Code Clone Detection using A Coarse and Fine-grained Hybrid Approach

Abdullah Sheneamer  
Department of Computer Science  
University of Colorado at Colorado Springs  
Colorado Springs, CO USA  
asheneam@uccs.edu

Jugal Kalita  
Department of Computer Science  
University of Colorado at Colorado Springs  
Colorado Springs, CO USA  
jkalita@uccs.edu

Abstract—If two fragments of source code are identical to each other, they are called code clones. Code clones introduce difficulties in software maintenance and cause bug propagation. Coarse-grained clone detectors have higher precision than fine-grained, but fine-grained detectors have higher recall than coarse-grained. In this paper, we present a hybrid clone detection technique that first uses a coarse-grained technique to analyze clones effectively to improve precision. Subsequently, we use a fine-grained detector to obtain additional information about the clones and to improve recall. Our method detects Type-1 and Type-2 clones using hash values for blocks, and detects gapped code clones (Type-3) using block detection followed by subsequent comparison among blocks using Levenshtein distance and Cosine measure with varying thresholds.

Keywords—Software Clone; Clone Detection; Software maintenance; Software evaluation.

I. INTRODUCTION

When a programmer copies and pastes a fragment of code, possibly with minor or even extensive edits, it is called code cloning. Code clones introduce difficulties in software maintenance and lead to bug propagation.

A coarse-grained approach detects clones of methods, statement blocks or classes. In contrast, a fine-grained approach detects clones of sequences of tokens, lines or statements. Our objective is to combine these two approaches. We perform a two-stage analysis which involves coarse detection, followed by fine-grained detection. We use coarse-grained detection to get an overview of clones in terms of blocks and fine-grained detection for detailed analysis. A coarse-grained analysis is used to detect Type-1 and Type-2 clones and the fine-grained analysis is used to detect Type-3 clones.

A coarse-grained technique has high precision since it detects fewer candidate clones than fine-grained. A fine-grained technique has high recall since it detects more reference clones than a coarse-grained technique. The reason we use a fine-grained technique as the second stage is because the first stage, the coarse-grained approach, detects only a few clones. We combine the two techniques to improve both recall and precision for a dataset. Existing text-based and token-based detection approaches produce many false positives. On the other hand, existing AST-based and PDG-based approaches require much time for transforming the source code into ASTs and PDGs and compare them [6].

We implement the proposed method and evaluate it by using Murakami’s benchmark dataset [6]. Murakami’s references represent code clones with information including where gaps of code clones start and where they end. In contrast, Bellon’s benchmark dataset [7] does not have information about where gaps are. The contributions of this paper are following.

• We use normalized blocks, followed by grouping, and hashing to detect Type-1 and Type-2 clones.
• We use two similarity measures to detect Type-3 clones. We tailor the Levenshtein distance algorithm to code clone detection. Levenshtien distance is a string metric for measuring the distance between two sequences. The tailored Levenshtien distance algorithm can measure distance between lines of code. We also use cosine similarity, tailored to measure angular distance between lines, represented as vectors.
• We demonstrate that our proposed method has higher precision and F-measure than existing methods.

The rest of the paper is organized as follows. Section 2 discusses background material. Related work on the topic of code clone detection is introduced in Section 3. In Section 4, the proposed method is discussed in detail. The similarity measures are described in Section 5. Experiment design is discussed in Section 6. We discuss the experiments we perform in Section 7. Discussions on our approach are covered in Section 8. Finally, the paper is concluded in Section 9.

II. BACKGROUND

A. Basic Definition

Here, we provide definitions which we use throughout our paper.

Definition 1: Coe Fragment. A code fragment (CF) is a part of the source code needed to run a program. It can contain functions, begin-end blocks or a sequence of statement.

Definition 2: Clone Pair. If a code fragment CF1 is similar to another code fragment CF2 syntactically or semantically, one is called a clone of the other.

Definition 3: Coarse-grained Approach. A clone detection approach that detects blocks, methods, statement blocks or classes as potential clones is called a coarse-grained approach.
Definition 4: Fine-grained Approach. A clone detection approach that detects sequences of tokens, lines or statements as potential clones is called a fine-grained approach.

