CCAligner: a token based large-gap clone detector

Pengcheng Wang[*]
University of Science and Technology of China
School of Computer Science
wpc520@mail.ustc.edu.cn

Jeffrey Svajlenko
University of Saskatchewan
Department of Computer Science
Canada
jeff.svajlenko@gmail.com

Yanzhao Wu
University of Science and Technology of China
School of Computer Science
wuyanzha@mail.ustc.edu.cn

Yun Xu[†‡]
University of Science and Technology of China
School of Computer Science
xuyun@ustc.edu.cn

Chanchal K. Roy
University of Saskatchewan
Department of Computer Science
Canada
croy@cs.usask.ca

ABSTRACT

Copying code and then pasting with large number of edits is a common activity in software development, and the pasted code is a kind of complicated Type-3 clone. Due to large number of edits, we consider the clone as a large-gap clone. Large-gap clone can reflect the extension of code, such as change and improvement. The existing state-of-the-art clone detectors suffer from several limitations in detecting large-gap clones. In this paper, we propose a tool, CCAligner, using code window that considers *e* edit distance for matching to detect large-gap clones. In our approach, a novel *e*-mismatch index is designed and the asymmetric similarity coefficient is used for similarity measure. We thoroughly evaluate CCAligner both for large-gap clone detection, and for general Type-1, Type-2 and Type-3 clone detection. The results show that CCAligner performs better than other competing tools in large-gap clone detection, and has the best execution time for 10MLOC input with good precision and recall in general Type-1 to Type-3 clone detection. Compared with existing state-of-the-art tools, CCAligner is the best performing large-gap clone detection tool, and remains competitive with the best clone detectors in general Type-1, Type-2 and Type-3 clone detection.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Clone Detection, Large-gap Clone, Evaluation

[*]Also with Key Laboratory on High Performance Computing, Anhui Province.
[†]Yun Xu is the corresponding author.
[‡]Also with Key Laboratory on High Performance Computing, Anhui Province.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180179

ACM Reference Format:
Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: a token based large-gap clone detector. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180179

1 INTRODUCTION

Reusing code via copying and pasting is a frequent activity in software development. The copied code is known as a *code clone*. Code clones may lead to software maintenance problems [29, 32, 51] and bug propagation [21, 37, 57]. Therefore, clone detection is useful for a variety of tasks (e.g., refactoring [31, 52], debugging [17, 19], software evolution study [5, 22], and software management [34, 44]). Copied code with further modifications like statements insertion/deletion in addition to changes in identifiers is called *Type-3 clone*. Ueda et al. [54] call such clones as *gapped code clones*, and thus we further call Type-3 clones with a large number of edits (i.e., statements insertion/deletion) as *large-gap clones*. We focus on the study of large-gap clones detection as well as all Type-1 to Type-3 clones detection in large code bases.

Large-gap clones can reflect difference between two similar code, corresponding to the extension (e.g., change or improvement) of code. Copying code with many statements insertion/deletion is a common behavior in software development. Fig. 1 shows an example of large-gap clones found in project *Ant 1.10.1*. Nearly the same size of the original statements are inserted (10/12), and lines 4-13 in Fig. 1b are large gaps which reflect the extension from the previous `run` program to the new `run` program. In addition to intra-project clones, inter-project clones may contain large-gap clones, like clones in different versions of the software or different software developed for similar applications. Hence, it is important to find the large-gap clones.

Numerous tools have been developed for clone detection [38]. According to different source representations, most clone detectors can be classified into six classes: text based [18, 40], token based [8, 27], tree based [16, 56], PDG based [24, 26], metrics based [30, 36], and hybrid approaches [11, 15]. However, majority of text and token based detectors cannot detect Type-3 clones. Although tree and PDG based techniques can support the detection of Type-3 clones, tools based on these approaches suffer from large execution times.

```
1  protected int run (Commandline cmd) {
2      try {
3          Execute exe = new Execute (new LogStreamHandler (this,
           Project.MSG_INFO, Project.MSG_WARN));
4          exe.setAntRun (getProject ());
5          exe.setWorkingDirectory (getProject ().getBaseDir ());
6          exe.setCommandline (cmd.getCommandline ());
7          exe.setVMLauncher (false);
8          return exe.execute ();
9      } catch (java.io.IOException e) {
10         throw new BuildException (e, getLocation ());
11     }
12 }
```

(a) Original code

```
1    private int run (Commandline cmd) {
2        try {
3            Execute exe = new Execute (new LogStreamHandler (this,
             Project.MSG_INFO, Project.MSG_WARN));
4  +         if (serverPath != null) {
5  +             String [] env = exe.getEnvironment ();
6  +             if (env == null) {
7  +                 env = new String [0];
8  +             }
9  +             String [] newEnv = new String [env.length + 1];
10 +             System.arraycopy (env, 0, newEnv, 0, env.length);
11 +             newEnv [env.length] = "SSDIR=" + serverPath;
12 +             exe.setEnvironment (newEnv);
13 +         }
14           exe.setAntRun (getProject ());
15           exe.setWorkingDirectory (getProject ().getBaseDir ());
16           exe.setCommandline (cmd.getCommandline ());
17           exe.setVMLauncher (false);
18           return exe.execute ();
19       } catch (IOException e) {
20           throw new BuildException (e, getLocation ());
21       }
22   }
```

(b) Clone with many statements insertion

Figure 1: Example of Large-gap Clone in Project *Ant 1.10.1*.

Existing studies have shown that there are more Type-3 clones in the software systems than other types [42, 46]. Therefore, Type-3 clones can be the most needed in code clone detection. Moreover, large-gap clones are complicated Type-3 clones, and the state-of-the-art tools suffer from several limitations. For example, CCFinderX [20] has good scalability and execution time, but it only supports Type 1-2 clone detection. iClones [14] can only detect Type-3 clones with small gaps and the method is not very scalable. Although Deckard [16] can detect Type-3 clones, its precision and recall are poor [45]. NiCad [40] can detect Type-3 clones effectively, but it is limited by large execution times and fails to detect when scales to large code bases. SourcererCC [45] measures overlap similarity of pairs of code blocks at token-level granularity to detect Type-3 clones, but the similarity of large-gap clones that have large difference in token is low. Therefore, SourcererCC needs to set a very low similarity threshold for large-gap clone detection, which could extremely hurt precision.

In order to support fast and accurate detection of Type-3 clones, especially the large-gap clones, we propose a technique and implement it as a tool, named CCAligner. Token based detection techniques have good scalability and execution time, and thus CCAligner uses tokenization to normalize for Type-1/Type-2 variations. In particular, CCAligner considers sliding code windows (i.e., continuous code fragments), instead of tokens, as basic unit for matching. Compared to a single token, code window captures the local sequence characteristics of source code (i.e., localness of software [53]) and can improve matching accuracy. To further enhance the detection capability of clone gaps, we consider pairs of code windows match with e-mismatch, instead of exactly matching. Furthermore, CCAligner uses asymmetric similarity coefficient as similarity function, which is more suitable for measuring the clone with large gaps, since this similarity measure is robust even if two code blocks have large difference in size. Our tool is needed to detect/evaluate the large-gap clones previously missed in clone studies. Without such tools, developers cannot mitigate the risks caused by them.

