LVMapper: A Large-variance Clone Detector Using Sequencing Alignment Approach

Anonymous Author(s)

***Abstract*—To detect large-variance code clones (i.e. clones with relatively more differences) in large-scale code repositories is difficult because most current tools can only detect almost identical or very similar clones. It will make promotion and changes to some software applications such as bug detection, code completion, software analysis, etc. Recently, CCAligner made an attempt to detect clones with relatively concentrated modifications called large-gap clones. Our contribution is to develop a novel and effective detection approach of large-variance clones to more general cases for not only the concentrated code modifications but also the scattered code modifications. A detector named LVMapper is proposed, borrowing and changing the approach of sequencing alignment in bioinformatics which can find two similar sequences with more differences. The ability of LVMapper was tested on both self-synthetic datasets and real cases, and the results show substantial improvement in detecting large-variance clones compared with other state-of-the-art tools including CCAligner. Furthermore, our new tool also presents good recall and precision for general Type-1, Type-2 and Type-3 clones on the widely used benchmarking dataset, BigCloneBench.**

***Index Terms*—clone detection, large-variance clone, dynamic threshold, sequencing alignment**

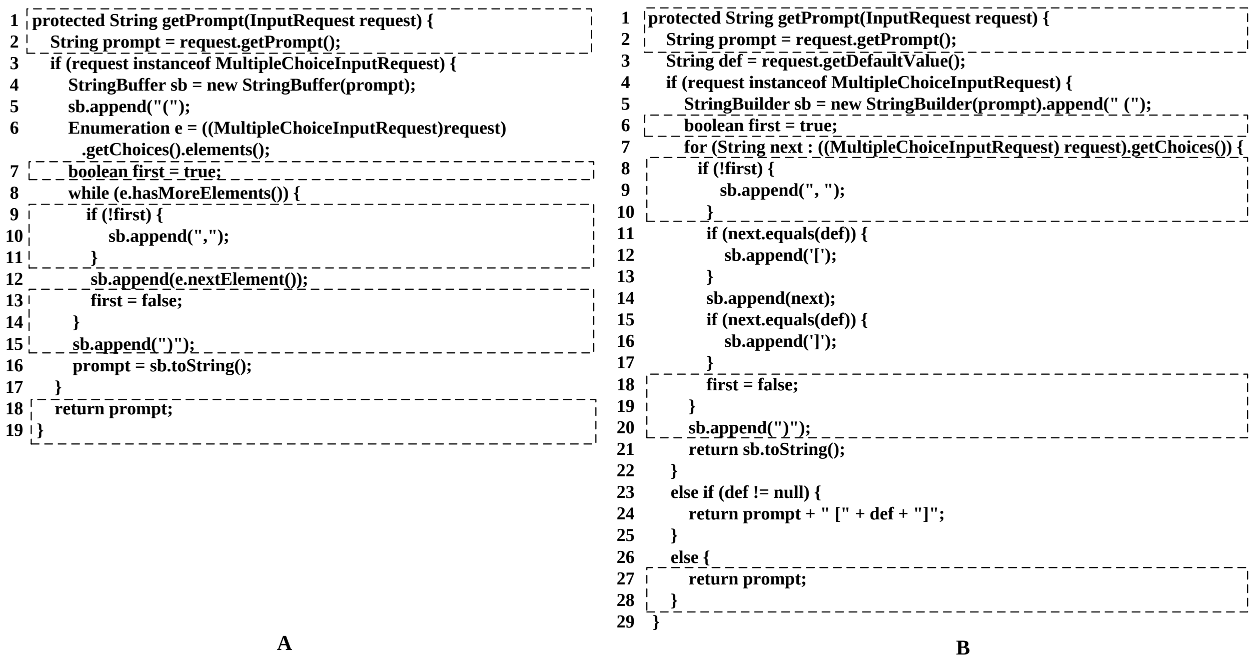
I. Introduction

Clone code is generated by copying, pasting and modifying code fragments for reuse, which are common operations in software development [1], [2]. In the past, code clones with relatively more modifications (called large-variance code clones) were difficult to be found by existing tools [3] because most of them were suitable for finding the identical or very similar clones. However, in our experimental observation, large-variance code cloning is ubiquitous. Compared with the clones provided by traditional tools, the large-variance code clones have a broader range. Accordingly, they have an important impact on and make changes to some software applications such as bug detection, code completion, software analysis and so on. Recently, a meaningful attempt has been made with CCAligner [3]. It is a large-gap clone detection tool which can detect the code clones with relatively concentrated modifications. Our study focuses on a more general case: to detect large-variance code clones that include not only the clones with concentrated code modifications but also those with scattered code modifications.

Software development usually involves two modes: *homologous modification* and *heterologous development*. In *homologous modification* there is always an original version of code and modifications on the original version generate new clone code. Hence, the similarity in lexical is the inherent characteristic of *homologous modification*. By contrast, in *heterologous development*, different programmers develop similar functionalities and these introduce semantic clones, where the lexical similarity is not inherent. Furthermore, large-variance clones of *homologous modification* are generated by two cases. One of the common cases is that the old version of software are modified and expanded iteratively. Another case is the reuse and modification of open source software. Both cases are very common in software development, therefore it is important to detect the clones with *homologous modification*. Fig. 1 shows an example of clones between two different versions of project *Ant 1.6.5* and *1.10.5*. In code B, the code segments of lines 1–2, 6, 8–10, 18–20, 27–28 are the same as those of the lines 1–2, 7, 9–11, 13–15, 18–19 in code A, respectively. Lines 4–5, 7, 14 and 21 in code B are modified from lines 3–5, 8, 12 and 16 in code A. Other lines in code B are new extension part of code A. These modifications in clone codes are scattered and occupy a certain portion of the code.

Existing tools have made attempts but still have more or less limitations in finding Type-3, especially large-variance code clones. For one of the best tools with good performance on Type-3 clone, SourcererCC [4] has to decrease the threshold of similarity to find large-variance clones at the cost of accuracy loss. Another popular detector CCFinderX is good at identifying Type-1 clones and Type-2 clones, but cannot directly support Type-3 clone detection. iClones identifies the Type-3 clones by merging the nearby small Type1-2 clone fragments, but the recall of Type-3 is low due to the simple strategy. NiCad uses a Longest Common Sub-sequence (LCS) algorithm and can tolerate discontinuous subsequences. However, it does not scale and its precision suffers with decreasing thresholds. CCAligner is a good recent attempt in detecting clones with relatively concentrated modifications. It can detect large-gap but misses scenarios where modifications are scattered. Besides, some semantic methods have certain ability to detect variance clones because there is an overlap between semantic clones and syntactical clones. Deckard [5] builds the characteristic vectors from abstract syntax tree (AST) to detect clones, but suffers from low precision and recall rate. Deep learning methods such as Oreo [6] encode software metrics into semantic vectors and achieve good results, but they mainly focus on semantic clones.