B. Types of Clones

There are four types of clone relations between two code fragments based on the nature of similarity in their text [5,6].

Type-1 (Exact clones): Two code fragments are the exact copies of each other except whitespaces, blanks and comments.

Type-2 (Renamed): Two code fragments are similar except for names of variables, types, literals and functions.

Type-3 (Gapped clones): Two copied code fragments are similar, but with modifications such as added or removed statements, and the use of different identifiers, literals, types, whitespaces, layouts and comments.

Type-4 (Semantic clones): Two code fragments are semantically similar, without being syntactically similar.

III. RELATED WORK

Kamiya et al. [1] develop a suffix tree-matching algorithm called CCFinder. CCFinder applies several metrics to detect interesting clones. CCFinder performs a more suitable transformation than a line-by-line method. It produces high recall whereas its precision is lower than that of some other techniques [13]. CCFinder uses only a fine-grained approach, which has high recall but tends to have low precision. Our method uses both approaches to improve recall and precision.

Roy and James [11] implement a clone detection technique called Accurate Detection of Near Miss-Intentional Clones (NICAD). The NICAD tool uses two clone detection techniques, text-based and abstract syntax tree-based, to detect Type-1, Type-2 and Type-3 cloned code. NICAD detects exact and near-miss block code clones. It is difficult using only coarse-grained approach to detect Type-3. Therefore, our approach also uses a fine-grained approach as a second stage to detect Type-3.

Jiang et al. [12] use a tree-based technique and compute characteristic vectors to capture structural information about ASTs in Euclidean space. The Jiang et al. tool is called Deckard. Deckard detects re-ordered statements and non-contiguous clones. Different from this work, our technique improves existing text and AST based clone detectors to detect Type-3 clones.

Murakami et al. [6] propose a method that detects gapped clones using the Smith-Waterman algorithm [15] which finds alignments between two sequences with gaps. Our method differs from Murkami’s method; we use a hybrid approach instead of only a fine-grained approach.

Hotta et al. [5] detect, compare and evaluate coarse-grained and fine-grained methods. They develop a coarse-grained detector that detects block-level clones. Their detection approach has good accuracy and achieves high precision, but compared to a fine-grained method, it does not have high recall. It also does not tackle gapped code clones, with low recall reflecting this. The approach detects Type-1 and Type-2 clones but cannot detect Type-3. Our approach is similar to Hotta et al.’s approach in the first two steps, but we detect not only Type-1 and Type-2 clones but also Type-3 clones.

Hotta et al.’s approach is coarse-grained, whereas Murakami et al.’s approach [6] is fine-grained. A coarse-grained detector is faster with higher precision and is more scalable than a fine-grained detector, but a coarse-grained detector reports fewer clones than a fine-grained detector. Our approach is a hybrid approach which combines both coarse-grained and fine-grained approaches. Therefore, it has higher precision and F-measure than Murakami et al. [6]. In addition, it can detect more clones than Hotta’s approach [5].

IV. THE PROPOSED METHOD

We hash normalized blocks and compare them to detect Type-1 and Type-2 clones. We use two similarity measures to detect Type-3 clones. Details are given later in this section. The proposed method consists of the following steps.

Step 1. Lexical analysis and normalization.
Step 2. Detecting blocks and extracting sub-blocks.
Step 3. Grouping and hashing normalized blocks.
Step 4. Detecting similar blocks using Levenshtein Distance/Cosine similarity.
Step 5. Identifying gapped lines.
Step 6. Mapping similar blocks to the original source code.

Algorithm 1: Comparison of Two Blocks using LevDist

A. Lexical Analysis and Normalization

The first step is to transform and normalize all source files into special token sequences to detect not only identical clones but also similar ones. This also helps in dealing with varying numbers of whitespaces occurring together. Figure 1(a) gives the original files and 1(b) gives the two program fragments after lexical analysis and normalization. Identifiers have been replaced by the $ sign.
Figure 1. The Proposed Method. Each step is illustrated as we analyze the code in two files for the existence of clones.
B. Detecting Blocks and Extracting Sub-Block

This step needs not only lexical analysis but also syntactic analysis to detect every block from the given source files. All blocks, including classes, methods and block statements, are extracted using the Java Development Tool (JDT). Figure 1(c) shows the detected blocks for the two files.