Our experiments show that CCAligner performs best in the detection of large-gap clones, and has a good performance in detection of all Type-1, Type-2 and Type-3 clones. To evaluate the ability of large-gap clone detection, we conduct an empirical study with eight subject systems and also compare with NiCad and SourcererCC, the best performing gapped clone detectors from the literature [45]. Furthermore, to see the extent of large-gap clone detection performance of CCAligner, we make use of the mutation-injection based approach, where we adapt an established benchmark (Roy and Cordy [41], and Svajlenko et al [50]) for syntactically creating gapped clone of different sizes, to evaluate and compare CCAligner with state-of-the-art gapped clone detection tools including NiCad [40], SourcererCC [45], iClones [14] and Deckard [16] for recall. Our study shows that CCAligner is the best performing large-gap clone detection tool to date. Moreover, we compare CCAligner with different state-of-the-art tools for Type-1, Type-2 and Type-3 clone detection, and the results show that CCAligner is fast and can scale to 10MLOC inputs with good precision and recall.

The major contributions of this paper include:

(1) We show that using code windows that consider e edit distance for matching is effective to detect large-gap clones. We design a novel e-mismatch index and use asymmetric similarity function in implementation. We experimentally demonstrate the best parameterization to detect clones.
(2) We implement the proposed techniques as a tool, CCAligner. We show the effectiveness of the proposed techniques by comprehensively evaluating CCAligner.
(3) The evaluations demonstrate that CCAligner is the best performing large-gap clone detection tool, and remains competitive with the best clone detectors in general Type-1, Type-2 and Type-3 clone detection.

The remainder of this paper is structured as follows. Section 2 describes some concepts about code clone, and gives definition of large-gap clones. Section 3 presents the detailed process of clone detection. Section 4 evaluates our tool both for detection of clones with large gaps, and for general clone detection. Section 5 surveys related work and Section 6 discusses our limitations. The paper concludes with discussion and future work in Section 7.

2 PRELIMINARY DEFINITION

In this section, we first introduce concepts and definition regarding code clones. We then propose the definition of Type-3 clones with edit distance measure. Furthermore, we present a formal definition of large-gap clones.

Code fragment is a continuous segment of source code, and *code block* can be a function or a sequence of statements within braces. *Clones* are code pairs or groups that have the same or similar fragment. Let $u_i^k = (i, s_k, e_k)$ be a code fragment, where i is the file id, s_k and e_k representing the start and end line of the kth fragment in this file, respectively. To define more precisely, a *clone pair* is a triple (u_i^k, u_j^l, t) where code fragments u_i^k and u_j^l are two similar code fragments and t is the clone type (Type-1, 2, 3 or 4). A *clone group* (or *clone class*) is a set of similar fragments, defined as the tuple $(u_i^k, u_j^l, ..., u_h^g, t)$ where each pair of code fragments is a clone pair. In addition, *Min clone size* is the minimum lines or tokens in length that could be seen as clone. Generally, 6 lines and 50 tokens in length or greater are the standard size for detecting [7, 39].

Roy et al. [39] and Bellon et al. [7] classify clones into four types: *Type-1* clones are identical code fragments except for variations in white space, comments and layout. *Type-2* clones are identical code fragments except for variations in identifiers, literals, and variable types, in addition to Type-1 clone variations. *Type-3* clones are copied fragments with further modifications such as added, deleted or changed statements, in addition to Type-2 clone variations. *Type-4* clones are code fragments that perform similar functionality but are implemented by different syntactic variants.

While Type-1 and Type-2 clones are precisely defined and form an equivalence relationship, there is no consensus on a suitable similarity measure for Type-3 clones. Type-3 clones are typically defined with respect to line/statement-level edits, and thus we apply widely-adopted edit distance [33] measure to Type-3 clone definition.

Definition 2.1. Edit Distance: The edit distance between two code fragments u_i^k and u_j^l, denoted by $dist(u_i^k, u_j^l)$, is the minimum number of single-line insertion, deletion, and substitution that are needed to transform u_i^k to u_j^l.

As an example, two code blocks in Fig. 1 are Type-3 clones with $e = 10$, corresponding to 10 lines insertion (lines 4-13 in Fig. 1b).

Unmapped statements within the clone fragments are known as *clone gaps* (e.g., lines 4-13 in Fig. 1b). When the proportion of clone gaps is large, we consider such clones as *large-gap clones*. In other words, pairs of code fragments are large-gap clones if satisfy: (1) they are Type-3 clones and (2) they have large difference in size. The quantitative definition of large-gap clones are as follows.

Definition 2.2. Large-gap clones: Given two code blocks u_i^k and u_j^l, assume their number of pretty-printed lines are L_i and L_j (assume $L_i \leq L_j$), respectively. Let $\lambda = L_i/L_j$, where λ is the ratio of code length, in terms of pretty-printed lines. If u_i^k and u_j^l are Type-3 clones and $\lambda \leq 0.7$, then they are large-gap clones.

We identify large-gap clones via a scale-difference value λ, and we next explain why selecting the lower bound as 0.7 for quantitative definition.

Let B_1 be the original code block whose code length is α and B_2 be the large-gap clone of B_1 with $\alpha/2$ lines insertion (i.e., the code length of B_2 is $3\alpha/2$). It means when clone gaps are half size of the original code, $\lambda = 2/3 \approx 0.7$, according to our definition. Therefore, we identify large-gap clones if the clone gaps are nearly half the size of the original code ($\lambda = 0.7$) or larger ($\lambda < 0.7$). According to Definition 2.2, two code blocks in Fig. 1 are large-gap clones since (1) they are Type-3 clones and (2) $\lambda = 12/22 \approx 0.55$ falls in the range of 0-0.7.

3 APPROACH

The entire process of our approach is summarized in Fig. 2. It can be considered in two phases: lexical analysis and clone detection. The following subsections describe the design of each phase.

3.1 Lexical Analysis

In the lexical analysis phase, code blocks are first extracted from the source files. Pretty-printing is used to layout the tokens for one statement per line. Tokenization is used to normalize for Type-1 and Type-2 variations.

In order to reduce the redundancy of code and constitute the object of clone detection, code blocks are first extracted from source files and then pretty-printed using TXL [10], as of many state-of-the-art approaches (e.g., SourcererCC [45]). After this step, a set of pretty-printed code blocks is produced. We then tokenize each code block by converting each item (such as keywords, operators, etc) in the code block to the corresponding token. In order to tolerate identifier renaming (i.e., Type-2), identifiers (e.g., variables and functions) are mapped into the same token *id*. Our lexical analyzer is mainly based on a scanner generated by Flex [35]. This scanner can parse a single code block to ordered elements. The original code blocks are transformed into normalized and tokenized code blocks, as the input of following clone detection.

3.2 Clone Detection

Given a set of tokenized code blocks, where Type-1/Type-2 variations have been eliminated, we then implement sliding windows which tolerate e edit distance across these blocks. We evaluate pairs of code blocks are clones by measuring their ratio of matched windows (considering edit distance e), and report those satisfying the similarity threshold.

Algorithm 1 describes the steps in detail. It works in two stages: (1) Matching via index and generating candidate set (lines 1-30); and (2) Verifying the candidate and reporting clone pairs (lines 32-40). Each step is described as follows.

3.2.1 Index and Match. Compared to a single token, code window can capture the local sequence characteristics of source code. Hence, CCAligner considers code windows (i.e., continuous code fragments) as basic unit for matching. It uses sliding windows to break the blocks. Since the windows are overlapping, all code fragments of the window size must be located in a window.