For these considerations, we present a tool aimed at detecting large-variance code clones called LVMapper. Our proposed code clone detector that can find clones with more general variance is based on locate-filter-verify method. Its key idea mainly comes from third-generation sequencing alignment method [7]–[9]. In bioinformatics, the third-generation se-



```
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

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

B

Fig. 1. Example of a large-variance clone.

quencing alignment based on seed-and-extend strategy performs well with sequence difference up to 30%. LVMapper uses small windows of continuous lines (called *seeds*) with lower costs to locate and filter the candidate pairs of clone codes. In order to verify whether these candidate pairs of codes are cloned, another feature that code clones always have certain proportion of order-preserving code lines is considered. Based on this property, a heuristic algorithm which is more efficient than the Longest Common Subsequence (LCS) algorithm is proposed. Besides, a dynamic threshold that changed with the code size is used for the verification of code clones. It makes LVMapper identify clones with more modifications while guaranteeing certain precision.

To evaluate the large-variance clone detection performance of LVMapper, we carried out experiments on self-synthetic and real datasets. In order to test the capability of finding large-variance clone, we compared our tool's performance with NiCad, SourcererCC and CCAligner on 4 Java and 4 C projects. For the self-synthetic dataset experiment, we generated clones by inserting scattered different lines to source code, and then evaluated the performance of LVMapper and other tools in detecting clones with various modifications. We also used the BigCloneBench [10], [11] to compare and measure the different type clones recall of LVMapper with CCFinderX [12], iClones [13], Deckard [5], NiCad [14], SourcererCC [4] and CCAligner [3]. The experiments show that LVMapper performed the best in detecting large-variance clones and had good recall and precision for general Type-1 to Type-3 clones.

The main contributions of our work are as follows:

(1) Goal contribution: CCAligner has advantages in detecting large-gap clones while our work extends the detection approach of large-variance clones to more general cases. It identifies not only the clones with concentrated code modifications but also the clones with the scattered code modifications. We also give a concrete definition of the large-variance clones.

(2) Method contribution: Inspired by the idea of the seed-and-extend method in bioinformatics, we develop a novel tool with locate-filter-verify procedure and it is suited to detect clone with large variance. We propose a dynamic threshold to promote the accuracy and recall, and a rapid method to verify, avoiding the time-consuming dynamic programming.

(3) Result contribution: We compared LVMapper with other state-of-the-art detectors on real cases of software projects, self-synthetic programs and the state-of-the-art benchmarks. The results show that LVMapper is much better than the state-of-the-art tools in large-variance clone detection. In addition, our new tool has good recall and precision for general Type-1, Type-2 and Type-3 clones.

The rest sections of the paper are organized as follows. Some terminologies and definitions of code clone are introduced in Section II. Section III provides the details of our detection tool. Section IV presents the results of the experiments to evaluate the detection ability of our approach. In Section V the related work of clone detection is discussed and Section VI describes the limitation. Finally, Section VII concludes the paper and briefly introduces future work.

II. Terminologies & Definitions

Code block is a statement sequence within braces and usually represents a single function. *Clone pair* is a pair of similar code portions. The *minimum clone size* is the minimum number of lines or tokens that either of a clone pair should have. The standard minimum clone size is 6 lines or 50 tokens which we also follow in this paper. Four primary clone types are agreed by researchers and the former work [1], [15]:

Type-1 (textual similarity) and *Type-2 (lexical similarity)* clones are syntactically identical code fragments except for variances in white space, layout, comments and variances in identifier names, literal values, white space, layout and comments, respectively. *Type-3 (syntactic similarity)* clones are code fragments which are similar but have statements added, modified and/or removed with respect to each other. *Type-4 (semantic similarity)* clones are code fragments that implement the same functionality but are different in syntax.

Type-3 and Type-4 clones are difficult to partition because there is no clear boundary between syntactically similar Type-3 clone and dissimilar Type-4 clone. Hence, BigCloneBench [10] further divided Type-3 and Type-4 into four types according to the syntactical similarity range: Very Strong Type-3 similarity in range [0.9, 1.0), Strongly Type-3, [0.7, 0.9), Moderately Type-3, [0.5, 0.7), and Weakly Type-3&4, [0.0, 0.5).

Before giving the definition of large-variance clones, we first define a new similarity of code pairs and then obtain the difference degree of code pairs. In the past, Jaccard similarity [16] is usually used to measure the similarity of code pairs, which is not suitable for code pairs with large difference in size. For example, given code block *A* and *B*, assume the size of *B* is double the size of *A* and all lines of *A* appears in *B*. According to the Jaccard similarity, the similarity of code blocks *A* and *B* are only 0.5, which is not practical. In fact, all of *A* are cloned by *B*.

Definition 1 Similarity and Difference of code pairs: Given two code blocks *A* and *B* consisting of $l(A)$ and $l(B)$ pretty-printed lines, respectively. Let $l(A \cap B)$ be the number of shared lines in *A* and *B*. The harmonic similarity *sim*(A, B) of *A* and *B*, the single-side similarity *sim*$(A|B)$ that *A* relative to *B*, and the single-side similarity *sim*$(B|A)$ that *B* relative to *A* are:

$$sim(A,B) = \frac{1}{2} \cdot \left(\frac{l(A \cap B)}{l(A)} + \frac{l(A \cap B)}{l(B)} \right) \tag{1}$$

$$sim(A|B) = \frac{l(A \cap B)}{l(A)}, sim(B|A) = \frac{l(A \cap B)}{l(B)} \tag{2}$$

The difference *diff(A,B)* of *A* and *B*,and the difference *diff*$(A|B)$ that *A* relative to *B*, and the difference *diff*$(B \mid A)$ that *B* relative to *A* are defined according to the similarity, respectively:

$$diff(A,B) = 1 - sim(A,B) \tag{3}$$

$$diff(A|B) = 1 - sim(A|B), diff(B|A) = 1 - sim(B|A) \tag{4}$$

Taken the previous example of code *A* and code *B* which is double size of *A*, the single-side similarity *sim*$(A|B) = 1$, which means that *A* is all in *B* and it matches the actual situation. The harmonic similarity *sim(A,B)* $= 1/2 \cdot (\frac{m}{m} + \frac{m}{2m}) = 0.75$, where *m* is the number of lines in code *A*, is greater than the Jaccard similarity. On these bases, we give the definition of large-variance clone as follows.

Definition 2 Large-variance clone: If code blocks *A* and *B* are clone and *diff(A,B)* $> \Delta$ (or *diff*$(A|B) > \Delta$, or *diff*$(B|A) > \Delta$), then *A* and *B* are Δ-difference clone ($0 \leq \Delta \leq 1$). Particularly, for *diff(A,B)* Δ is set as 0.15, i.e. *diff(A,B)* > 0.15, then *A* and *B* are called large-variance clone (abbreviated as *LV clones*).