C. Grouping and Hashing Normalized Blocks

After identifying all normalized blocks, we group them into similar blocks such as class blocks, method blocks, loop statements, branch statements and assignment statements. This helps detect similar clones later using Levenshtein distance or cosine similarity. These two similarity measures are discussed in Section 5. In Figure 1(d), we see the blocks detected in File 1 on top and blocks from File 2 at bottom in each group.

Next, this step calculates a hash value of the text of a block. We use HashCode() in Java as the hash function, which is a simply number; a 32-bit signed int. This step can find both of Type-1 and Type-2 clones by looking for two blocks or statements that have the same hash value. This happens if their text representations after normalization are equal to each other. For example, in File 1, a method block between lines 887 to line 888 has been detected as a Type-1 or Type-2 clone of a method block between lines 666 to line 667 in File 2.

D. Detecting Similar Blocks Using Levenshtein Distance/Cosine Similarity

Normalized blocks, which are similar, are detected using the Levenshtein distance algorithm. Levenshtein distance measures the distance between two blocks, which is the minimal number of insertions, deletions, and substitutions that will transform one block into the another. We also use cosine similarity. These two are discussed in detail in Section 5.

E. Identifying Gapped Lines

After similar blocks are detected in Step 4, we use a string differences algorithm to detect gaps between blocks and identify line-level gaps.

F. Mapping Similar Blocks to the Original Source Code

All of the code and similar blocks that are detected in Steps 3 and 4 are mapped back to the source code, using by file path, start line and end line.

V. Similarity Measures

We use two similarity measures: 1) Levenshtein similarity and 2) Cosine similarity in Step 4 of our approach discussed in Section 4. We use these two metrics for detecting Type-3 clones.

A. Levenshtein Similarity

Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who proposed this algorithm [8]. It is a metric for measuring the difference between two sequences [7]. It is one of the most widely used algorithms to calculate edit distance. We use Levenshtein distance to detect similar clones in two blocks of code. If the Levenshtein similarity (Eq. 1) is above a threshold value, we declare two fragments, i.e., $B_1$ and $B_2$) are candidate Type-3 clone pairs.

$$\text{Similarity} = 1 - \frac{\text{LevDist}(B_1, B_2)}{\max(\text{Len}(B_1), \text{Len}(B_2))} \times 100$$

where LevDist is the Levenshtein distance and Len($B_i$) and Len($B_j$) are the lengths of two blocks in numbers of lines. The complexity of Algorithm 1 is $O(m \times n)$, where $n$ and $m$ are the lengths of $B_i$ and $B_j$. 

---

Algorithm 2 Type-3 Clone Detection

```java
procedure CloneDetection(BlockFragments)
    Clones ← 0
    for i ← 0, BlockFragments.Length do
        for j ← 0, BlockFragments.Length do
            Sim ← LevenshteinDistance($B_i$, $B_j$)
            if Sim ≥ %60 then
                Clones ← Clones + 1
            end if
        end for
    end procedure

Algorithm 3 Comparison of Two Blocks using CosineSim
1: procedure CosineSimilarityScore(Block1, Block2)
2: for i ← 0, BlockLines1, Length do
3:     if Line.length > 0 then
4:         if FreqVector.containsKey(Line) then
5:             freq1 ← value1 + 1
6:             freq2 ← value2 + 1
7:         end if
8:     UniqueLines.add(Line);
9: end if
10: end for
11: end procedure

Algorithm 4 Algorithm for Type-3 Clone Detection
1: procedure Type-3 Clone Detection
2: for i ← 0, BlockFragments do
3:     for j ← 0, BlockFragments do
4:         Sim ← LevenshteinDistance($B_i$, $B_j$)
5:         if Sim ≥ %60 then
6:             Clones ← Clones + 1
7:         end if
8:     end for
9: end for
10: return Clones
11: end procedure
```