To further enhance the detection capability of clone gaps, we consider pairs of code windows match with e-mismatch, instead of exactly matching. For example, assume the window size is 6 and $e = 1$, given two code windows $W_a = w(3, 8)$ and $W_b = w'(13, 18)$ in Fig. 1, where $w(3, 8)$ means the code window containing lines 3-8

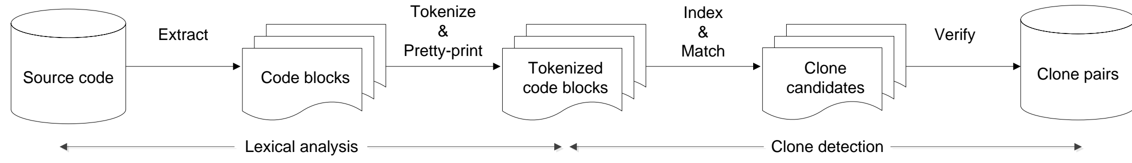


Figure 2: The Overview of Our Approach.

Algorithm 1: Clone detection

```
   Input: F is set of tokenized code blocks {f1, f2, ..., fn}, q is the window size, e is edit
          distance threshold, and θ is the similarity threshold; q, e and θ are specified by the
          user
   Output: All clone pairs
 1 candMap = φ;
 2 hashSet = φ;
 3 for each block f in F do
 4 |   L=Num of lines in block f;
 5 |   numWin_id =L-q+1;
 6 |   for i = 1 → L-q+1 do
 7 |   |   T_i = token statement of ith line;
 8 |   |   for j = 1 → C_q^e do
 9 |   |   |   h = diffCombination(T_i, T_{i+1}, ..., T_{i+q-1});//h is different
   |   |   |     (q-e)-grams;
10 |   |   |   k = Hash(h);
11 |   |   |   v = f_id;
12 |   |   |   hashSubSet.insert(k, i);
13 |   |   |   if candMap.find(k) then
14 |   |   |   |   candMap(k).value = candMap(k).value ∪ v;
15 |   |   |   else
16 |   |   |   |   candMap.insert(k, v);
17 |   |   |   end
18 |   |   end
19 |   end
20 |   hashSet(f_id).insert(hashSubSet);
21 |   hashSubSet = φ;
22 end
23 candPair = φ;
24 for each map in candMap do
25 |   if NumOfValue >= 2 then
26 |   |   candMap.value = sort(candMap.value, f_id);
27 |   |   candPair = candPair ∪ each pair in candMap.value;
28 |   end
29 end
30 Remove duplicate element in candPair;
31
32 for each pair (f_m, f_n) in candPair do
33 |   setIntersection(hashSet(f_m), hashSet(f_n));
34 |   numMatch1 = matched num of windows in f_m;
35 |   numMatch2 = matched num of windows in f_n;
36 |   if numMatch1 >= θ · numWin_m or numMatch2 >= θ · numWin_n then
37 |   |   clonePair = clonePair ∪ (f_m, f_n);
38 |   end
39 end
40 return clonePair;
```

code in Fig. 1a and $w'(13, 18)$ means the code window containing lines 13-18 code in Fig. 1b, we consider W_a and W_b are matched with 1-mismatch.

Assume q is the window size (i.e., there are q lines code in a window) and e is edit distance threshold. We further use an effective method to guarantee that each window just needs to be processed once by designing the e-mismatch index, which avoids using time-consuming dynamic programming to verify the edit distance between each pair of windows. When CCAligner scans a window, it generates all *(q-e)*-grams[1] of this window and builds an index mapping each *(q-e)*-gram to corresponding code block and window. Next, for each sliding window, its *(q-e)*-grams are generated and the index is updated dynamically.

Detailed steps are shown in Algorithm 1. Assume the number of lines in block f is L, then there will be *L-q+1* code windows (line 5), where q is the window size. Each code window is a set of q lines token statements $\{T_i, T_{i+1}, ..., T_{i+q-1}\}$, where T_i is the token statement of ith line. Define each code window as a q-gram, then CCAligner builds e-mismatch index and finishes matching (lines 8-18). When two code windows (q-grams) can be matched under e edit-distance, their *(q-e)*-grams should be matched in the index. Formally, it can be stated in the form of the following theorem (Its proof is available online[2], due to limited space):

THEOREM 3.1. *Given code windows A and B consisting of q token statements (the length of A and B is q), if A and B can be matched with maximum edit distance of e, then A and B must have at least one matching subsequence with minimum length of q-e.*

To understand this theorem, we give an example in Fig. 3. Assume the window size is 6 ($q = 6$), and let edit distance be specified as $e = 1$, then two code windows W_1 and W_2 could be seen as two 6-grams. After the W_1 is processed, all its 5-grams are updated to the index. W_2 is the code window (g, b, c, d, e, f), and when its all 5-grams are updated to the index, we can find a match with W_1 by *bcdef*.

Code window	W1=(a,b,c,d,e,f)	W2=(g,b,c,d,e,f)
Index (e=1)	**bcdef**	**bcdef** (matched)
	acdef	gcdef
	abdef	gbdef
	abcef	gbcef
	abcdf	gbcdf
	abcde	gbcde

Figure 3: Example for Theorem 3.1.

According to the Theorem, in order to find out if code windows A and B are matched, we only need to check if their *(q-e)*-grams have at least one match. Therefore, for each sliding window, CCAligner extracts all the different *(q-e)*-grams of the q-grams, and it builds an inverted index (k, v) mapping *(q-e)*-grams to the code blocks containing them (lines 13-17). Then the code blocks containing similar code windows will be grouped (line 14).

To improve efficiency, a hash value is computed on *(q-e)*-grams sequence using the MurmurHash hash function[2], chosen for its low collision rate. Besides, CCAligner saves a set of tuples ⟨*hashVal*, *winId*⟩ for each code block (line 20), which is used for the next

[1] Assume code window $W_1 = (a, b, c, d)$, where $q = 4$. When we set $e = 1$, its all 3-grams are bcd, acd, abd and abc.

[2] https://goo.gl/qDduYQ

step to verify the candidates. Overall, when the cutting window is sliding, CCAligner builds the index along with matching. The whole process finishes when the sliding window scans all code blocks. Since the code blocks containing similar code windows have been merged in the *value* of *candMap(k)*, each pair in *candMap(k).value* is a clone candidate (lines 24-30 in Algorithm1).

3.2.2 Verify. This step is to verify whether the candidates are clones. For each pair $\langle f_n, f_m \rangle$ in *candPair*, code blocks f_n and f_m contain at least one matched code window with e edit distance. In order to further measure the similarity of two code blocks, we use *asymmetric Dice similarity coefficient* [3] as similarity function, defined as follows:

Definition 3.2. Asymmetric Dice similarity coefficient:

$$\mathrm{sim}(f_m, f_n) = \frac{|W_{f_m} \cap W_{f_n}|}{\min(|W_{f_m}|, |W_{f_n}|)},$$

where W_{f_m} is the set of code windows contained in the code block f_m, W_{f_n} is the set of code windows contained in the code block f_n and the min function normalizes the similarity score with respect to the number of windows contained in the smaller code block.

