper and detected a similar number of large-variance clone pairs, indicating that it can detect many large-variance clones that LVMapper cannot. After checking large-variance clones that NIL detected but LVMapper did not, we found that the large-variance clone pairs share a small number of consecutive lines. LVMapper regards three consecutive lines in code blocks as code windows (see Section 2) and identifies clone candidates based on these code widows. Therefore, if a pair of code blocks shares a little or no code window, LVMapper cannot detect the pair as a large-variance clone pair. In contrast, NIL can detect such large-variance clones because it uses an N-gram representation of token sequences.

4.3.2 Recall. Recall is the ratio of clones detected by a tool to the total number of true clones in the target codebase. A clone detector with higher recall can detect true clones more exhaustively. To evaluate recall, all true clone pairs in the codebase are required. However, it is not realistic to manually check all code block pairs in the codebase to determine the total number of true clone pairs.

Therefore, we generated large-variance clone pairs automatically using mutation techniques to evaluate recall. Mutation techniques are frequently used for clone detector recall evaluation [31]. The studies on CCAligner [37] and LVMapper [39] used them to evaluate large-variance clone detection recall. In this study, we reproduced large-variance clones by randomly inserting various numbers of statements into original code blocks (i.e., the large-variance clones are mutants of the original code blocks). We targeted JDK6 and Apache Commons because they were used in the study on CCAligner [37]. We randomly selected 100 functions with 15–50 lines in these systems as the original code blocks. A minimum length of 15 lines is often used for recall evaluation using mutation techniques [29, 31]. We used a maximum length of 50 lines because if the number of lines of the original code blocks is too large, even if a large number of statements are inserted into the code blocks, the generated clones will not be large-variance clones. For example, if 20 lines of statements are inserted into an original code block, if the number of lines of a code block is 100, the ratio of lines of the clone pair is $100/120 > 0.7$, and thus the clone pair does not satisfy the large-variance clone definition. To reproduce large-variance clone pairs, 1–20 one-line statements were inserted into each code block at random locations. 20 clone pairs were generated per original code block, for a total of 2,000 clone pairs, including large-variance clone pairs[3].

Figure 7 shows the recall of NIL, LVMapper, and CCAligner for clone pairs generated by inserting various numbers of statements. NIL detected all generated clone pairs. This is because even though many statements are inserted into a code block, the order of many tokens between the large-variance clone pair is preserved, and thus the clone pair shares a certain number of N-grams and has a long common subsequence. In summary, using N-gram-based clone candidate identification and token-LCS-based clone validation is effective in large-variance clone detection.

On the other hand, as shown in Figure 7, the recall of LVMapper and CCAligner decreased with increasing number of inserted lines. LVMapper can also detect clones in which a large number

[3]The generated large-variance clone pairs are available at https://zenodo.org/record/4491016

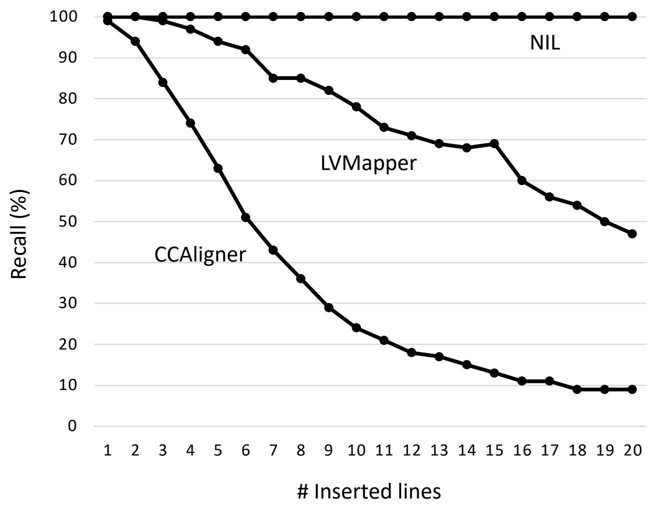

Figure 7: Recall results for various numbers of inserted lines

of statements are inserted because it verifies clones based on line-based LCS. However, our evaluation results show that the recall of LVMapper decreased with increasing number of inserted lines. This is because LVMapper failed to identify many large-variance clones in its locating phase. LVMapper identifies a pair of code blocks that share several code windows (see Section 2) as a clone pair. Therefore, with increasing number of inserted lines, the number of shared code windows decreases, and thus LVMapper failed to identify a pair of code blocks as a large-variance clone pair. CCAligner uses e-mismatch code windows to identify clone candidates, and this affects its recall. In addition, CCAligner uses code windows in verification and thus fails to detect large-variance clones.