The difference degree of large-variance clones is set higher than 0.15 mainly based on two considerations. First, it's difficult for most code clone detection tools or methods to find such clones, even the better tools that are able to detect Type-3 clones cannot find them well. Another consideration is the compatibility with the large-gap clones described in [3], in which the volume difference between blocks *A* and *B* is used to specify variance clones. Assuming that *A* is the block with smaller size and *B* is the block with larger size, their method is to detect the large-gap clones for the set of blocks λ-difference volume ($\lambda = l(A)/l(B) \leq$ 0.7). According to our definitions, the *sim(A,B)* is at most 0.85, then the *diff(A,B)* is at least 0.15. Hence, the code clones described in [3] are included in our large-variance clones but ours have a broader scope.

III. METHOD

Inspired by the seed-and-extend approach which is typically used in sequencing alignment from bioinformatics [7], [8], [17], we proposed a locate-filter-verify procedure for clone detection. In bioinformatics, the seeding step uses the subsequences of the query to quickly locate exact match in reference and the extending step extends and refines the candidate positions by a dynamic programming alignment. Our method includes three phases: locate-filter-verify. The first two phases are designed to seek out the candidate clone pairs with low cost and high recall. In the last phase we design a heuristic algorithm to further eliminate the false clone pairs and improve the accuracy. The uses of dynamic threshold, seeds index and avoiding time-consuming dynamic programming are the keys and innovations of our method. Fig. 2 shows the general procedure. The rest of this section will provide detail descriptions of each phase.

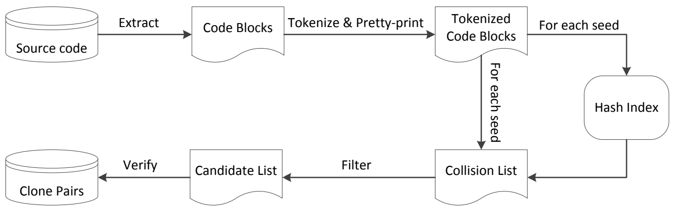


Fig. 2. General procedure of LVMapper.

A. Lexical Analysis

Lexical analysis for code includes extracting code blocks from source code and tokenizing the code. TXL [18], which is commonly used in previous tools [3], [4], is adapted to extract code blocks from source code files. After obtaining the code blocks, the tokenizing step that mostly based on Flex [19] begins. Identifiers including variables and function names are replaced by the same token 'id' to tolerate Type-2 changes. The extracted code blocks are pretty-printed. The tokens of each line are concatenated into a single token sequence except the white spaces.

B. Seeds Indexing

It is necessary to establish a seed index to speed up the computation for the locating and filtering phases, where the *seeds* are all of the *k*-line sliding windows (i.e., code fragments of continuous *k* lines) for a code block. The seeds are basic units for matching instead of tokens. For example, given a code block with 10 lines and sliding windows size *k* = 3, the number of all seeds is obviously 8. In LVMapper, these seeds are converted to hash value and the seed is also regarded as the hash value.

Here are the detailed steps of indexing. LVMapper scans all code blocks, collects all seeds and indexes them in a hash table. The key of the element in hash table is seed's hash and the value is a set of corresponding positions (e.g., block number plus line number). Like CCAligner, we use MurmurHash hash function [20] in order to guarantee the efficiency with the low collision rate. For the value in the hash table, we packed the block number and line number into a 64-bit integer.

While scanning code blocks, the pairs *(l_hash, l_num)* are also saved in the line hash vector for each code block, where *l_hash* is one-line hash of the block (ranges from line 1 to the block size) and *l_num* is the corresponding line number. This information will be used in the next phases.

C. Locating via the Shared Seed

The locating phase is a preliminary selection for possible code clone pairs. In this phase, the goal is to collect as many candidates as possible without losing real clone pairs. Because the standard minimum clone size is 6 lines [10], [15], CCAligner chooses 6-line or longer windows to match the possible clone pairs. The experiments in CCAligner [3] shows that 6 lines with 1 mismatch windows balanced recall with precision. However, the cost of this approach is still considerably high. We use a lower cost and more efficient way to achieve this. Here, the 3-line sliding windows are chosen as seeds to collect the possible clone pairs that share seed(s).

For any code segments of 6-line with 1 mismatch that can be found by CCAligner, in LVMapper, they can also be identified by two non-overlapping 3-line exact windows. The reason is that the 6 lines window in CCAligner can be covered by two non-overlapping 3-line windows. According to pigeonhole principle, one line modification affects at most one 3-line window and the other one remains unchanged. As a result, the identification ability of our method is better than that of CCAligner.

Algorithm 1 lists the steps of clone detection process, in which lines 4–11 belong to the locating phase. To retrieve these seeds, we create a hash table for efficiency. Once the index has been built, the candidates of each code clock can be obtained by utilizing this index. Let the current inquiring code block be *A*. Every sliding overlapping 3-line window in *A* is the seed, whose hash value is used to find positions in hash table with the same key (*line 9*). If the block id of the position, denoted as *B*, is greater than *A*, the position will be added to *CollisionList (A)* the collision list of block *A* (*line 11*). This practice eliminates the duplication of detecting clone for the same two blocks with reverse order. When *B* is the current inquiring block, block *A* is not considered anymore because the pair of *A* and *B* has been considered before. After the last seed of current block is queried, the positions in the *CollisionList (A)* are sorted according to block id and then according to line id (*line 12*). Every block *B* that has the collided seed with *A* will be further filtered and verified.

D. Filtering via the Common Seeds Number

Through the locating phase, two code blocks that share common seed(s) may be clone pair. Then we take into account the clone possibility of these *Collision* code blocks. This phase is called the filtering phase and it picks out candidate clone pairs. It gives us two benefits. First, the number of candidate code pairs can be reduced significantly. Therefore the processing time and the false positive rate can also be reduced. Besides, the probability of the candidate pairs being the true clones increase significantly. Our idea of the filtering phase is mainly based on considering the number of shared seeds for two code blocks to measure the possibility of being clone.

The selection of candidate clone pairs depends on the similarity between the two code blocks, and the similarty is calculated by the number of seeds they share. In our method, we adopt the Equation (2) to compute the similarity of a code pair, because the definition of single-side similarity has better ability to discover large variance code clones. As we get the number of shared seeds between two code blocks, the former of Equation (2) can be equivalently changed to:

$$SR_A(B) = \frac{s}{t} = \frac{s}{L-k+1} \tag{5}$$

where *s* is the number of shared seeds, *t* is total seeds number of *B* and *L* is the length in line of *B*. For any pair of the collision code blocks, the higher the $SR_A(B)$ value is, the more likely they are to be a clone pair.