---

For example in File 1, it has detected a class block between lines 886 and 896, and a method block from line 887 to line 888.
B. Cosine Similarity

We also use another measure of similarity, which is cosine similarity. Cosine similarity between two vectors measures the cosine of the angle between them. The bigger the return value, the more similar the two code fragments. Our approach converts each block of code to a vector. The value of cosine similarity (Eq.2) between two code fragments is compared against a constant threshold value, to decide whether two fragments, $B_1$ and $B_2$, should be reported as candidate clone pairs.

$$\cosSim(v_1, v_2) = \cos(\alpha) = \frac{\sum_{i=1}^{n} v_{1,i} \times v_{2,i}}{\sqrt{\sum_{i=1}^{n} v_{1,i}^2} \times \sqrt{\sum_{i=1}^{n} v_{2,i}^2}} \times 100$$

(2)

where $\cosSim$ is the cosine similarity between two vectors $v_1$ and $v_2$ and $\alpha$ is the angle between them.

VI. EXPERIMENT DESIGN

To compare our approach with other detectors in detecting Type-1 and Type-2 clones, we choose eight (CloneDr [3], LD [14], CCFinder [1], Dup [2], Duploc [10], Deckard [12], CDWS [6], and Coarse-grained [5]) detectors and depend on results reported by Hotta et al. [5]. To compare our approach with other detectors in detecting Type-3 clones, we also choose eight (CloneDr [3], CLAN [4], CCFinder [1], Dup [2], Duploc [10], Nick [11], Deckard [12], and CDSW [6]) detectors and use results reported by Murakami et al. [6]. To evaluate our tool, we use source code of four Java projects. Details of the source codes used are given in Table I. Our implementation handles programs in Java only because we use the JDT tool for development. We perform two experiments to answer the following research questions?

RQ1: Is the proposed method more accurate than existing detectors for Type-1 and Type-2 clones?

RQ2: Does the proposed method have higher precision and F-measure than existing detectors for Type-3 clones?

RQ3: Does the proposed method have higher precision and F-measure than existing detectors for all of Type-1, Type-2, and Type-3 clones?

We use the following terms in evaluating our results. A reference is a clone pair that is included in the reference or the true clone set. A candidate is a clone pair that is detected by clone detectors. The contained metric we use is also used in the study of Bellon et al. [7]:

$$\text{contained}(CF_1, CF_2) = \frac{\text{lines}(CF_1) \land \text{lines}(CF_2)}{\text{lines}(CF_1)}$$

(3)

where is $CF_1$ and $CF_2$ refer to the set of lines of code in code fragment $CF$.

We also use the metric $ok$ value to indicate whether a candidate subsumes a reference.

$$ok(CP_1, CP_2) = \min(\max(\text{contained}(CP_1, CF_1, CP_2, CF_1), \text{contained}(CP_2, CF_1, CP_1, CF_1)), \max(\text{contained}(CP_1, CF_2, CP_2, CF_2), \text{contained}(CP_2, CF_2, CP_1, CF_1)))$$

(4)

where is $CP Cf_1$ and $CP Cf_2$ are two code fragments when a candidate clone subsumes a reference clone and satisfies the following condition:

$$ok(CP_1, CP_2) \geq \text{threshold}.$$

The good value metric is defined by Bellon et al. [7], to indicate whether a candidate sufficiently matches a reference. The good-value metric is stronger than the ok-value metric [7]. However, we only use the ok value metric because it is enough to detect Type-1, 2 and 3 clones. We say $CP_1, CP_2$ are clones of each other if the ok value metric is satisfied. We use 0.7 as the threshold, which is also used in Bellon et al.'s study [7], to evaluate the accuracy of detected clones for a given target software system and a given detector $D$. $Cands$ refers to a set of clone pairs, $Refs$ refers to the set of the clone references and $\text{DetectedRefs}$ refers to the set of the clone candidates. The following formulas define precision, recall, and F-measure.