4.4 General clone detection

We evaluated the general Type-1, Type-2, and Type-3 clone detection accuracy of NIL using two benchmarks, namely Mutation Framework [31] and BigCloneEval [33]. In addition, we compared the results of NIL to those of existing state-of-the-art tools, namely CCAligner, LVMapper, SourcererCC, and NiCad.

4.4.1 Mutation Framework. Mutation Framework automatically generates clone pairs based on mutation techniques. We executed Mutation Framework with all the default settings and input the generated clones[4] into each clone detector. Table 5 shows the results of recall for each tool measured by Mutation Framework. NIL detected all clone pairs generated by Mutation Framework.

Table 5: Recall results for Mutation Framework

Tool	NIL	LVMapper	CCAligner	SourcererCC	NiCad
Type-1	100.0	100.0	100.0	100.0	100.0
Type-2	100.0	100.0	100.0	100.0	100.0
Type-3	100.0	99.9	99.9	100.0	100.0

[4]The generated clone pairs are available at https://zenodo.org/record/4491052

4.4.2 BigCloneEval. BigCloneEval [33] automatically measures the recall of clone detectors using BigCloneBench [32]. We also measured precision, as done in the prior studies [29, 37, 40]. For each tool, we randomly selected 400 of the detected clone pairs from BigCloneBench and validated them manually. The clones were shuffled, and the validation was conducted without knowledge of the clone source.

Table 6 shows the results of recall[5] and precision[6] for each tool on BigCloneBench. As shown, NIL has high recall of 96.6% for Type-2 clone detection even though it does not normalize identifiers in the *Preprocessing* phase. NIL had the second highest recall of Moderately Type-3 clones, which contain large-variance clones, behind only LVMapper. This is because the verifying threshold of LVMapper is variable, and in some cases it can be low as 0.4.

The precision (see bottom of Table 6) of both LVMapper and CCAligner, which are large-variance and large-gap clone detectors, was low. In contrast, that of NIL was very high (94.0%). Even though SourcererCC and NiCad also had high precision, they had poor large-variance clone detection performance. Therefore, compared to the existing tools, NIL has equivalent general clone detection accuracy and higher precision than that of the existing large-variance clone detectors.

4.5 Scalability

We evaluated the scalability of NIL using codebases with various sizes and compared the execution time of NIL to those of the existing tools. We used IJaDataset [24], a large inter-project Java dataset, as done in the prior studies [29, 37, 40]. We created datasets with 1, 10, 100, and 250 MLOC[7]. We used CLOC [26] to measure the LOC of the datasets. A computer with a quad-core CPU and 12 GB of memory was used for the evaluation, as done in the prior studies [29, 37].

Table 7 shows the execution times for each tool for various input sizes. As shown, the execution time of NIL is the shortest for all input sizes. Even though both LVMapper and SourcererCC detected clones from the 250-MLOC codebases, their execution times are longer than three days, indicating poor scalability. In addition, CCAligner and NiCad were not able to complete detecting clones from the 100- and 10-MLOC codebases, respectively, indicating their limited scalability. Therefore, NIL has the highest scalability.