The method of threshold setting for $SR_A(B)$ is the key point of our method, and it is also significantly different from other methods. In order to enhance the ability of detecting large variance code clones, here we use a dynamic threshold to filter the code pairs. The function of the threshold is designed as a multi-segment function, corresponding to the length of code blocks A and B. As the length of code blocks A and B increases, this threshold can be reduced. Actually,

Algorithm 1: Clone Detection

```
Input: $A$ is a list of tokenized code blocks $\{a_1, a_2, ...a_n\}$, Hash Table $H$ of
       $A$, window size k, threshold $\theta$ for filtering phase, threshold $\delta$ for
       verifying phase
Output: All clone pairs $CP$
1  $H \leftarrow \varnothing$;
2  $CP \leftarrow \varnothing$;
3  $len$ = number of lines in $a_i$;
4  for each $a_i$ in $A$ do
       /* Locating phase */
5      for $j = 1; j \leq len - k + 1; j++$ do
6          $l_j = a_i.linej$;
7          $win_j$ = CONCAT($l_j, l_{j+1}, ...l_{j+k-1}$);
8          $key$ = HASH($win_i$);
9          $pos$ = FIND($H, key$);
10         if $pos > a_i$ then
11             $CollisionList_i = CollisionList_i \cup pos$;
12     SORT($CollisionList_i$);
13     for each block $B$ in $CollisionList_i$ do
           /* Filtering phase */
14         $s$ = number of $B$ in $CollisionList_i$;
15         $L$ = number of lines in $B$;
16         $SR = s/(L - k + 1)$;
17         if $SR \geq \theta$ then
               /* Verifying phase */
18             $block1$ =block with smaller size between $a_i$ and $B$;
19             $block2$ =block with larger size between $a_i$ and $B$;
20             $line1 = 1$;
21             $comm\_lines = 0$;
22             $lastline = 0$;
23             while $line1 \neq block1.end$ do
24                 $k1$ = HASH($block1.line1$);
                   /* FIND_IN_BLOCK find the first line after
                      $lastline$ in $block2$ and has the same hash
                      value of $k1$ */
25                 $line2$ = FIND_IN_BLOCK($k1, block2, lastline$);
26                 if $line2 \neq NULL$ then
27                     $seglen = 0$;
28                     $m = 1$;
29                     while $block1.line1 + m = block2.line2 + m$
                         do
30                         $seglen++$;
31                         $m++$;
32                     if $seglen \geq 2$ then
33                         $comm\_lines+ = seglen$;
34                 $line1++$;
35             $min\_lines$ = minimum lines of $a_i$ and $B$;
36             $OS = comm\_lines/min\_lines$;
37             if $OS \geq \delta$ then
38                 $CP = CP \cup (a_1, B)$;
39 return $CP$;
```

the reason can be explained by an understandable analogy. For example, given real-life conversations, how to judge whether two different conversations belong to the same topic? We have such a consensus that long conversations with lower rate of common sentences could discuss the same subject while short conversations should have higher rate to judge as discussing the same subject. In the experiments of next section, the filtering threshold is set to about 0.1 for both code blocks with lengths more than 10. Note that SourcererCC uses the fixed ratio of shared tokens to verify clone pairs and it sets the ratio threshold as 0.7 to guarantee the precision.

Lines 13–16 in Algorithm 1 belongs to filtering phase. In practice, once we get the candidate blocks list *CollisionList (A)* of current inquiring block *A*, for every candidate block *B* in the list, LVMapper counts the number of *B* in *CollisionList (A)* (*line 14*). As we mentioned above, we treat each overlapping *k*-line windows as seed to vote for potential clone blocks, then every position added in the collision list is the block that have the same seed with *A*. In this case, the number *B* in *CollisionList (A)* is the number of votes that *B* gets from *A*. If *B* gets more votes, then it is more likely to be clone code of *A*. The idea is similar to the idea of seed-and-extend in the sequencing alignment [8]. The threshold for $SR_A(B)$ is θ (*line 17*).

E. Verifying via the Ordered Common Lines

Unlike the first two phases, the last phase (called the verifying phase) further measures the clone possibility of the candidate clone pairs output by the filtering phase from another perspective: if two blocks are large-variance clone, an important feature of them is that the common lines of code in them have order preserving property. Actually, previous tool NiCad [14] used similar idea. It is based on a Longest Common Sub-sequence (LCS) algorithm. Not as complex as NiCad, we design a heuristic simple algorithm for this order preserving property. The idea of the heuristic algorithm is to count the order preserving number of two adjacent code lines in one code block.

Similarly, the similarity of the candidate pair is measured by another characteristic quantity: the rate of ordered common lines. This characteristic quantity *OS (A, B)* is defined as:

$$OS(A, B) = \frac{comm\_lines}{min\_lines(A, B)} \tag{6}$$

where *common_lines* is the ordered common lines of *A* and *B*, and *min_lines(A, B)* is the minimum size in line of *A* and *B*. Like the threshold mentioned in the filtering phase, we also use a dynamic threshold for verifying candidate clone pairs. The detailed setting will also be discussed in Section IV.

We implement the heuristic algorithm as follows and lines 17–38 in Algorithm 1 show the steps. For every candidate pair of block *A* and *B* survived from the locating and filtering phase, assume block *A* is smaller than *B*. We scan from the first line in *A* to find contiguous lines which also appear in *B* (*line 25*). LVMapper keeps a variable *comm_lines* to record the sum of matching lines. If there are at least 2 contiguous shared lines, then the length is added to *comm_lines* (*line 33*). The contiguous lines of A and B are in order and the segments of the contiguous lines are also in order. The threshold δ of *OS (A, B)* to verify candidate pairs (*line 37*) is set according to the minimum size of *A* and *B*.

IV. Evaluation

The performance of LVMapper for detecting large-variance clones and general Type-1, Type-2, Type-3 were thoroughly evaluated in real and self-synthetic dataset. We first introduce the parameter setting of seed length and dynamic threshold of different phases. Then the detailed information of different experiments is provided.

A. Parameter Setting

1) Choice of Seed Length: As the seeds play a major role in locating and filtering phases, the choice of seeds length is important and has an impact on the performance of LVMapper. If the seeds are too long, the recall of our method will be affected. In contrast, if the seeds are too short, the effectiveness of the locating and filtering will be eroded. In Section III we analyzed the choice of seed length theoretically. Here we also used experiments to evaluate the performance of detection for different seed lengths k quantificationally. We used the BigCloneBench [10], [11] to evaluate the detection ability of LVMapper for different seed lengths, because it is not only a benchmark for general clones but also contains large-variance clones. Besides, we considered the memory use and execution time of different seed lengths for Linux kernel dataset.