By calculating an intersection between *hashSet(fn)* and *hashSet(fm)*, then the number of matched code windows (considering e edit distance) is obtained. If the $\mathrm{sim}(f_m, f_n)$ satisfies θ specified by user, then CCAligner reports $\langle f_n, f_m \rangle$ as a clone pair (lines 32-40).

We use the asymmetric Dice coefficient instead of other similarity measures (e.g., the Jaccard coefficient) because large-gap clones have a large difference in size. Therefore, considering the minimum cardinality of the sets of code windows at the denominator of the formula allows to weigh the similarity between code blocks better.

Consider the code in Fig. 1 as an example to show how CCAligner enables large-gap clone detection. These two code blocks have been pretty printed, and we could directly take a look at the source code, since their corresponding tokens are the same if the code blocks are the same. In clone detection phase, assume the window size is 6 and each window tolerates 1 mismatch (i.e., $q = 6$ and $e = 1$), then for each window it will produce six 5-grams for matching. If there exists one 5-grams match between two windows, these two windows are considered as a match. Code block in Fig. 1a will produce 7 sliding code windows each containing 6 lines of code, and code block in Fig. 1b will produce 17 sliding code windows. Consider $WinSet(f_a) = \{w(1,6), w(2,7), \ldots w(7,12)\}$ for code block in Fig. 1a, where $w(1,6)$ means the the first code window containing first 6 lines of code, and $WinSet(f_b) = \{w'(1,6), w'(2,7), \ldots w'(17,22)\}$ for code block in Fig. 1b. Finally, 5 windows in $WinSet(f_a)$ match with the windows in $WinSet(f_b)$. Note that $w(3,8)$ can match with $w'(13,18)$, due to our 1-mismatch strategy. According to our similarity measure, $\mathrm{sim}(f_a, f_b) = 5/7 \approx 0.71$, so this pair of large-gap clones can be detected setting similarity threshold as 70%.

However, some existing state-of-the-art tools suffer from several limitations in detection of the large-gap clone in Fig. 1. For example, NiCad [40] will miss this clone since it uses LCS similarity for Type-3 and the LCS similarity of these two blocks is very low. SourcererCC [45] will miss such clone unless a low similarity threshold (50%) is set, which could extremely hurt the performance. Besides, we run line-based detector iClones [14], and it even fails to report the similar regions before and after the gap.

Table 1: Recall per Clone Type and Precision Measured for BigCloneBench with Different Parameterization

q	e	Recall					Precision
		T1	T2	VST3	ST3	MT3	
6	0	100	99	89	31	1	93
	1	100	99	97	70	10	80
	2	100	99	99	80	24	61
7	0	100	99	82	15	1	94
	1	100	99	97	59	6	82
	2	100	99	98	77	16	77
8	0	100	99	78	9	1	96
	1	100	99	97	51	4	83
	2	100	99	98	71	11	82

4 EVALUATION

In this section we thoroughly evaluate CCAligner both for large-gap clone detection, and for general Type-1, Type-2 and Type-3 clone detection. We begin by demonstrating experimentally that $q = 6$ and $e = 1$ work best for clone detection which balance recall with precision and performance.

4.1 Parameter Setting

CCAligner implements sliding windows which tolerate e edit distance across code blocks, and evaluates pairs of code blocks are clones by measuring their ratio of matched windows (considering edit distance e). Hence, the choice of window size q and edit distance threshold e will affect the performance of clone detection.

To find the most suitable parameterization for detecting, we try various combinations of q, e to evaluate CCAligner with BigCloneBench [46, 48] using the BigCloneEval [49] framework for minimum clone size of 6 lines, and a similarity threshold of 60%. BigCloneBench is a large benchmark of manually validated clones in a large inter-project Java repository. BigCloneEval allows the user to conduct custom recall measurement experiments on top of BigCloneBench, and automatically handles aspects such as tool execution and recall analysis [49].

Since 6 lines in length or greater are the standard size for detecting [7, 39], and when e is large, the accuracy of the match would be low. Hence, we select $q \geq 6$ and e as a small number. In our experiments, we try various combinations of $q = 6, 7, 8$ and $e = 0, 1, 2$. When $e = 0$, pairs of code windows are exactly matched. Recall is summarized per clone type, as per the BigCloneBench definitions [48]. Specifically, BigCloneBench splits the Type-3 clones into multiple categorizes by their syntactical similarity. Very-Strongly Type-3 (VST3) clones are those that are 90-100% similar by syntax, Strongly Type-3 (ST3) clones are 70-90% similar, and Moderately Type-3 (MT3) clones are 50-70% similar. We measure precision by manually validating a random sample of the clones detected by CCAligner during the BigCloneBench experiments. We randomly select 200 clones for each parameterization to validate (1,800 clone pairs in total). The recall is reported by the BigCloneEval framework.

Detailed results are summarized in Table 1. We can see that CCAligner has perfect or near-perfect Type-1 and Type-2 recall even $e = 0$, since the Type-1/Type-2 variations has been normalized in tokenization. For the same q, when e is larger, the recall of Type-3

(i.e., VST3, ST3, and MT3) gets better but the precision gets worse, because allowing more mismatch in windows will enhance the detection capability of clone gaps but cause more false positive. Although the recall is the best when $q = 6$ and $e = 2$, the precision is rather poor. We can see that $\langle q = 6, e = 1\rangle$, $\langle q = 7, e = 2\rangle$, and $\langle q = 8, e = 2\rangle$ work better for clone detection which balance recall with precision.

Table 2: Execution Time and Memory Space with Different Parameterization for Linux 4.8.12

Configuration	q,e	6,1	7,2	8,2
Performance	Time	13m 2s	32m 37s	28m 16s
	Space	3.4 GB	12.2 GB	11.5 GB

We next evaluate the execution time of CCAligner with these three configurations to justify the best choice for detection. To obviously show the difference of performance, we use large codebase, Linux kernel 4.8.12, as our target. The source code contains 23424 code files with 11,505,767LOC (i.e., line of code), measured by tool cloc [9]. The results are in Table 2. We can see that CCAligner needs less time and memory space for detection when $q = 6$ and $e = 1$. Therefore, $q = 6$ and $e = 1$ are the most suitable configurations for detecting which balance recall with precision and performance. We thus set $q = 6$ and $e = 1$ for the next clone detection evaluation.

4.2 Large-gap Clone Detection

After experimentally determining the parameters (i.e., q and e) used for clone detection, we now demonstrate the ability of our CCAligner for large-gap clone detection. We first conduct an empirical study with eight subject systems and compare with state-of-the-art gapped clone detectors in terms of precision, recall and F1-score. Furthermore, to see the extent of large-gap clone detection performance of CCAligner, we develop a variant of an established syntactic benchmarking framework, the Mutation-Injection based framework [41, 50], to evaluate and compare the recall of CCAligner for different gap sizes.

4.2.1 ***Empirical Study.*** We select a total of eight C and Java open source projects as our dataset for evaluation. The size of the source code of the projects varies from 43KLOC to 138KLOC. The detailed statistics of subject systems can be seen in Table 3, measured by tool cloc [9].