Precision($S, D$) = \frac{\text{DetectedRef}(S, D)}{\text{Cands}(S, D)}

Recall($S, D$) = \frac{\text{DetectedRef}(S, D)}{\text{Refs}(S, D)}

$$F \text{- measure}(S, D) = \frac{2 \times \text{Precision}(S, D) \times \text{Recall}(S, D)}{\text{Precision}(S, D) + \text{Recall}(S, D)}$$

(7)

VII. EXPERIMENTS

We perform two experiments on four target systems that are shown in Table I. The purpose of the first experiment is to determine whether Steps 2 and 3 of our approach produce good recall and higher precision than existing clone detectors. The second experiment is to discover whether Steps 4 and 5 of our approach produce higher precision and F-measure than existing clone detectors or not. We use Murakami's dataset\(^1\), which consists of clone references with gaps [9].

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>Lines</th>
<th>Files</th>
<th>Methods</th>
<th>References/Oracles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netbeans</td>
<td>Java</td>
<td>14,360</td>
<td>97</td>
<td>972</td>
<td>55</td>
</tr>
<tr>
<td>Eclipse-ant</td>
<td>Java</td>
<td>34,744</td>
<td>161</td>
<td>1,754</td>
<td>30</td>
</tr>
<tr>
<td>Java-swing</td>
<td>Java</td>
<td>204,037</td>
<td>414</td>
<td>10,971</td>
<td>777</td>
</tr>
<tr>
<td>Eclipse-jdkcore</td>
<td>Java</td>
<td>147,634</td>
<td>741</td>
<td>7,383</td>
<td>1,345</td>
</tr>
</tbody>
</table>

A. Experiment A

Table II shows the number of detected clone pairs from the Type-1 and Type-2 clone references [5]. In this experiment, we choose detectors that were used in the experiment of Hotta et al. [5] for comparison with our approach. We calculate recall, precision, and F-measure for our approach. For Step of our approach, which is lexical analysis and normalization, we use open source code available at github\(^2\).

2. https://github.com/k-hotta/ECTEC
Recall

Precision

F-measure

Figure 2. The Results of Type-1 and Type-2

The existing detectors results are obtained from Hotta et al.\[5\].

Figure 2(a) shows the comparison of recall of all the clone detectors for Type-1 and Type-2 clone references. CCFinder is the best among all the clone detectors for Eclipse-ant and Javax-swing datasets. LD is the best for Eclipse-jdtcore and Netbeans datasets. Our approach cannot achieve highest recall but is not the lowest in all cases.

Our approach achieves highest precision compared with others. Figure 2(b) shows the values of precision of all the clone detectors for Type-1 and Type-2 clone references. Our approach gets first position for Netbeans, Eclipse-jdtcore, and Java-swing datasets. It gets the third position for Eclipse-ant dataset because the Eclipse-ant dataset has only a few reference clones and some of these clones are divided into two parts: one part in one block and the second part in another block.

Figure 2(c) shows the values of F-measure for all the clone detectors. Our approach gets the first position for Eclipse-jdtcore and Java-swing datasets. It gets second position in Eclipse-ant and Netbeans datasets. It achieves a good balance of recall and precision for the Type-1 and Type-2 clones references.

B. Experiment B

The goal of Experiment B is to answer questions RQ2 and RQ3. In this experiment, we choose the detectors used in the Murakami et al.'s [6] experiment to compare with our approach using Levenshtein distance and cosine similarity. We also calculate recall, precision, and F-measure of our approach.

Figure 3(a) shows the comparison of recall for all the clone detectors for Type-3 clone references. CCFinder is the best among the clone detectors. The median and average of CCFinder are the best. The median and average of our approach are in the middle behind CCFinder, CDSW, Dup, and Deckard. Figure 3(b) shows the case of precision. Our approach using Levenshtein distance ranks first in precision and our approach using Cosine similarity gets the second position. We conclude that our approach achieves high precision compared with other detectors. Figure 3(c) shows the comparison of F-measure. The median and average of our approach using Levenshtein distance gets first and our approach using cosine similarity gets second positions. Because of the value precision of our approach using Levenshtein distance or cosine similarity is high and the value F-measure of our approach either using Levenshtein distance or cosine similarity is high as well. Figure 3(b) and 3(c) show that our approach in both cases is the best in precision and F-measure. Therefore, we achieve our objective and answer RQ2 positively.

