PAPER

NCDSearch: Sliding Window-Based Code Clone Search Using Lempel-Ziv Jaccard Distance

Takashi ISHIO[†a)], ***Member***, **Naoto MAEDA**[††], **Kensuke SHIBUYA**[††], **Kenho IWAMOTO**[††], ***Nonmembers***, ***and*** **Katsuro INOUE**[†††], ***Fellow***

SUMMARY Software developers may write a number of similar source code fragments including the same mistake in software products. To remove such faulty code fragments, developers inspect code clones if they found a bug in their code. While various code clone detection methods have been proposed to identify clones of either code blocks or functions, those tools do not always fit the code inspection task because a faulty code fragment may be much smaller than code blocks, e.g. a single line of code. To enable developers to search code clones of such a small faulty code fragment in a large-scale software product, we propose a method using Lempel-Ziv Jaccard Distance, which is an approximation of Normalized Compression Distance. We conducted an experiment using an existing research dataset and a user survey in a company. The result shows our method efficiently reports cloned faulty code fragments and the performance is acceptable for software developers.

key words: *Source code search, Normalized compression distance*

1. Introduction

Software developers may write a number of similar source code fragments including the same mistake in software products [1]–[3]. To fix the same bug in multiple locations at once, code clone detection tools have been employed in industry [4]. When developers identified a faulty source code fragment, they execute a code clone detection tool and investigate code clones of the faulty code fragment.

To efficiently perform such a bug-fixing task, software developers require an efficient code clone detection tool that takes as input a query code fragment and report its code clones in a software product. A tool specialized for the task is CBCD (Cloned Buggy Code Detector) [5]. The tool translates a query code fragment into a graph representation based on a program dependence graph [6], and then detects isomorphic subgraphs in the entire source code. Although the tool effectively detects faulty code clones, the tool requires a program dependence graph for a target program. While GrammaTech CodeSurfer is available for C language, it is difficult for developers to prepare similar tools for various programming languages (e.g. Java and C#) used for their software products.

As a programming langauge-independent search tool, we have developed a tool named NCDSearch[†]. The tool uses Normalized Compression Distance [7] for source code similarity because the distance is resilient to content changes such as renaming and reordering [8]–[10]. Our previous work [11] shows that NCDSearch effectively identifies faulty code clones for a query code fragment. On the other hand, the tool takes around 10 minutes per query on average to scan a large software product like Linux kernel comprising millions of lines of code. The performance is acceptable for some developers who have used existing code clone detection tools; however, the tool is still slow for daily bug-fixing tasks in industry.

In this study, we propose an efficient code clone search method using Lempel-Ziv Jaccard Distance (LZJD) [12], [13]. Our method is a sliding window algorithm that takes as input a query code fragment and source code. Our key component is an efficient comparison between the query and a code fragment extracted by a sliding window; while the original definition of LZJD [12] is to compare a pair of fixed-size strings, we define an algorithm that compares a query with multiple sub-strings of a code fragment at once. To further improve the efficiency, we also introduce a file selection step that quickly decides whether a file should be investigated or not.

We implemented the algorithm as a new version of NCDSearch and evaluated the performance using the CBCD benchmark [14]. The result shows that the new version is 20x faster than the previous version while keeping accuracy. The tool is deployed in NEC Corporation that employs more than ten thousand developers. In response to our tool announcement, 101 users have tried the tool. We conducted a user survey and received 15 responses from those users; 73% of them responded that they will continue to use the tool. Since the previous version has not been widely accepted in the company, the performance improvement is significant for developers.

The remainder of this paper is organized as follows: Section 2 explains the research background. Section 3 describes the proposed method. Section 4 shows the result of our performance evaluation using a benchmark dataset. Section 5 describes our user evaluation. Section 6 summarizes the results and future directions of the study.

Manuscript received January 1, 2015.
Manuscript revised January 1, 2015.