Table 3: Subject System's Statistics

System	Language	Files	LOC
Cook 2.34	C	296	43,900
Redis 4.0.0	C	213	85,664
PostgreSQL 6.0	C	339	94,087
Linux 1.0	C	282	103,677
JDK 1.2.2	Java	115	17,140
OpenNLP 1.8.1	Java	903	66,291
Maven 3.5.0	Java	952	79,840
Ant 1.10.1	Java	1223	138,505

From the literature [45], NiCad and SourcererCC are the best performing gapped clone detectors. Hence, we compare CCAligner against high-recall tools NiCad and SourcererCC to demonstrate that CCAligner can detect large-gap clones that the best of the competing tools are missing.

We first detect code clones in these projects using CCAligner, NiCad and SourcererCC, all with the configurations of min length 10 lines, min similarity 70%. We define

$$\text{Recall} = \frac{LG - FP}{\text{Union of } TP},$$

where *LG-FP* is the true positive large-gap clones reported by one tool, and denominator is the union of true positive large-gap clones (removing duplicate elements) reported by different tools. We also define

$$\text{F1-score} = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}.$$

After the clone results are obtained, we measure precision by manually validating a random sample of the clones detected by different tools (100 for each). To measure the recall of large-gap detection, we choose the union of true positive large-gap clones reported by different tools as the reference corpus. To create the large-gap recall benchmark, we select the clones satisfying our Definition 2.2 (i.e., large-gap clones) by calculating the scale-difference value λ. If $\lambda \leq 0.7$, we consider this clone pair as a large-gap clone. Furthermore, we manually validate all detected large-gap clones, and remove the false positive clones.

Table 4 shows the detailed values of precision, recall, and F1-score for each tool in detection of C projects. *LG* in Table 4 is the number of detected large-gap clones, and *FP* is the number of false positive clones among these large-gap clones. Compared to NiCad and SourcererCC, our CCAligner has the best recall and F1-score in each project. SourcererCC and NiCad miss many large-gap clones that CCAligner detects, reflected as much lower recall. The total recall and F1-score of CCAligner in C projects are the best, whereas the other tools are much lower. Finally, a manual validation identifies 377 true positive large-gap clones in these C projects.

Table 4: Large-gap Clone Evaluation Results for C.

System	Tool	LG	FP	Precision	Recall	F1-Score
Cook 2.34	NiCad	0	0	84	0	0
	SourcererCC	14	0	89	19	31
	CCAligner	63	2	86	81	**83**
Redis 4.0.0	NiCad	1	0	87	4	8
	SourcererCC	7	0	90	27	42
	CCAligner	22	2	88	77	**82**
PostgreSQL 6.0	NiCad	0	0	82	0	0
	SourcererCC	38	0	84	16	27
	CCAligner	219	13	83	85	**84**
Linux 1.0	NiCad	1	0	82	3	6
	SourcererCC	12	1	87	32	47
	CCAligner	27	1	85	76	**80**
Total	NiCad	2	0	84	0.5	1
	SourcererCC	71	1	88	19	31
	CCAligner	331	18	86	83	**84**

Table 5 shows the detailed results for Java projects. Compared to NiCad and SourcererCC, CCAligner also has the best recall and F1-score in all projects. For example, CCAligner gets the best recall

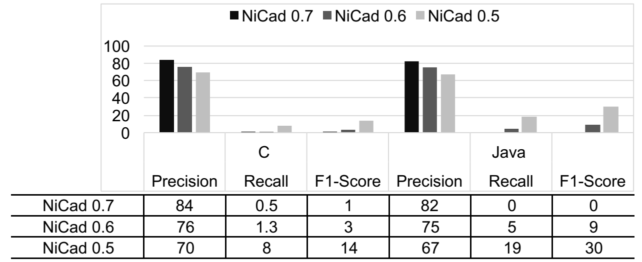
of 93% and best F1-score of 87% among all the tools in Maven 3.5.0 project. NiCad fails to detect large-gap clone in these projects, and SourcererCC still has much lower recall and F1-score although its precision is a little better than CCAligner. Finally, a manual validation identifies 525 true positive large-gap clones in these Java projects.

Table 5: Large-gap Clone Evaluation Results for Java.

System	Tool	LG	FP	Precision	Recall	F1-Score
JDK 1.2.2	NiCad	0	0	87	0	0
	SourcererCC	4	0	88	24	38
	CCAligner	15	1	86	78	**82**
OpenNLP 1.8.1	NiCad	0	0	81	0	0
	SourcererCC	5	0	85	2	4
	CCAligner	221	7	83	99	**90**
Maven 3.5.0	NiCad	0	0	80	0	0
	SourcererCC	38	1	83	18	30
	CCAligner	217	30	82	93	**87**
Ant 1.10.1	NiCad	0	0	80	0	0
	SourcererCC	13	0	84	15	25
	CCAligner	87	10	83	88	**85**
Total	NiCad	0	0	82	0	0
	SourcererCC	60	1	85	11	19
	CCAligner	540	48	84	94	**89**

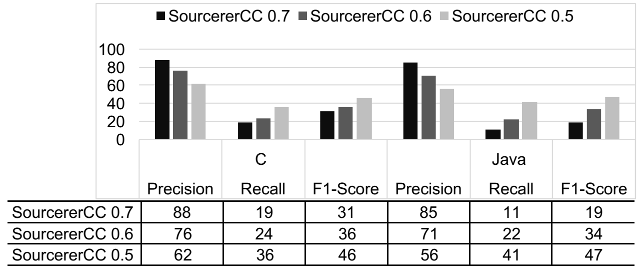
Since SourcererCC and NiCad only detect a few large-gap clones in our reference corpus containing 377 C and 525 Java true positive large-gap clones. We further run NiCad and SourcererCC with allowing more similarity thresholds to see how the precision and recall vary. We run NiCad and SourcererCC all with min similarity threshold as 60% and 50%. We measure precision by manually validating a random sample of the clones reported by SourcererCC and NiCad (400 for each language, same as CCAligner), and the recall reflects how many large-gap clones can be detected in our reference corpus. The detailed results for NiCad are in Fig. 4, and the detailed results for SourcererCC are in Fig. 5. The precision, recall and F1-score of CCAligner for the reference corpus are summarized in the total item in Table 4 and Table 5. The results show that the recall of the tools in both C and Java projects only increases a bit, but the precision becomes worse. For example, even setting similarity as 50%, SourcererCC only detects 135 large-gap clones out of 377 large-gap clones (36%) contained in the C reference corpus with 62% precision, whereas CCAligner can detect 313 large-gap clones (83%) with 86% precision. Hence, the experiments demonstrate that CCAligner is able to detect large-gap clones that the best of the competing tools are missing without harming the precision.

To further show empirically the large-gap clones we detect, we summarize the number of different types of clones detected by CCAligner in these C and Java projects in Table 6. Since λ is the ratio of code length in terms of pretty-printed lines, given pairs of clones, we approximately consider them as Type-1 or Type-2 clones if $\lambda = 1$, and when λ is not equal to 1, we approximately consider them as Type-3 clones. We also summarize the proportion of large-gap clones in Type-3 clones and all clones. We can see that there are more Type-3 clones than other types in most projects, which is consistent with existing studies [42, 46]. Among Type-3 clones, the proportion of large-gap clones varies from 11% to 37%, and among all clones, the proportion of large-gap clones varies from 3% to 22%. For example, there are 22% of clones (37% of Type-3 clones) in PostgreSQL 6.0 are large-gap clones. It shows that large-gap clones are common and exist in real software systems.