BigCloneBench is a benchmark which contains different types of manually validated clones in the repository IJaDataset-2.0 [21] and it defines clone types by syntactic similarity as described in Section II. The framework BigCloneEval [22] summarizes recall performance for different clone types of clone detectors automatically and it is widely used in previous work [4], [6]. We configured the BigCloneEval with minimum clone size 6 lines and 50 tokens which are consistent with the standard minimum clone size. The seed length of LVMapper was set as 2-line, 3-line and 4-line, with other parameters fixed. The recall was reported by BigCloneEval. And for each parameter, we measured the precision by randomly validating 400 reported clone pairs.

Table I shows the detailed results. Because the recall rate of Weakly Type-3&4 is under 1%, we provided the number of detected clones instead, denoted as *# of Weakly Type-3&4*. As seen from Table I, the recall of Type-1, Type-2 and Very Strongly Type-3 were nearly 100% for all seed length. When the seed length became longer, the recall of type-3 with lower similarity decreased. For seed length of 4-line, the recall of Strongly Type-3 and Moderately Type-3 was under 80% and 20%, respectively. And the number of Weakly Type-3&4 fell to 22998. However, while the recall for seed length of 2-line was the highest, especially in Weakly Type-3&4, the precision declined apparently. The recall and precision for seed length 3-line strike a balance. The number of Weakly Type-3&4 was over 30000 and the precision kept at 88%.

Besides, we took into consideration the memory use and execution time of different seed lengths. The Linux kernel 4.18 was used as the target source code and it has 25782 files with 12964738 lines of code (LOC) measured by cloc [23]. As shown in Table II, the execution time of 2-line method was as much as 19 times compared to the execution time of 3-line method and the memory use increased about 100MB. The configuration of seed length 3-line had the least memory use and medium execution time. The execution time of 4-line method was the shortest, but it had more memory requirement. Note that the configuration of seed length 3-line has the least memory use. The reason is that for the method using 4-line, larger window space allows greater variation of seed, resulting in greater hash index space. For the method using 2-line, more position values (i.e., the block id and line id corresponding to the seeds) occupy the storage space. Taken together, the seed length of 3-line balanced not only the recall and precision, but also the space and time. Therefore, we selected 3-line as our default configuration.

TABLE I
RECALL PER CLONE TYPE AND PRECISION MEASURED FOR BIGCLONEBENCH WITH DIFFERENT SEED LENGTH

k	2	3	4
Type-1	100	100	100
Type-2	99	99	99
Very Strongly Type-3	98	98	98
Strongly Type-3	85	81	77
Moderately Type-3	21	20	18
# of Weakly Type-3&4	35613	30250	22998
Precision	85	88	89

TABLE II
EXECUTION TIME AND MEMORY SPACE WITH DIFFERENT PARAMETERIZATION FOR LINUX 4.18

k	2	3	4
Time	11h 28m 48s	36m 16s	13m 32s
Memory	841 MB	760 MB	834 MB

2) Threshold for Filter and Verification: In filtering and verifying phases, we use dynamic threshold for judgment of results. The threshold is defined according to the block size in order to be better adapted to code clone judgment.

For θ, the threshold of $SR_A(B)$ in the filtering phase, it is defined as:

$$\theta = \begin{cases} \mu * L + \nu & \text{if } 6 < L \leq 10, \\ 0.1 & \text{if } L > 10. \end{cases} \tag{7}$$

where L is the size in length of the collided block B. We set θ = 0.5 when B has 6 lines and we set θ = 0.1 when the size of B is greater than 10 lines. When the code size is small, there are plenty of statements that have similar forms. So LVMapper filters the smaller candidate block with stricter standards. The use of the dynamic threshold utilizes the characteristic of source code.

Moreover, for the ratio of ordered common sequences in the verifying phase, assume the smaller block of the candidate pair is block A. We adapt a more-refined piecewise function to define the threshold δ, which is used in the verifying phase, according to *A.size l*:

$$\delta = \begin{cases} 0.55 & \text{if } 6 < l \leq 10, \\ g(l) & \text{if } 10 < l \leq 20, \\ 0.3 & \text{if } l > 20. \end{cases} \tag{8}$$

In Equation (8), g(l) = $-\alpha \cdot l + \beta$ relies on size of A. In our implementation, we set α = 0.025, β = 0.8 empirically. For the smaller blocks whose length is smaller than 10 lines, the ratio of minimum matching continuous lines is 0.55, because in our observation the precision will be slashed when $\delta < 0.55$. For medium size blocks with length from 10 to 30 lines, the ratio of ordered common sequences linearly decreases with the smaller blocks length. As big blocks are more likely to be modified in code clone, the lower limit of δ is 0.3 which allows large-variance for big code blocks and ensures certain accuracy.

B. Large-variance Clone Detection

To test the large-variance clone detection ability, we first compared LVMapper with others in eight open source projects dataset. Then we tried to construct synthetic dataset by inserting different number of lines to further evaluate the detecting ability for different variance proportions.

1) Empirical Test: Here we evaluated the large-variance clone detection ability and studied the existence and pervasiveness of large-variance clones. Considering that CCAligner is a good large-gap clone detection method in a very recent study [3], for all the methods in the experiments, we calculate the number of reported clone pairs that satisfied the setting of volume difference $\lambda \leq 0.7$. So we can directly compare with their experimental results for fairness. To validate the *FP* (false positive number), for each projects, we randomly selected 100 samples from our results to validate if they are true clone pairs or not and calculated the false positive rate. Then we used the false positive rate to estimate the *FP*.

In order to compare the detection ability of LVMapper with the state-of-the-art tools, we selected the best two clone detection tools for Type-3 and large-gap clone detection SourcererCC and CCAligner. The results data of SourcererCC and CCAligner were taken from that study straightforwardly. We did not provide the result of NiCad here, because NiCad detected almost none of clones with largely different sizes or variances. For all the experiments using these 8 projects, we considered the clones with minimum length of 10 lines, which is consistent with that of the experiments in paper of CCAligner.

The detecting number of large-variance clones (shorted as *LV*) in 8 projects are shown in Table III. The number of large variance clones detected by LVMapper was markedly more than that detected by SourcererCC and CCAligner. In project JDK1.2.2, the large-variance clones reported by LVMapper are 970 while CCAligner only reported 15 and SourcererCC only reported 4. CCAligner performed better than SourcererCC at detecting the clones with largely different sizes, which was in fact the target of CCAligner. For all projects we tested on, the precisions (which are $1 - \frac{LV}{FP}$) of LVMapper were all over 85%. Among the reported pairs of LVMapper, we found that many clone pairs has scattered modifications and insertions which were missed by CCAligner.

We summarized the number of different types clones and the proportion of *LV* clones detected by LVMapper in Table IV.