[†]The author is with Nara Institute of Science and Technology, Japan.
[††]The author is with NEC Corporation, Japan.
[†††]The author is with Osaka University, Japan.
a) E-mail: ishio@is.naist.jp

DOI: 10.1587/transinf.E0.D.1

[†]https://github.com/takashi-ishio/NCDSearch

```
Code change 1:
- perm_fmgr_info(typeStruct->typoutput,
-                 &(prodesc->arg_out_func[i]));
+ fmgr_info_cxt(typeStruct->typoutput,
+                &(prodesc->arg_out_func[i]),
+                proc_cxt);

Code change 2:
- perm_fmgr_info(typeStruct->typinput,
-                 &(prodesc->result_in_func));
+ fmgr_info_cxt(typeStruct->typinput,
+                &(prodesc->result_in_func),
+                proc_cxt);
```

Fig. 1 A pair of similar code changes performed in PostgreSQL [14]. Both changes have been performed in commit `6f7c0ea32`.

2. Background

2.1 Code Clone Detection

Duplicated source code, also known as code clones, is considered as a code smell [15]. To support software maintenance activities, various code clone detection tools have been proposed [16]. Most of the code clone detection tools are designed to efficiently detect large code clones, e.g. block-level and function-level code clones, in software products. For example, CCFinderX [17] reports code clones that are longer than 50 tokens by default. NiCad [18] detects only cloned functions or code blocks. On the other hand, a bug fix may be a small code fragment, e.g. a single line of code [19]. Fig.1 is an example pair of code clones of such a small change. When the code change 1 has been performed, we would like to detect the source code location for code change 2 (the lines of `perm_fmgr_info` function call) as a code clone. However, the code clones are ignored by the detection tools.

CBCD [5] has been proposed to detect code clones of a small faulty code fragment by comparing context information of code fragments using program dependence analysis [6]. While the tool is very effective for our purpose, it is hard to prepare program dependence analysis components for various programming languages used in software companies.

ReDeBug [20], Vuddy [21], and CLORIFI [22] have been proposed to efficiently apply security patches to software products. They take a bug fix patch as input and report source code fragments where the patch has not been applied yet. Although the usage scenario is close to our motivation, those tools use unsuitable heuritics to the bug-fixng task. For example, ReDeBug detects only source files including the entire patch content. Vuddy detects code clones of known vulnerable functions that are syntactically identical except for identifier names. Their conservative analyses do not detect modified code clones. CLORIFI introduces constraints specialized for security patches that are unavailable for general bugs.

Balachandran [23] proposed a code search algorithm using a structural similarity of abstract syntax trees. It enables a user to search code examples using syntactic patterns ignoring semantic differences such as data types and function names in a query code fragment. Our method uses a textual similarity, since clones of a buggy code fragment likely use similar functions and variables.

Siamese [24] is a code clone search tool that takes as input a query code fragment and detects its clones in a large code base such as GitHub projects. To efficiently search a large code base, the tool builds an index database for the target code base in prior to queries. However, such a database is redundant for a bug-fixing task because developers need to search code clones only once for a particular version of a product. Our method simply scans the entire source code on-the-fly.

CCGrep [25] is a code clone search tool supporting meta-tokens that are similar to regular expressions but specialized for source code. While it enables developers to specify exact code patterns they want to detect, specifying such patterns may be time-consuming for developers because they have to understand code clone patterns at first.

2.2 NCDSearch

Our previous work [11] implements a code clone search tool named NCDSearch. The tool takes as input a query code fragment q and source code. The tool extracts source code fragments of different sizes using a sliding window and compare them with the query.

The tool uses Normalized Compression Distance (NCD) for source code similarity. Given a query code fragment q and a source code fragment s, the distance is defined as follows:

$$NCD(q,s) = \frac{C(qs) - min\{C(q), C(s)\}}{max\{C(q), C(s)\}}$$

where $C(qs)$ denotes the compressed size of the concatenation of q and s, $C(q)$ denotes the compressed size of q, and $C(s)$ denotes the compressed size of s. The distance regards two source code fragments as similar if they are highly compressed by a data compression algorithm. In implementation, we use Deflate algorithm (i.e., `gzip`) for data compression. We use experimentally determined window sizes: $\{0.80|q|, 0.85|q|, 0.90|q|, 0.95|q|, |q|, 1.05|q|, 1.10|q|, 1.15|q|, 1.20|q|\}$, where $|q|$ is the number of tokens in the query code fragment.