	C Precision	C Recall	C F1-Score	Java Precision	Java Recall	Java F1-Score
NiCad 0.7	84	0.5	1	82	0	0
NiCad 0.6	76	1.3	3	75	5	9
NiCad 0.5	70	8	14	67	19	30

Figure 4: Performance of NiCad with Different Similarity Thresholds.



	C Precision	C Recall	C F1-Score	Java Precision	Java Recall	Java F1-Score
SourcererCC 0.7	88	19	31	85	11	19
SourcererCC 0.6	76	24	36	71	22	34
SourcererCC 0.5	62	36	46	56	41	47

Figure 5: Performance of SourcererCC with Different Similarity Thresholds.

Table 6: Proportion of Large-gap Clones Detected by CCAligner

System	Type-1&2	Type-3	All	LG/Type-3	LG/All
Cook 2.34	551	413	964	15%	7%
Redis 4.0.0	107	118	225	19%	10%
PostgreSQL 6.0	395	593	988	37%	22%
Linux 1.0	217	253	470	11%	6%
JDK 1.2.2	89	43	132	35%	11%
OpenNLP 1.8.1	661	723	1384	31%	16%
Maven 3.5.0	5432	917	6349	24%	3%
Ant 1.10.1	309	375	684	23%	13%

Our comparison (Section 4.3) with the state-of-the-art tools will also show that CCAligner is also competitive with SourcererCC and NiCad in Type 1-3 clone detection in terms of execution time, precision and recall.

4.2.2 ***Gapped Clone Evaluation***. Since precision could be approximated with randomly validating a significant samples of the detected clones, our way of reporting the precision above is aligned with the state of the art [7, 41, 43, 45, 47]. However, for measuring recall, one needs to have an oracle [43, 50]. We have built the oracle by unioning the gapped clones detected by the subject tools from our subject systems and then manually validating from them. While this gives an approximate and relative comparison of recall among the competing tools, it does not guarantee [4, 41, 47]

the recall values we reported above. We thus wanted to examine the extent of large-gap clone detection performance of CCAligner for different gap sizes using an established procedure borrowed from Mutation [1] analysis community. In particular, we adapt an established mutation-injection based benchmarking framework [41, 50] for syntactically creating gapped clones of different sizes, and evaluate and compare CCAligner with the state of the art gapped clone detection tools including NiCad [40], SourcererCC [45], iClones [14] and Deckard [16] for recall.

Mutation-based approach can automatically and efficiently measure (and compare) the recall of clone detection tools for different types of clones. By using code mutation operators to generate and track a large number of artificial clones, we can then automatically measure how efficiently (i.e., recall) these known clones are detected by group of tools for comparing different tools.

We designed a clone-producing mutation operator that copies a code fragment and inserts a single gap of a certain length into the copied version. We used the framework with our mutation operator to generate 200 synthetic gapped clones per gap length ranging from one source line to 20 source lines. In total, our reference corpus contains 2,000 synthetic gapped clones. We constrained the clone synthesis to clones that are 15 source lines or greater before insertion of the gaps. We selected the code fragments from a large repository of Java source code (JDK and various Apache commons libraries), and injected the gapped synthetic clones one at a time into the subject system. We use the same 200 original code fragments and injection locations for each gap length, so we can compare recall across the gap lengths without bias due to the original source code used and injection locations.

We measured CCAligner's recall for the clones per inserted gap length. We configured NiCad for a 70% threshold, with identifier and literal value normalizations enabled. We configured CCAligner, Deckard and SourcererCC also with a 70% threshold. We executed iClones for a minimum clone length of 50 tokens and a minimum block length of 20 tokens.

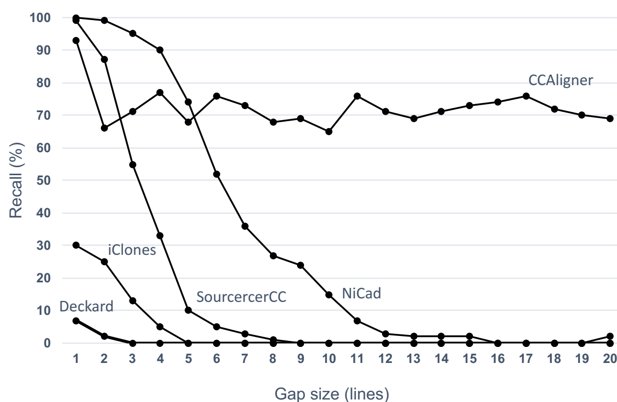


Figure 6: Recall For Gapped Clones by Gap Length

We show the recall of CCAligner and the competing tools across the gap lengths in Figure 6. As can be seen, while all the tools can detect clones with small gaps, CCAligner is the only tool that can detect clones with larger gaps. SourcererCC has good recall for gaps of one of two source lines, but recall quickly diminishes after three source lines. NiCad, due to powerful normalization, maintains high recall until gaps of length five, where its recall begins to diminish. We see that iClones and Deckard perform much worse in the gapped clone detection.

CCAligner has similar recall to NiCad and SourcererCC for single line gaps. Due to more conservative configuration, CCAligner has lower recall for clones with the smaller gaps. However, it is able to maintain this recall even for the clones with large gaps. The recall of the other tools drops off because with larger gaps it becomes more likely the clones can be classified as large-gap clones, and cannot be detected with a standard threshold (70%) by the other tools. While NiCad and SourcererCC could detect these large-gap clones by lowering their threshold, this would result in poor precision. In contrast, CCAligner can detect these large-gap clones while maintaining precision.

4.3 General Clone Detection

To evaluate the ability of CCAligner to detect general-purpose clones (i.e., Type-1, Type-2 and Type-3 clones), we extend the tool evaluation experiment in the SourcererCC publication by Sajnani et al. [45]. We execute these evaluations for our CCAligner, under the same conditions used by them, and directly compare against the previous results in the literature [45, 47, 48]. For experiments on execution time, recall and precision, we execute CCAligner for minimum clone size of 6 lines, window size $q = 6$, an edit distance $e = 1$, and a similarity threshold of 60%. The configurations of the competing tools from the experiment we extend can be found in Sajnani et al.'s work [45].

4.3.1 ***Execution Performance.*** We use the same input as used by Sajnani et al. [45] in their experiment with SourcererCC. We execute on a comparable machine with a quad-core CPU, 12GB of memory and a solid state drive. Execution performance for CCAligner and the competing tools is shown in Table 7.

We find that CCAligner has fast execution performance with scalability up to ten million lines of code. CCAligner fails for the 100MLOC input with an out of memory error. Specifically the e-mismatch index, which is kept in memory to enable fast detection, exceeds the available memory (12GB). Still, this is very good scalability, and most software systems do not reach even 10MLOC.

Compared to the state of the art, CCAligner shares the best execution time, while achieving the second best scalability. Execution time is very similar to CCFinderX and SourcererCC. However, CCFinderX only detects Type-1 and Type-2 clones, and SourcererCC misses many large-gap clones which can be detected by CCAligner, as we showed previously. CCAligner's scalability falls behind CCFinderX and SourcererCC. However, both of these tools still require days of execution time to scale to 100MLOC. SourcererCC specifically was designed to enable large-scale clone detection, while we focus on the detection of large-gap clones with CCAligner. While CCFinderX scales well, its clone detection is much simpler, focusing only on Type-1 and Type-2 clones.