Moreover, we examined how effective the location and filtration phases are for NIL's scalability. Figure 8 shows growth in the number of clone candidates with the increased number of code blocks. Note that this figure is a logarithmic graph. "Naive" is comparing

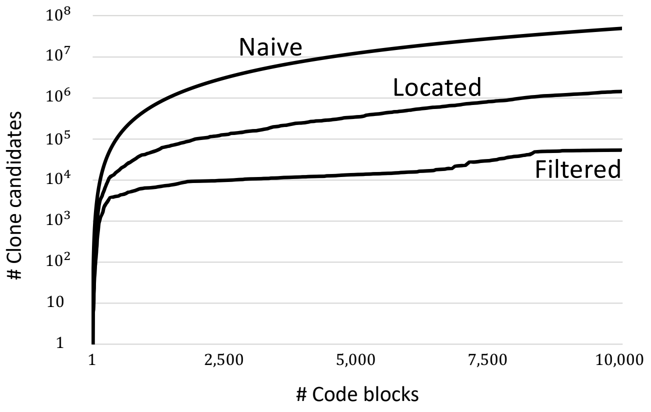

Figure 8: Growth in the number of clone candidates with the increased number of code blocks

[5]Note that in our experiments, BigCloneEval reported different recall of the existing clone detectors from the prior studies. Type-1 recall of several clone detectors was 99.9% because BigCloneBench contains faulty clone pairs [9].

[6]Manual validation descriptions are available at https://zenodo.org/record/4493069

[7]These datasets and an executable file of NIL are available at https://zenodo.org/record/4491208.

Table 6: Recall and precision results for BigCloneBench

Tool	NIL	LVMapper	CCAligner	SourcererCC	NiCad
Type-1 Recall	99.9	99.9	99.8	93.8	99.9
Type-2 Recall	96.6	99.2	98.9	96.6	99.2
Very Strongly Type-3 Recall	93.5	98.1	97.4	68.2	98.4
Strongly Type-3 Recall	67.1	81.8	69.0	59.0	69.7
Moderately Type-3 Recall	10.6	19.1	10.0	4.8	0.5
Precision	94.0	58.5	33.7	99.2	80.2

Table 7: Scalability results

Tool	NIL	LVMapper	CCAligner	SourcererCC	NiCad
1 M	10s	29s	52s	3m 1s	1m 48s
10 M	1m 38s	13m 38s	26m 3s	27m 37s	—
100 M	1h 38m 29s	17h 23m 39s	—	19h 38m 5s	—
250 M	7h 40m 7s	3d 13h 47m 39s	—	5d 6h 55m 1s	—

all pairs of code blocks for clone detection[8]. As shown, both the location and filtration phases reduced the number of clone candidates drastically. For example, when there were 10,000 code blocks, the number of clone candidates was reduced from 49,995,000 in Naive to 1,449,634 (about one-thirty) in the location phase. In the filtration phase, their number was further reduced to 54,000 (about one thirty-fifth) from the location phase. Therefore, the two phases reduced their number to about one-thousandth from Naive and very effective for scalable clone detection.

5 THREATS TO VALIDITY

To measure precision for each tool, we manually validated clones each tool detected, as done in the prior studies [29, 37, 40]. Because the clone detector names were not disclosed during the manual validation, there was no bias in the evaluation. However, because the criteria for whether a pair of code fragments is a clone pair can vary, manual validation by other researchers may yield different values. To ensure the validity of this study, the clones used in the manual validation are made public so that a third party can conduct replication studies.

We used the widely used benchmarks, BigCloneEval [33] and Mutation Framework [31], to evaluate the recall of clone detectors. Different results may be obtained using other benchmarks [1, 41].

It is known that the accuracy and execution time of a clone detector is greatly influenced by its settings [38]. In this study, we optimized the N value for N-grams. However, the filtration threshold θ and the verification threshold δ were set based on values used by other clone detectors. The results can be improved by optimizing these values for NIL.

6 RELATED WORK

6.1 Complicated Type-3 clone detection

In addition to large-variance clones, because detecting complicated Type-3 clones is difficult, several techniques specialized for detecting them have been proposed.

Program dependence graph (in short, PDG) [5] is frequently used for complicated Type-3 clone detection. Krinke was the first to use PDGs for clone detection [17]. His technique detected isomorphic parts of PDGs as clones. He reported that PDG-based clone detection was good at recall and precision. Zou et al. pointed out that PDG-based clone detection techniques have still been quite time-consuming and missed many clones due to their exact graph matching using subgraph isomorphism. They proposed CCGraph [42], using an approximate graph matching algorithm based on the reforming Weisfeiler-Lehman graph kernel [30].