A limitation of the approach is the computational cost. As we cannot predict the result of a data compression, the tool has to separately calculate NCD values for each source code fragment, even though different sliding windows extract very similar code fragments. As a result, NCDSearch takes around 10 minutes per query to scan a large software product like Linux kernel comprising millions of lines of code. The performance is unsatisfactory for daily bug-fixing tasks of developers.

3. Code Clone Search using LZJD

Our method identifies code clones of a query code fragment q in a set of source files F. Our method comprises three steps: file selection, code search, and filtering. The file selection step selects a subset of source files that likely include code clones of the query code fragment. The code search step scans the selected files using a sliding window and compare code fragments with the query. Finally, the filtering step removes redundant reports.

To support various programming languages used in industry (e.g. C, Java, COBOL, JavaScript, and Python), our method is designed as language independent except for lexical analysis. Our method requires only a lexer for the programming language of q and F. any programming language can be handled if the lexer can translate the query and files into sequences of tokens excluding comments and white space. Our method translates the token sequences into byte strings by concatenating null-terminated strings. For example, a source code fragment "`int i=0;`" is tokenized and then translated into a byte sequence "`int␣i␣=␣0␣;␣`", where "␣" is the null character. As our method utilizes textual similarity of source code, visual programming languages are unsupported by the method.

3.1 File Selection

The file selection step checks if each source file $f \in F$ likely includes the query code fragment q. If the file likely includes the query, we analyze the file in the code search step. Otherwise, we skip the code search step for the file. This step is inspired by ReDeBug [20] that efficiently detects unpatched files using a simple condition: N-grams(p) $\subseteq$ N-grams(f) that compares N-grams of tokens extracted from a patch p and a file f. We use a similar but relaxed condition because code clones may be different from the query code.

We define an Overlap coefficient $Overlap_N(q, f)$ that becomes higher if a greater amount of the query q is included in f:

$$Overlap_N(q, f) = \frac{|\text{N-grams}(q) \cap \text{N-grams}(f)|}{|\text{N-grams}(q)|}$$

where N-grams(x) is a multiset of N-grams of characters in a byte string representation of x. We use N-grams of characters instead of tokens because a query can be a single line of code; N-grams of tokens may be too small to be compared.

Using the Overlap coefficient, we select a subset of files F_s from source files F as follows:

$$F_s = \{f \in F \mid Overlap_N(q, f) \geq \theta_f\}$$

where θ_f is a threshold for the file selection. The selected files F_s are analyzed by the code search step. In our implementation, we experimentally decided the parameters: $N = 5$ and $\theta_f = 0.5$.

Algorithm 1 LZSet Extraction [13]

Inputs
b: A byte array representation of a query.
Outputs
$LZSet(b)$: A Lempel-Ziv Set for the byte string b.

```
Initialize S ← φ
start ← 0
end ← 1
while end ≤ length(b) do
    b_s ← b[start : end]
    if b_s ∉ S then
        S ← S ∪ {b_s}
        start ← end
    end if
    end ← end + 1
end while
return S as LZSet(b)
```

$LZSet$(`Code Change 1`) = { `p|e|r|m|_|f|mg|r_|i|n|fo|␣|(|␣t|y|pe|S|t|ru|c|t␣|-|>|␣ty|`***`po`***`|u|`***`tp`***`|`***`ut`***`|␣,|␣&|␣(|␣p|ro|d|es|c␣|->|`***`␣a|rg|_o|ut_|fu|nc|␣[|␣i|␣]`***`|␣)|`***`␣)␣|;`*** }

$LZSet$(`Code Change 2`) = { `p|e|r|m|_|f|mg|r_|i|n|fo|␣|(|␣t|y|pe|S|t|ru|c|t␣|-|>|␣ty|`***`pi|np`***`|u|`***`t␣,`***`|␣&|␣(|␣p|ro|d|es|c␣|->|`***`␣r|esu|l|t_|in|_f|un|c␣)`***`|␣)|`***`␣;`*** }

$|LZSet$(`Code Change 1`)$|$ = 49
$|LZSet$(`Code Change 2`)$|$ = 46
$|LZSet$(`Code Change 1`) $\cap$ $LZSet$(`Code Change 2`$|$ = 34
$LZJD$(`Code Change 1, Code Change 2`) = 0.443

Fig. 2 LZSet for two code changes (removed source code indicated by "-") in Fig.1 and their LZJD. Items in ***bold italic*** text are unique elements to the LZSet.