Figure 4(a) shows the recall for all the clone detectors for Type-1, 2 and 3 clones. The median of CCFinder is still the best among all the clone detectors. Our approach cannot achieve the highest recall but we conclude that our approach is not the lowest recall. Figure 4(b) shows the value of precision. Our approach using Levenshtein distance ranks second in precision after CDSW, and our approach using cosine similarity gets the third position out of eight detectors in this case. CloneDr is the fourth position. We conclude that our approach achieves high precision for Type-1, 2, and 3 clones. Figure 4(c) shows F-measure. Both median and average of our approach in both cases gets the first and second positions. Figure 4(c) also shows our approach in both cases is the best in F-measure. Therefore, we answer RQ3 positively as well.
TABLE II. COMPARISON OF THE MEDIAN OF OUR APPROACHES FOR TYPE-3 CLONES, RESULTS WITH OTHER EXISTING APPROACHES. THE EXISTING DETECTORS RESULTS ARE OBTAINED FROM MURAKAMI ET AL.[6]. THE BEST ENTRIES ARE IN BOLDFACE. H=HIGH, M=MEDIAN, A=AVERAGE.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H.</td>
<td>M.</td>
<td>A.</td>
</tr>
<tr>
<td>CloneDr</td>
<td>10.8</td>
<td>8.55</td>
<td>17.55</td>
</tr>
<tr>
<td>CLAN</td>
<td>100</td>
<td>79.4</td>
<td>73.4</td>
</tr>
<tr>
<td>CCFinder</td>
<td>100</td>
<td>64.5</td>
<td>73.0</td>
</tr>
<tr>
<td>Dup</td>
<td>100</td>
<td>79.4</td>
<td>73.0</td>
</tr>
<tr>
<td>Duploc</td>
<td>100</td>
<td>79.4</td>
<td>73.0</td>
</tr>
<tr>
<td>DECKARD</td>
<td>88.9</td>
<td>53.85</td>
<td>56.03</td>
</tr>
<tr>
<td>NICAD</td>
<td>100</td>
<td>53.85</td>
<td>56.03</td>
</tr>
<tr>
<td>CDSW</td>
<td>100</td>
<td>53.85</td>
<td>56.03</td>
</tr>
<tr>
<td>LeveDist</td>
<td>64.25</td>
<td>50.76</td>
<td>53.94</td>
</tr>
<tr>
<td>CosineSim</td>
<td>66.73</td>
<td>56.85</td>
<td>56.22</td>
</tr>
</tbody>
</table>

TABLE III. COMPARISON OF THE MEDIAN OF OUR APPROACHES FOR TYPE-1,2, AND 3 CLONES, RESULTS WITH OTHER EXISTING APPROACHES. THE EXISTING DETECTORS RESULTS ARE OBTAINED FROM MURAKAMI ET AL.[6]. THE BEST ENTRIES ARE IN BOLDFACE. H=HIGH, M=MEDIAN, A=AVERAGE.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H.</td>
<td>M.</td>
<td>A.</td>
</tr>
<tr>
<td>CloneDr</td>
<td>48.10</td>
<td>21.60</td>
<td>71.2</td>
</tr>
<tr>
<td>CLAN</td>
<td>56.1</td>
<td>37.0</td>
<td>38.43</td>
</tr>
<tr>
<td>CCFinder</td>
<td>100</td>
<td>83.95</td>
<td>85.03</td>
</tr>
<tr>
<td>Dup</td>
<td>80.2</td>
<td>73.2</td>
<td>71.2</td>
</tr>
<tr>
<td>Duploc</td>
<td>46.7</td>
<td>25.0</td>
<td>24.18</td>
</tr>
<tr>
<td>DECKARD</td>
<td>85.4</td>
<td>69.0</td>
<td>65.85</td>
</tr>
<tr>
<td>NICAD</td>
<td>76.3</td>
<td>40.2</td>
<td>47.35</td>
</tr>
<tr>
<td>CDSW</td>
<td>70.6</td>
<td>51.93</td>
<td>55.29</td>
</tr>
<tr>
<td>LeveDist</td>
<td>62.73</td>
<td>45.66</td>
<td>47.68</td>
</tr>
<tr>
<td>CosineSim</td>
<td>62.58</td>
<td>48.16</td>
<td>48.89</td>
</tr>
</tbody>
</table>