TABLE III
LARGE-VARIANCE CLONE EVALUATION RESULTS FOR 8 PROJECTS

Project	LVMapper		CCAligner		SourcererCC	
	LV	FP	LV	FP	LV	FP
JDK 1.2.2	970	87	15	1	4	0
Ant 1.10.1	437	56	87	10	13	0
Maven 3.5.0	382	34	217	30	38	1
Opennlp 1.8.1	2598	78	221	7	5	0
Cook 2.34	760	68	63	2	14	0
Redis 4.0.0	173	26	22	2	7	0
PostgreSQL 6.0	1018	102	219	13	38	0
Linux 1.0	482	53	27	1	12	1

We classified the clone pairs reported by LVMapper to Type-1&Type-2, Type-3 and *LV* clones. As we can see from Table IV, the majority of reported pairs belong to the clones of Type-3. The proportion of the large-variance clones LVMapper reported ranges from 18% to 62% in these projects. There is a high proportion of large-variance clones in open source projects and they should not be overlooked.

TABLE IV
PROPORTION OF LARGE-VARIANCE CLONES DETECTED BY LVMAPPER

Project	Type-1&2	Type-3	All	LV	LV/ALL
JDK 1.2.2	1102	4223	5325	970	18.2%
Ant 1.10.1	114	1420	1534	437	28.5%
Maven 3.5.0	467	1285	1752	382	21.8%
Opennlp 1.8.1	180	4006	4186	2598	62.1%
Cook 2.34	134	1847	1978	760	38.4%
Redis 4.0.0	41	507	536	173	32.3%
PostgreSQL 6.0	123	2141	2238	1018	45.5%
Linux 1.0	119	1379	1493	482	32.3%

2) Large-variance Clone Injection Evaluation: Considering that the real datasets in previous experiments may be unbalanced in data distribution, we further designed the experiment using self-synthetic data to simulate clones with various proportions of insertions and to measure the impacts on the clone detection tools. Given the source code fragment, we tried to insert scattered lines in different quantities and tested the large-variance detection ability for the clone of the file after insertion with the original one.

To evaluate the detection ability of detectors for large-variance clones, we used 200 original code fragments from open source project *jdk 1.8.0* as target code fragments. About one third of them have 15 to 20 lines, one third have 20 to 25 lines, and the remaining part of them have 25 to 30 lines. The number of inserted lines ranges from 1 to 20 lines. For each number of inserting lines, we generated 200 synthetic clones and tested if the tools can detect the clone pairs. We evaluated the state-of-the-art tools CCAligner, SourcererCC, NiCad, iClones with their default configuration. Fig. 3 shows the recall of tools detecting clones for different numbers of inserting lines.

From Fig. 3, we see that when the number of inserting line is 1, all of the detection tools except iClones had the recall of over 95%. The recall fell down with the number of inserting lines increasing. When the inserting line was more than 3, SourcererCC, CCAligner and NiCad descended faster while the performance of LVMapper descended slowly and it maintained the highest recall. For number of inserting lines were smaller than 11 NiCad performed better than CCAligner, while CCAligner showed its advantage when the number of inserting lines continued to increases. At the same time, iClones could hardly detect clones when number of inserting lines was more than 7. After insertion of 16 or more lines, the recall of all the other detection tools was lower than 30% while the recall of LVMapper still remained at above 80%. In this experiment, NiCad showed good performance when the clone pairs have small variance. CCAligner had the advantage in detecting part of the clones that has relatively concentrated gaps. LVMapper performed the best in both small insertions cases and large-variance cases.

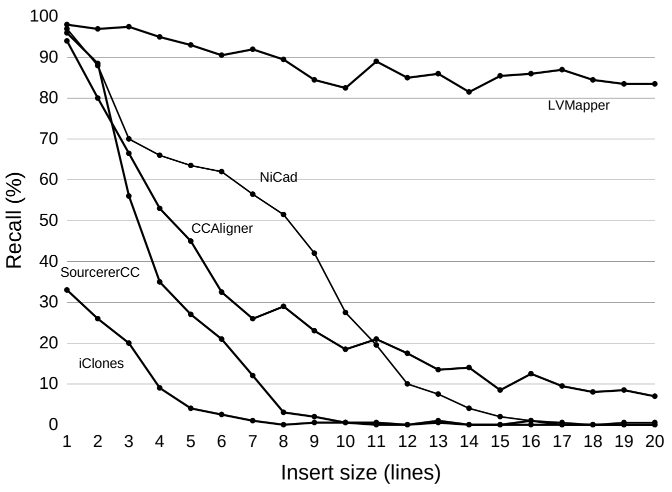


Fig. 3. Recall for different insert sizes.

C. General Clone Detection

Apart from the evaluation of detection ability for large-variance clones above, we also compared the detection performance of LVMapper for general clones (i.e. from Type1 to Type3) with other clone detectors. We used BigCloneEval [22] to test the recall of tools on BigCloneBench [10]. The configuration of NiCad was minimum length 6 lines, similarity threshold 70%, blind renaming and literal abstraction. We used the default configuration of LVMapper, which has seed length of 3-line, and the threshold is described previously. The configuration of SourcererCC was minimum one token and similarity threshold 70%. We set CCAligner with minimum clone size of 6 lines, window size q = 6, edit distance e = 1, and similarity threshold of 60%. The result of Deckard [5], iClones [13] and CCFinderX [12] were from [4]. The number of Weakly Type-3&4 of Deckard was estimated by the recall rate in [4]. As iClones and CCFinderX did not perform well in detecting Moderately Type-3 clones, we did not run them for the performance of Weakly Type-3&4 (denoted as "–" in Table V). We evaluated the clone pairs in BigCloneEval with the setting of considering minimum clone size pretty-printed 6 source lines and minimum clone size 50 tokens. For the measurement of precision, as a common practice in the researches [3], [4], [6], we randomly picked 400 pairs from the reported clones of each tool and manually validated the true clone pairs.

The results for general clone detection performance in BigCloneBench listed in Table V has two parts: the last line is the precision and the rest are the recall. The recall of LVMapper for Type-1, Type-2 and Very Strongly Type-3 were 100% or nearly 100%. For Strongly Type-3, NiCad performed the best with recall of 95% and LVMapper was the second with recall of 81%. With the decreasing in the similarity of the clone pairs, more variance exists within clone pairs. LVMapper had the best performance in Moderately Type-3 with 10% higher than the second best. The number of Weakly Type-3&4 clones LVMapper detected was more than double by that of CCAligner. Although Deckard detected the most pairs in Weakly Type-3&4, it had poor recall for other types of clones and the precision is only 34.8%. LVMapper kept its precision at about 88% while showing excellent detecting capability. Compared with the state-of-the-art clone detectors, LVMapper has good recall and precision for all types of clones.