3.2 Code Search

The code search step identifies code clones in each file $f \in F_s$ selected by the previous step. In this step, we use a sliding window whose size is w tokens. For each line in the file f, we extract a code fragment c that starts from the line. The first k tokens of the code fragment c is denoted by c_k ($1 \leq k \leq w$). We detect c_k as a code clone if it is similar to q. If there exists multiple candidates, we select only the most similar code fragment.

For source code similarity between a query q and a string c_k, we adopt Lempel-Ziv Jaccard Distance (LZJD) [12] defined as follows:

$$LZJD(q, c_k) = 1 - \frac{|LZSet(q) \cap LZSet(c_k)|}{|LZSet(q) \cup LZSet(c_k)|}$$

where $LZSet(q)$ and $LZSet(c_k)$ are simplified Lempel-Ziv Sets of the byte strings of q and c_k, respectively.

Algorithm 1 shows the algorithm to calculate $LZSet(b)$ for a byte string b defined in [12]. It splits a byte string b into sub-strings, using a simplified version of the Lempel-Ziv 78 algorithm (LZ78) [26][†]. $LZSet(b)$ represents a dynamic

[†]The LZJD paper [12] says that the authors use the LZ77 algorithm [27]. However, the definition of LZSet is closer to the LZ78 algorithm in our understanding.

Algorithm 2 LZJD-based Comparison

Inputs
b: A byte array of the source code in the sliding window.
w: A window size represented by the number of tokens
$pos[i]$ $(1 \le k \le w)$: The position of the last byte of k-th token in b
$LZSet(q)$: LZSet for the query
Outputs
k_{best}: The number of tokens that the most similar to the query
$lzjd_{best}$: $LZJD(q, c_{k_{best}})$

1: Initialize $S \leftarrow \phi$
2: $start \leftarrow 0$
3: $end \leftarrow 1$
4: $intersection \leftarrow 0$
5: $lzjd_{best} \leftarrow \infty$
6: **for** $k = 1$ to w **do**
7: **while** $end \le pos[k]$ **do**
8: $b_s \leftarrow b[start : end]$
9: **if** $b_s \notin S$ **then**
10: $S \leftarrow S \cup \{b_s\}$
11: $start \leftarrow end$
12: **if** $b_s \in LZSet(q)$ **then**
13: $intersection \leftarrow intersection + 1$
14: **end if**
15: **end if**
16: $end \leftarrow end + 1$
17: **end while**
18: $lzjd \leftarrow 1 - \frac{intersection}{|LZSet(q)|+|S|-intersection}$
19: **if** $lzjd < lzjd_{best}$ **then**
20: $k_{best} \leftarrow k$
21: $lzjd_{best} \leftarrow lzjd$
22: **end if**
23: **end for**
24: **return** k_{best} and $lzjd_{best}$

Algorithm 3 Code Search

Inputs
q: A query
w: A window size represented by the number of tokens
θ_l: Threshold for code clones
f: A target file that passed file-level search
Outputs
C: Code clones detected in the file f

1: Initialize $C \leftarrow \phi$
2: Translate q into $LZSet(q)$
3: **for** each line l in f **do**
4: $b \leftarrow$ A byte string for w tokens that start from the line l
5: $pos \leftarrow$ token positions in b
6: $k_{best}, lzjd_{best} \leftarrow$ Algorithm 2 $(b, w, pos, LZSet(q))$
7: **if** $lzjd_{best} < \theta_l$ **then**
8: $C \leftarrow C \cup \{Clone(f, l, k_{best}, lzjd_{best})\}$
9: **end if**
10: **end for**
11: **return** C

dictionary for compressing the byte sequence b in the LZ78 algorithm. Conceptually, LZJD compares dictionaries for data compression instead of the actual data compression results. Fig.2 shows an example pair of LZSets for code changes in Fig.1 and their LZJD.