Intermediate representation (in short, IR) is also used for complicated Type-3 clone detection. Several syntactical differences (e.g., `for`-loop and `while`-loop) in source code are transformed into the same or similar instructions in IRs of the source code. Caldeira et al. proposed a clone detection technique using IRs [2]. They devised a C-like IR based on LLVM and ran NiCad [4] on it. Their experimental results suggested that IRs were beneficial for improving clone detection and had a large impact on complicated Type-3 clones.

[8]The curve can be represented using $y = x(x-1)/2$ quadratic function where x is the number of code blocks, and y is the number of clone candidates.

Machine learning is also useful for complicated Type-3 clone detection. Saini et al. proposed a clone detector, Oreo, for Weakly Type-3 clones [28]. It combines machine learning, information retrieval techniques, and software metrics to detect clones.

However, those three techniques based on PDG, IR, or machine learning do not always detect large-variance clones. For example, a prior study showed that Oreo has higher precision but lower recall in large-variance clone detection than those of LVMapper [40]. Moreover, these techniques require a long execution time and thus are limited in scalability. PDG-based clone detection requires a long time to construct PDGs and perform subgraph isomorphism. IR-based clone detection requires a long time to transform source code into IRs. Machine learning-based clone detection is necessary to complete training before clone detection.

6.2 Scalable clone detection

Kamiya et al. proposed CCFinder [13] and its successor, CCFinderX [25]. CCFinder transforms the target source code into a token sequence, normalizes identifiers, and then uses a suffix tree algorithm to detect matching token sequences as clones. As shown in prior studies [18, 29], CCFinderX has high scalability and can detect clones even for a 100-MLOC codebase.

Ishihara et al. proposed a scalable method-level clone detection technique [10]. The technique hashes each normalized method and detects methods whose hash values are the same as clones. They detected clones in a large codebase (360 MLOC) in 3.5 hours.

Hummel et al. proposed ConQat [7], an index-based clone detector. ConQat creates clone indexes by hashing consecutive six lines of source code and detects the clone indexes whose hash values are the same as clones. ConQat is capable of distributed processing in clone detection, so that it can be applied for ultra-large codebase (2.9 GLOC) using cluster computing.

However, those scalable clone detectors cannot detect gapped clones due to their algorithms. Though there are several tools for scalable near-miss clone detection [18, 23, 29, 34], they still cannot detect complicated Type-3 clones, including large-variance clones. In this study, we proposed NIL, which achieves both large-variance clone detection and scalability.

7 CONCLUSION

In this study, we proposed a clone detection technique for the scalable detection of large-variance clones from a large codebase and described its implementation, called NIL. NIL uses N-grams, an inverted index, and the LCS to detect large-variance clones. Our experimental results show that NIL has higher precision and recall in large-variance clone detection than those of existing large-variance and large-gap clone detectors. In addition, the general clone detection accuracy of NIL is equivalent to that of existing state-of-the-art tools. Moreover, NIL can detect clones from large codebases more quickly than do existing clone detectors.

As future works, we consider doing research on software engineering applications such as code recommendation and completion, refactoring, and bug propagation for large-variance clones using NIL.

ACKNOWLEDGMENTS

This work was supported by MEXT/JSPS KAKENHI 20H04166.

REFERENCES

[1] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591.

[2] Pedro M Caldeira, Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa, and Takahisa Shimada. 2020. Improving Syntactical Clone Detection Methods through the Use of an Intermediate Representation. In *Proceedings of 2020 IEEE 14th International Workshop on Software Clones.* 8–14.

[3] Debarshi Chatterji, Jeffrey C Carver, Beverly Massengil, Jason Oslin, and Nicholas A Kraft. 2011. Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study. In *Proceedings of 2011 International Symposium on Empirical Software Engineering and Measurement.* 20–29.

[4] James R Cordy and Chanchal K Roy. 2011. The NiCad Clone Detector. In *Proceedings of 2011 IEEE 19th International Conference on Program Comprehension.* 219–220.