D. Comparison with semantic method

In the experiments above, the detection ability of LVMapper and other non-semantic tools were tested thoroughly. In order to compare the clone detection ability of LVMapper with semantic method and analyze the difference between them, here we compared LVMapper with the latest machine learning based tool Oreo [6]. Because Oreo only supported clone detection with Java code, we used the same Java projects mentioned in Section IV-B. Oreo was executed with the default configuration.

Table VI shows the *LV* (number of large-variance clones) detected by LVMapper and Oreo for the Java projects and *O/L* is the ratio of *LV* that Oreo could detect among the *LV* detected by LVMapper. The results shows that the large-variance clones detected by Oreo are only a small part of that detected by LVMapper. The main reason is that the semantic clone detector focuses on detecting clones that are almost the same or very similar in semantic. As the semantics of code with more modifications may be changed, the large-variance clones are difficult to be detected by semantic methods.

E. Scalability

To test the scalability of LVMapper, we selected 1M LOC, 10M LOC, 20M LOC and 30M LOC from the inter-project

TABLE V
RECALL PER CLONE TYPE AND PRECISION MEASURED FOR BIGCLONEBENCH

Type	LVMapper	CCAligner	NiCad	SourcererCC	Deckard	iClones	CCFinderX
Type-1	100	100	100	100	60	100	100
Type-2	99	99	100	98	58	82	93
Very Strongly Type-3	98	97	100	93	62	82	62
Strongly Type-3	81	70	95	61	31	24	15
Moderately Type-3	20	10	1	5	12	0	1
# of Weakly Type-3&4	30250	12540	12	1892	77293	–	–
Precision	88	78.8	94.5	98.8	34.8	91	72

TABLE VI
LARGE-VARIANCE CLONES NUMBER FOR 4 JAVA PROJECTS

Project	LVMapper		Oreo		O/L
	LV	All	LV	All	
JDK 1.2.2	970	5325	229	3553	23.6%
Ant 1.10.1	437	1534	143	1941	32.7%
Maven 3.5.0	382	1752	121	1441	31.7%
Opennlp 1.8.1	2598	4186	151	1775	5.8%

Java repository IJaDataset-1.0 [24] as the target files to detect clones. We used a standard desktop with a 3.5GHz quad-core i7-4770k CPU and 24GB of memory. As CCAligner and SourcererCC had relative good scalability in a recent study [3], we compared the execution time of LVMapper with CCAligner and SourcererCC. The minimum lines was set as 6 for LVMapper and CCAligner, and the minimum tokens was set as 50 for SourcererCC.

The execution time across different scales of datasets are shown in Table VII. SourcererCC had less execution time for 1M LOC to 30M LOC and it ran faster than LVMapper on 30M LOC. CCAligner scaled to 10M LOC and it failed for the 20M LOC and 30M LOC inputs with an out of memory error (denoted as "–" in Table VII). LVMapper was the fastest for 1M LOC, 10M LOC and 20M LOC. For 30M LOC, LVMapper was slightly slower than SourcererCC but they were of the same order of magnitude. Although SourcererCC has good scalability but the large-variance detection ability of SourcererCC is limited.

TABLE VII
EXECUTION TIME FOR DIFFERENT LOC

LOC	1M	10M	20M	30M
LVMapper	34s	22m 40s	1h 22m 2s	3h 11m 18s
CCAligner	41s	1h 1m 40s	–	–
SoucererCC	4m 48s	31m 12s	1h 32m 21s	2h 18m 50s

V. RELATED WORK

There are many code clone detection tools proposed in the literature. More descriptions of these tools and methods can be found in [1], [2], [15], [25]–[30]. At present, the code clone detection of Type-3 is still a difficult task, especially for large-variance code clones. According to the types of clone similarity, the clone detection methods can be divided into two categories. One is the non-semantic (lexical and syntactic similarity) method, and the other is the semantic (functional and semantic similarity) method. Our method belongs to the former.

A. Non-semantic Methods

These methods or tools determine whether the code pairs are clones or not base on the similarity of code words and code sentences. These clone detection methods mainly include the text based, the token based, the tree and graph based and the metrics based methods. Among these methods, some researchers classified the latter two as the semantic method.

For the text based tools [14], [31], [32], two code blocks are compared in the form of text or strings. Johnson [31] proposed a fingerprinting technique to identify similar source code and to speed up processing speed. Ducasse [32] developed a line based comparison detection tool. NiCad [14] is based on a two phases process, viz., identification of potential clones and code comparison using longest common subsequences. Compared with LVMapper, it adopts similar locating and verifying strategy. NiCad can detect Type-3 clones, but did not perform well in the test of detection ability for large-variance clones as shown in Fig. 3.

For the token based tools [4], [13], [33], tokens are firstly extracted from the source code by lexical analysis, and it is better than simple keyword matching since it tolerates different identifiers. CCFinder [33] is a popular tool based on token, but it does not support Type-3 clone detection. In their work, they used suffix tree to find identical subsequences and increase the threshold to filter small clones. Essentially, these operations are equivalent to the indexing and filtering technology in LVMapper. iClones [13] and SourcererCC [4] are also influential representatives of such tools. Göde and Koschke [13] developed the incremental tool iClones by merging neighboring Type-1/Type-2 clones to big clones or Type-3 clones. However, iClones can only detect Type-3 clones with small variance. Sajnani [4] developed a fast clone detection tool SourcererCC which uses tokens composition to verify clones, but it is constrained to the identification ability of token granularity. CCAligner [3] has good performance in detecting clones with relatively concentrated modifications but it misses scenarios where modifications are scattered.

For the tree and graph based tools [5], [34]–[37], abstract syntax tree (AST) is frequently used as the representation of source code, and program dependency graph (PDG) is used to represent the control and data flow dependencies in source code. Yang [34] and Deckard [5] proposed AST approaches for finding the syntactic differences between two programs. Duplix [35] and PDG-DUP [36] are PDG-based tools which use program slicing to find isomorphic subgraphs. These tree and graph based tools suffer from large execution times and poor scalabilities. To this end, CCSharp [37] improves the time performance and accuracy of the PDG-based method, but it still cannot achieve good performance in large scale dataset. Besides, these tools will fail to detect large-variance clones since structure of tree and graph may be changed during the extension and modification of the code.

For the metrics based tools [38]–[40], some metrics and characteristic features for tree and graph of source code can be used for code clone detection. Both Mayrand [38] and Balazinska [39] extracted metrics from an AST representation of source code and used the metrics for clone identification. Patenaude [40] used the metrics of source code that can be divided into five categories, viz., classes, coupling, methods, hierarchical structure and clones. These methods extract some features from tree or graph or source code to verify the semantic similarity of two code blocks. They have similar limitations to that of the tree and graph based tools for large-variance clones.