4.3.2 ***Recall.*** We measure clone detection recall using Svajlenko et al.'s two proven benchmarks: (1) The Mutation and Injection Framework [41, 50] and (2) BigCloneBench [46, 48, 49].

Table 7: Execution Time for Varying Input Sizes

LOC	CCAligner	CCFinderX	Deckard	iClones	NiCad	SourcererCC
1K	**1s**	3s	2s	**1s**	1s	3s
10K	**1s**	4s	9s	**1s**	4s	6s
100K	7s	21s	1m 34s	**2s**	21s	15s
1M	**1m 13s**	2m 18s	1hr 12m 3s	—	4m 1s	1m 30s
10M	**24m 56s**	28m 51s	—	—	11hr 42m 47s	32m 11s
100M	—	3d 5hr 49m	—	—	—	**1d 12h 54m**

Table 8: Recall Measured by The Mutation Framework

Language	Clone Type	CCAligner	CCFinderX	Deckard	iClones	NiCad	SourcererCC
Java	1	100	99	39	100	100	100
	2	100	70	39	92	100	100
	3	99	0	37	96	100	100
C	1	100	100	73	99	99	100
	2	100	77	72	96	99	100
	3	100	0	69	99	99	100
C#	1	100	100	-	-	98	100
	2	100	78	-	-	98	100
	3	98	0	-	-	98	100

Table 9: Recall Per Clone Type and Precision Measured for BigCloneBench

Tool	CCAligner	CCFinderX	Deckard	iClones	NiCad	SourcererCC
Type-1	100	100	60	100	100	100
Type-2	99	93	58	82	100	98
Very Strongly Type-3	97	62	62	82	100	93
Strongly Type-3	70	15	31	24	95	61
Moderately Type-3	10	1	12	0	1	5
Precision	80	72	28	91	56	83
Precision (10LOC)	83	79	30	93	80	86

For the Mutation and Injection Framework, recall is summarized per clone type in Table 8. We also include the results of the competing tools as measured in the previous work [45]. CCAligner has perfect or near-perfect recall for clones of the first three types in all three of the tested languages. NiCad and SourcererCC perform similarly. CCAligner performs better than iClones for Type-2 clones, and performs much better than CCFinderX and Deckard. CCAligner recall is significantly better than that of CCFinderX and Deckard.

For BigCloneBench, recall is summarized in Table 9. We also show the recall of the competing tools as measured in the previous work [45]. Recall is summarized per clone type, as we described in Section 4.1. CCAligner and most of the competing tools have prefect or near-perfect Type-1 and Type-2 recall. More interesting is the comparison of Type-3 recall. CCAligner has near-perfect recall for the very-strongly Type-3 clones, falling between NiCad and SourcererCC. CCAligner has the second best strongly Type-3 recall, falling behind NiCad but ahead of SourcererCC. CCAligner also has the second best moderately Type-3 recall, although none of the tools have a significant recall for this category. These results show that CCAligner is a strong Type-3 clone detector. Overall it has the second best recall, second only to that of NiCad.

4.3.3 ***Precision***. We measured precision by manually validating a random and statistically significant sample of the clones detected by CCAligner during the BigCloneBench experiment. We randomly selected 400 clones to validate, a statistically significant sample. Precision is summarized in Table 9. The results for the other tools were taken from the previous work [45]. CCAligner has the 3rd best precision at 80%, just slightly behind SourcererCC. The previous work [45] suggests that tools that target high Type-3 recall can have low precision for small clone sizes. While 6 lines is a common minimum clone size in benchmarking [7, 45, 47, 48], many modern Type-3 clone detectors recommend a larger minimum clone size in the range 10-15 lines or equivalent [14, 40]. Therefore, as done in the experiment [45], we also measure precision for just those clones that are 10 lines in length or larger. In this case, precision increases for CCAligner to 83%.

4.3.4 ***Summary***. We summarize our results of the extension of Sajnani et al.'s [45] experiment with our CCAligner in Table 10. We show recall for all clone types and for just Type-3 clones (both considering just the Very Strongly and Strongly Type-3 categories in BigCloneBench). We show precision measured for a minimum clone size of 6 lines and 10 lines of code. We also show the maximum scalability of each tool, as well as their execution time for the 10MLOC input (the max scalability of CCAligner). Using these results we can demonstrate the position of CCAligner amongst the state of the art.

CCAligner has the second best recall, both overall and specifically for Type-3 clones, behind only NiCad, but ahead of tools like SourcererCC and iClones. While NiCad has very high recall, its execution time is much longer than CCAligner for 10MLOC inputs. As well, NiCad's precision suffers for small clones (6-9 lines),

Table 10: Recall and Precision Summary

Tool	CCAligner	CCFinderX	Deckard	iClones	NiCad	SourcererCC
Recall[1]	92	75	53	78	99	90
Recall (T3)[2]	75	26	38	38	96	68
Precision (6LOC+)	80	72	28	91	56	83
Precision (10LOC+)	83	79	30	93	80	86
Scalability	10MLOC	100MLOC+	1MLOC	100KLOC	10MLOC	100MLOC+
Time (10MLOC)	24m56s	28m51s	-	-	11hr24m	32m11s

[1] Including T1, T2, VST3 and ST3 categories. [2] Including VST3 and ST3 categories.

which CCAligner can detect these small clones with good precision. CCAligner has similar precision to SourcererCC, but better recall, specifically for the Type-3 clones. CCAligner has the best execution time for the 10MLOC input. CCAligner has competitive scalability amongst the tools. Only SourcererCC (designed for scalability) and CCFinderX (does not detect Type-3 clones) can scale to 100MLOC. While Deckard and iClones detect Type-3 clones, their recall is lower than CCAligner and they cannot scale to even 10MLOC on our standard workstation.

Our CCAligner is well situated amongst the competing tools, with excellent execution time, and good recall and precision. While NiCad has better recall, and SourcererCC better scalability, CCAligner is the best performing large-gap clone detection tool.

5 RELATED WORK

There have been numerous clone detectors in the literature, and Rattan et al. [38] summarized most of the tools in their research.

Among the text based tools [12, 18, 40], Johnson [18] applied a fingerprinting technique for comparison of source code, and Ducasse et al. [12] used dynamic pattern matching for line based comparison. However, both techniques do not support Type-3 clones detection. NiCad [40] is a text based hybrid tool, which used longest common subsequence algorithm for code comparison, and can detect Type-3 clones effectively. However, this method will fail to detect large-gap clones since the clones have large difference in LCS similarity.

Among the token based tools [14, 20, 45], SourcererCC [45] and iClones [14] support Type-3 clone detection. Sajnani et al. [45] measured overlap similarity of pairs of code blocks at token-level granularity to identify clones, which will miss many large-gap clones since the similarity measure is not robust to large gaps. iClones [14] uses suffix tree based token by token comparison to detect Type-1/Type-2 clones and then merges neighboring Type-1 and Type-2 clones to form Type-3 clones. However, iClones can only detect Type-3 clones with small gaps.

Tree and Program Dependency Graph (PDG) based tools can support the detection of Type-3 clones well, such as CloneDR [6], Deckard [16], Duplix [26] and GPLAG [28], where CloneDR and Deckard are tree based, and Duplix and GPLAG are PDG based. However, large-gap clones may affect the tree and PDG structure due to the extension of the code, and thus these tools will fail to detect large-gap clones. Besides, most tools based on these approaches are slow with poor scalability.