Figure 3. The Results of Type-3

VIII. DISCUSSION

A. Clone References

It is hard to manually check for true clones by manual comparison in a target software system. Therefore, for fair comparison, we use datasets of Murakami and Bellon, which recent detectors have used, to compare our approach with others. Murakami et al.’s clone references [9], reannotate the clone references of Bellon et al. [7] with information about gapped lines. A change in clone references can affect the results of precision.

B. Hashing Collision

We use hash values, which as mentioned earlier, are computed using `hashcode()` method that produces a product sum over the entire text of the string, for comparing two blocks of code. We use the same hash function, which is supported by a Java library that Hotta et al. [5] use in their approach. The difference between their approach and ours is that our approach classifies blocks as class blocks, method blocks, if statement blocks, and for statement blocks and hashes these normalized blocks separately for reducing the chances of collision.

C. Different Programming Languages of Target Software Systems

Because we use the Java Development Tool (JDT) that parses only Java code, we are able to perform experiments in this study for Java projects only.
We plan to perform experiments in C and other programming languages to judge how our approach extends to them.

D. Thresholds for Levenshtien Distance and Cosine Similarity

In this study, we choose the threshold of similarity between two blocks to be 35% - 99%. When we apply 100% threshold value, Type-1 and Type-2 clones are detected. With less than 35% threshold value, some Type-3 clones are missed or more false positives clones are detected. We apply different threshold values for Levenshien distance and Cosine similarity computations as shown in Figures 5 and 6. We conclude that the best range of threshold is \( \text{threshold} \geq 60\% \) for Levenshtien distance and \( \text{threshold} \geq 70\% \) for Cosine similarity. We compare the median and average of our results with the other existing tools for Type-3 clones as shown in Table II and Figure 3. We also compare the median and average of our results with the other existing tools for Type-1, 2 and 3 clones as shown in Table III and Figure 4.

IX. CONCLUSION

This paper has presented a hybrid clone detection technique that first uses a coarse-grained technique to improve precision and then a fine-grained technique to get more information about clones and to improve recall. We use hash values and grouping of blocks to detect Type-1 and Type-2 clones, and Levenshien distance and cosine measures for blocks to detect gapped code clones (Type-3). Our experimental results indicate that our method achieves high precision and F-measure in most cases compared to other detectors. In this paper, we demonstrate the following.

- Normalizing blocks followed by grouping and hashing helps detect Type-1 and Type-2 clones.
- We use two similarity measures to detect Type-3 clones and tailor the Levenshien distance algorithm to use for code clone detection. Levenshien distance is a string metric for measuring the distance between two sequences. The tailored Levenshien distance algorithm can measure distance between lines of code. We also use cosine similarity, tailored to
measure angular distance between lines, represented as vectors.

- We demonstrate that our proposed method has higher precision and F-measure than existing methods.

In future, we plan to extend our experiments using C datasets and larger software systems. If different programming languages and large datasets are used for evaluation, the results are more likely to reflect the true quality of our approach. At present, only a few techniques can detect Type-4 clones. We also plan to extend our work to detect Type-4 clones. Since references from Bellon et al. [7] or Murakami et al. [6] do not explicitly contain Type-4 clones, we also plan to construct clone references for Type-4 clones.

ACKNOWLEDGMENT

This work was performed in the LINC Lab at University of Colorado at Colorado Springs. It was supported by Jazan University, Saudi Arabia. The authors would like also to thank Keisuke Hotta and Hiroaki Murakami for helpful suggestions and answers of their questions.

REFERENCES