B. Semantic Methods

Apart from the tree and graph based and the metrics based tools mentioned above, these kind of clone detection tools include the semantic space mapping based tools [41], the software behavior based tools [42]–[44] and so on. Substantially, the tools based on semantics adopt semantic abstraction or modeling for source code rather than abstraction of lexical and syntactic similarity. Due to overlap of semantic clones and large-variance clones, however, the methods based on semantics can also find a small part of large-variance clones shown as Table VI.

Machine learning is always an effective way to deal with complex problems, including code clone detection especially for semantic clone. With the spread of deep learning method, the clone detection using deep learning technologies is an emerging area. White [45] presented an unsupervised deep learning approach to detect clones, which can automatically learn discriminating features of source code. Wei [46] proposed a method to detect clones by learning representations and Hamming distance of code fragments. Zhao [47] encoded code control flow and data flow into a semantic matrix for detecting semantic clones. Recently, Saini [6] put forwarded a machine learning based method called Oreo, which can find the code clones in the overlap between syntactic and semantic zone. Oreo used the clones that are almost identical or very similar to train the deep learning model and the large-variance clone detection ability of Oreo is limited. Machine learning methods always face the issues of efficiency and dependency on the initial training data. The experiment in Section IV-D shows that the machine learning method could not detect the large-variance clones well, which means the modifications between clones may change the semantics of code.

VI. Limitation

Our tool is aimed at detecting the large-variance clones with *homologous modification*. As shown in Fig. 3 for the large-variance clone injection evaluation, the large-variance clone detection ability of LVMapper is significantly better than others. Note that this is the experiment of large variance clone detection just for syntactic similarity and our method did not detect semantic clones. In the BigCloneBench experiment shown as Table V, our tools have made better progress in Moderately Type-3 and Weakly Type-3&4, but it has not yet reached a satisfactory level. The reason is probably that Moderately Type-3 and Weakly Type-3&4 clones are more of semantic clones that are generated by *heterologous development*.

The scalability of LVMapper on larger scale dataset needs to be tested. LVMapper on 30M LOC shows satisfactory performance (3h) in the evaluation. It is meaningful to test it on over 100M LOC for validating its scalability further.

VII. Conclusion & Future Work

The large-variance code clones are generated by homologous modification and can be used in software development and other applications. Our experiments found that these clones were widespread in Type-3 clones, even in some datasets up to half or more. And the large-variance code clone changes the past clone detection methods that focus on finding almost identical or very similar code pairs. Therefore, The research on large-variance code clone is important and meaningful. In this paper, we proposed a novel concrete definition and a detector LVMapper borrowing from the idea of sequencing alignment in bioinformatics for large-variance code clones. Effective and innovative technologies such as dynamic threshold, avoiding time-consuming dynamic programming and seeds index are designed in our method. A series of testing on real cases of software projects, self-synthetic programs and the state-of-the-art benchmarks showed the large-variance clone detection performances of LVMapper are much better than the other state-of-the-art tools, and it has good recall and precision for general Type-1 to Type-3 clones. Furthermore, we will make LVMapper more scalable for the clone detection on larger scale datasets. And it will be important work to do research on software engineering applications such as code recommendation and completion, refactoring and bug propagation for large-variance clones in the future.

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, pp. 64–68, 2007.

[2] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[3] 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 (ICSE'18)*. ACM, 2018, pp. 1066–1077.

[4] 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 IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. IEEE, 2016, pp. 1157–1168.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[6] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, 2018, pp. 354–365.

[7] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[8] B. Liu, D. Guan, M. Teng, and Y. Wang, "rhat: fast alignment of noisy long reads with regional hashing," *Bioinformatics*, vol. 32, no. 11, pp. 1625–1631, 2015.

[9] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[10] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 2014, pp. 476–480.

[11] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, 2015, pp. 131–140.

[12] T. Kamiya, "The official ccfinderx website," *http://www.ccfinder.net/ccfinderx.html*, 2008.

[13] N. Göde and R. Koschke, "Incremental clone detection," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. IEEE, 2009, pp. 219–228.

[14] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*. IEEE, 2008, pp. 172–181.

[15] 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.

[16] P. Jaccard, "The distribution of the flora in the alpine zone," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[17] H. Cheng, H. Jiang, J. Yang, Y. Xu, and Y. Shang, "Bitmapper: an efficient all-mapper based on bit-vector computing," *BMC Bioinformatics*, vol. 16, no. 1, p. 192, 2015.

[18] J. R. Cordy, "The txl source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[19] V. Paxson *et al.*, "Flex–fast lexical analyzer generator," *Lawrence Berkeley Laboratory*, 1995.

[20] A. Appleby. (2016) Murmurhash hash functions. [Online]. Available: https://github.com/aappleby/smhasher/

[21] A. S. E. Group. (2013) Ijadataset 2.0. [Online]. Available: http://secold.org/projects/seclone

[22] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. IEEE, 2016, pp. 596–600.

[23] Cloc. (2015) Count lines of code. [Online]. Available: http://cloc.sourceforge.net/

[24] A. S. E. Group. (2011) Ijadataset 1.0. [Online]. Available: http://secold.org/projects/seclone

[25] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2002, pp. 36–43.

[26] 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, pp. 470–495, 2009.

[27] F. V. Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society, 2004, pp. 336–339.

[28] F. Arcelli Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti, "Software clone detection and refactoring," *ISRN Software Engineering*, vol. 2013, 2013.

[29] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[30] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, vol. 137, no. 10, pp. 1–21, 2016.

[31] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proceedings of International Conference on Software Maintenance*, vol. 94, 1994, pp. 120–126.

[32] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of International Conference on Software Maintenance*. IEEE, 1999, pp. 109–118.

[33] 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, pp. 654–670, 2002.

[34] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[35] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE, 2001, pp. 301–309.

[36] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the International Static Analysis Symposium*. Springer, 2001, pp. 40–56.

[37] M. Wang, P. Wang, and Y. Xu, "Ccsharp: An efficient three-phase code clone detector using modified pdgs," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. IEEE, 2017, pp. 100–109.

[38] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the International Conference on Software Maintenance*, vol. 96, 1996, p. 244.

[39] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Proceedings of the 6th International Software Metrics Symposium*. IEEE, 1999, pp. 292–303.

[40] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë, "Extending software quality assessment techniques to java systems," in *Proceedings of the 7th International Workshop on Program Comprehension*. IEEE, 1999, pp. 49–56.

[41] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE'01)*. IEEE, 2001, pp. 107–114.

[42] S. Choi, H. Park, H.-i. Lim, and T. Han, "A static api birthmark for windows binary executables," *Journal of Systems and Software*, vol. 82, no. 5, pp. 862–873, 2009.

[43] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*. ACM, 2009, pp. 81–92.

[44] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 301–310.

[45] 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*. ACM, 2016, pp. 87–98.

[46] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017, pp. 3034–3040.

[47] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 141–151.