Among the metrics based tools [23, 25, 30], Kodhai et al. [23] applied metrics on textual representations of source code and only support Type-1 and Type-2 clones detection. Mayrand et al. [30] and Kontogiannis et al. [25] detected clones using metrics extracted from AST of source code. However, like tree based tools, they will fail to detect large-gap clones since large gaps may affect the AST structure and further affect the metrics.

There exists some other techniques for clone detection, such as MeCC [21]. MeCC detects code clones by comparing programs' abstract memory states. While they have a very good recall in Type-3 even Type-4 clones, their methods will miss the large-gap clones where large gaps affect the memory states. Besides, MeCC also suffers from large execution time.

6 LIMITATIONS

One limitation of the current implementation of CCAligner is that it cannot scale to detect clones in 100MLOC input on a standard workstation with 12GB of memory, since CCAligner stores the whole *e*-mismatch index in memory during the process of generating clone candidates. Another limitation is the evaluation work. The configurations of the tools affect the performance of clone detection [55]. To reduce this limitation, we follow the configurations of previous literature work [45] which seemed to be standard.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel clone detecting technique, and implemented the proposed technique as a tool, CCAligner. We have demonstrated the correctness of our technique, and showed the effectiveness of the proposed technique by experimentally evaluating it. We conduct an empirical study for large-gap clones detection with eight systems to demonstrate that CCAligner can detect large-gap clones that the best of the competing tools are missing, shown as the best recall and F1-score in all selected open source projects. Moreover, we develop a variant of the Mutation-Injection framework to evaluate and compare the recall of CCAligner for different gap sizes. The results further show that CCAligner is the best performing large-gap clone detection tool. We further demonstrate that CCAligner remains competitive with the best clone detectors in general Type-1, Type-2 and Type-3 clone detection, shown as the best execution time for 10MLOC input with good precision and recall. We offer CCAligner as a large-gap clone detector, and other clone related research could benefit from the detection of large-gap clones, such as clone refactoring [52].

In the future work, we are planning on improving the scalability of our proposed approach. The data structure of our algorithm is organized as $\langle key, value \rangle$, so it is especially suitable for distributed computing, like Hadoop [13].

ACKNOWLEDGMENTS

This work was supported by the National Nature Science Foundation of China under grant No. 61672480.

REFERENCES

[1] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering*. ACM, 402–411.

[2] Austin Appleby. 2016. Murmurhash hash functions. (2016). https://github.com/aappleby/smhasher/

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. 1999. *Modern information retrieval*. Vol. 463. ACM press New York.

[4] Brenda S Baker. 2007. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering* 33, 9 (2007).

[5] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. 2007. Clone smells in software evolution. In *IEEE International Conference on Software Maintenance*. IEEE, 24–33.

[6] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 368–377.

[7] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33, 9 (2007).

[8] Xiao Cheng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2016. On the feasibility of detecting cross-platform code clones via identifier similarity. In *Proceedings of the 5th International Workshop on Software Mining*. ACM, 39–42.

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

[10] James R Cordy. 2016. The TXL Programming Language. (2016). https://www.txl.ca/

[11] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. 2012. XIAO: Tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 369–378.

[12] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 109–118.

[13] The Apache Software Foundation. 2017. Apache Hadoop. (2017). http://hadoop.apache.org/

[14] Nils Göde and Rainer Koschke. 2009. Incremental clone detection. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, 219–228.

[15] Yue Jia, David Binkley, Mark Harman, Jens Krinke, and Makoto Matsushita. 2009. KClone: a proposed approach to fast precise code clone detection. In *Third International Workshop on Detection of Software Clones (IWSC)*.

[16] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 96–105.

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

[18] J Howard Johnson. 1994. Substring Matching for Clone Detection and Change Tracking.. In *Proceedings of the International Conference on Software Maintenance*, Vol. 94. 120–126.

[19] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 485–495.

[20] Toshihiro Kamiya. 2016. The official CCFinderX website. (2016). http://www.ccfinder.net/

[21] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: memory comparison-based clone detector. In *2011 33rd International Conference on Software Engineering*. IEEE, 301–310.

[22] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 187–196.

[23] E Kodhai, S Kanmani, A Kamatchi, R Radhika, and B Vijaya Saranya. 2010. Detection of type-1 and type-2 code clones using textual analysis and metrics. In *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on*. IEEE, 241–243.

[24] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*. Springer, 40–56.

[25] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. 1996. Pattern matching for clone and concept detection. *Automated Software Engineering* 3, 1-2 (1996), 77–108.

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

[27] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[28] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 872–881.

[29] Angela Lozano and Michel Wermelinger. 2008. Assessing the effect of clones on changeability. In *IEEE International Conference on Software Maintenance*. IEEE, 227–236.

[30] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. 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. 244.

[31] Shane McIntosh, Martin Poehlmann, Elmar Juergens, Audris Mockus, Bram Adams, Ahmed E Hassan, Brigitte Haupt, and Christian Wagner. 2014. Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 145–154.

[32] Manishankar Mondal, Md Saidur Rahman, Ripon K Saha, Chanchal K Roy, Jens Krinke, and Kevin A Schneider. 2011. An empirical study of the impacts of clones in software maintenance. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 242–245.

[33] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM computing surveys (CSUR)* 33, 1 (2001), 31–88.

[34] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2012. Clone management for evolving software. *IEEE transactions on software engineering* 38, 5 (2012), 1008–1026.

[35] Vern Paxson. 2016. Flex–fast lexical analyzer generator. (2016). https://github.com/westes/flex/

[36] A Perumal, S Kanmani, and E Kodhai. 2010. Extracting the similarity in detected software clones using metrics. In *Proceedings of the International Conference on Computer and Communication Technology*. IEEE, 575–579.

[37] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. 2012. Clones: What is that smell? *Empirical Software Engineering* 17, 4-5 (2012), 503–530.

[38] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

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

[40] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the IEEE 16th International Conference on Program Comprehension*. IEEE, 172–181.

[41] Chanchal K Roy and James R Cordy. 2009. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 157–166.

[42] Chanchal K Roy and James R Cordy. 2010. Near-miss function clones in open source software: an empirical study. *Journal of Software: Evolution and Process* 22, 3 (2010), 165–189.

[43] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 2014. The vision of software clone management: Past, present, and future (keynote paper). In *Proceedings of the Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*. IEEE, 18–33.

[44] Julia Rubin and Marsha Chechik. 2013. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 1233–1236.

[45] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1157–1168.

[46] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[47] Jeffrey Svajlenko and Chanchal K. Roy. 2014. Evaluating Modern Clone Detection Tools. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. IEEE Computer Society, Washington, DC, USA, 321–330.

[48] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 131–140.

[49] J. Svajlenko and C. K. Roy. 2016. BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 596–600.

[50] Jeffrey Svajlenko, Chanchal K Roy, and James R Cordy. 2013. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of the 7th International Workshop on Software Clones*. IEEE, 8–9.

[51] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. 2010. An empirical study on the maintenance of source code clones. *Empirical Software Engineering* 15, 1 (2010), 1–34.

[52] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 60–70.

[53] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 269–280.

[54] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. On detection of gapped code clones using gap locations. In *Software Engineering Conference Ninth Asia-Pacific*. IEEE, 327–336.

[55] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 455–465.

[56] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[57] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 665–676.