[5] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.

[6] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2004. Refactoring Support Based on Code Clone Analysis. In *Proceedings of International Conference on Product Focused Software Process Improvement.* 220–233.

[7] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. 2010. Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of 2010 IEEE International Conference on Software Maintenance.* 1–9.

[8] James W Hunt and Thomas G Szymanski. 1977. A fast algorithm for computing longest common subsequences. *Communications of the ACM* 20, 5 (1977), 350–353.

[9] Katsuro Inoue, Yuya Miyamoto, Daniel M German, and Takashi Ishio. 2020. Code Clone Matching: A Practical and Effective Approach to Find Code Snippets. *arXiv preprint arXiv:2003.05615* (2020).

[10] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2012. Inter-Project Functional Clone Detection Toward Building Libraries - An Empirical Study on 13,000 Projects. In *Proceedings of 2012 19th Working Conference on Reverse Engineering.* 387–391.

[11] Takashi Ishio, Yusuke Sakaguchi, Kaoru Ito, and Katsuro Inoue. 2017. Source File Set Search for Clone-and-Own Reuse Analysis. In *Proceedings of 2017 IEEE/ACM 14th International Conference on Mining Software Repositories.* 257–268.

[12] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* 55–64.

[13] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[14] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering.* 187–196.

[15] Donald Ervin Knuth. 1997. *The art of computer programming.* Pearson Education.

[16] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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

[18] Guanhua Li, Yijian Wu, Chanchal K Roy, Jun Sun, Xin Peng, Nanjie Zhan, Bin Hu, and Jingyi Ma. 2020. SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration. In *Proceedings of 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering.* 272–283.

[19] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[20] Hou Min and Zhang Li Ping. 2019. Survey on Software Clone Detection Research. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences.* 9–16.

[21] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. 2020. A survey on clone refactoring and tracking. *Journal of Systems and Software* 159 (2020), 110429.

[22] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects. In *Proceedings of the 15th International Conference on Mining Software Repositories.* 153–164.

[23] Manziba Akanda Nishi and Kostadin Damevski. 2018. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software* 137 (2018), 130–142.

[24] [Online]. 2021. Ambient Software Evolution Group: IJaDataset 2.0. http://secold.org/projects/seclone.

[25] [Online]. 2021. CCFinderX. http://www.ccfinder.net/.

[26] [Online]. 2021. CLOC: Count lines of code. http://cloc.sourceforge.net/.

[27] Chanchal K Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[28] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. 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.* 354–365.

[29] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering.* 1157–1168.

[30] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research* 12, 9 (2011).

[31] Jeffrey Svajlenko and Chanchal Roy. 2019. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering* (2019).

[32] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating Clone Detection Tools with BigCloneBench. In *Proceedings of 2015 IEEE International Conference on Software Maintenance and Evolution.* 131–140.

[33] Jeffrey Svajlenko and Chanchal K Roy. 2016. BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. In *Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution.* 596–600.

[34] Jeffrey Svajlenko and Chanchal K Roy. 2017. CloneWorks: a fast and flexible large-scale near-miss clone detection tool. In *Proceedings of the 39th International Conference on Software Engineering Companion.* 177–179.

[35] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone Refactoring with Lambda Expressions. In *Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering.* 60–70.

[36] Andrew Walker, Tomas Cerny, and Eungee Song. 2020. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. *ACM SIGAPP Applied Computing Review* 19, 4 (2020), 28–39.

[37] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering.* 1066–1077.

[38] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 455–465.

[39] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K. Roy. 2019. LVMapper: A Large-variance Clone Detector Using Sequencing Alignment Approach. arXiv:1909.04238 [cs.SE]

[40] Ming Wu, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu, and Chanchal K Roy. 2020. LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach. *IEEE Access* 8 (2020), 27986–27997.

[41] Yusuke Yuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2016. Generating clone references with less human subjectivity. In *Proceedings of 2016 IEEE 24th International Conference on Program Comprehension.* 1–4.

[42] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of 2020 35th IEEE/ACM International Conference on Automated Software Engineering.* 931–942.