While we use Algorithm 1 to obtain $LZSet(q)$, we define another algorithm to efficiently compare $LZSet(q)$ with $LZSet(c_k)$ extracted by the sliding window. Our key insight is: Algorithm 1 constructs $LZSet(b)$ by linearly scanning a byte string b. Hence, when we calculate $LZSet(c_w)$, the algorithm produces $LZSet(c_1), LZSet(c_2), \cdots, LZSet(c_{w-1})$ as intermediate results. We compare $LZSet(q)$ with those intermediate results to identify the most similar sub-string c_k.

Algorithm 2 shows our source code comparison function. It takes as input a byte string b including w tokens selected by the sliding window. For each k-th token in the code fragment, the algorithm updates S at lines 7–17. A variable $intersection$ keeps the number of common elements in $LZSet(q)$ and S. At line 18, the algorithm calculates LZJD between the query q and c_k. The entire algorithm reports the most similar sub-string c_k in the sliding window.

Using Algorithm 2 as a subroutine, we define our code search step as shown in Algorithm 3. The code search takes an additional parameter w that specifies the size of a sliding window. The sliding window moves line by line within the file f. For each line, the algorithm extracts a code fragment b including w tokens, and then calls Algorithm 2 to identify a sub-string $c_{k_{best}}$ that is the most similar to q. If the $lzjd_{best}$ value is less than a threshold θ_l, the sub-string from the line is detected as a code clone.

The size of the sliding window is decided by the number of tokens in the query ignoring comments and white space. We experimentally determined the window size $w = \lfloor 1.20|q| \rfloor$, where $|q|$ is the number of tokens in the query q. This is the maximum window size we have used in our previous work [11].

The time complexity of this code search step is $O(wn)$, where n is the total size of source code. As a sliding window includes w tokens, each token in a file is processed at most w times even if a file contains one token per line. To reduce the computation time, Algorithm 2 can be executed in parallel for multiple lines or files because the algorithm have no side effect on the global data structure such as $LZSet(q)$.

The space complexity is $O(n)$. The space is required to keep tokens and its byte string representation on memory. Since the data structure for a file can be discarded after a search on the file, the actual memory footprint depends on the maximum size of individual files.

3.3 Filtering

The code search step may detect a number of overlapping code fragments as code clones. To exclude such redundant reports, we choose the most similar code fragment (i.e. that has the shortest distance) from the overlapping clones. If tied, we choose the shortest code fragment.

After the filtering, four attributes are reported for each clone: the file name, the first line number, the length, and the distance from a query. A user can easily sort the reported code clones by the locations and distances.

Table 1 Cloned Buggy Code Detector Dataset [14]

Projects	#Queries	$\|q\|$ (Med.)	#Bugs	#Files (Med.)	LOC (Med.)
PostgreSQL	14	15	39	1,058	277,959
Git	5	19	8	261	67,028
Linux	34	17.5	41	22,181	6,931,715
Total	53	1208	88	792,432	241,074,652

4. Benchmark-based Evaluation

4.1 Benchmark

To evaluate the accuracy and efficiency of the proposed method, we use a benchmark dataset for the CBCD tool [14]. The dataset includes 53 cloned bugs extracted from issue tracking systems of three OSS projects: PostgreSQL, Git, and Linux. The main programming language of the projects is C/C++. Each bug item comprises a query code fragment, a commit ID of a product version, and a list of faulty code clones in the version. Each faulty code clone is represented by a file name and the first and last line numbers in the file. The queries have the following properties:

- Most of queries include a few lines of code. The median is 2 lines of code (17 tokens). The longest query includes 14 lines (93 tokens).
- In case of 42 bugs, a single buggy clone is included in source code. The maximum number of buggy clones of a query is 18.
- In case of 25 bugs, the cloned fragments are type-1 clones (i.e. exact copies). The other clones have some differences from the query code fragments.

Table 1 shows the numbers of queries for each project, the median number of tokens of queries, and the median size of C/C++ files in the analyzed versions of the projects. The lines of code exclude comment and empty lines. Since the dataset written in [14] included some errors (e.g. incorrect line numbers and commit ids), we manually examined the commit history of three programs and updated the dataset. The full content is included in the NCDSearch source code repository[†].

For each query in the benchmark, we evaluate a list of reported code clones $C = \{c_1, c_2, \cdots, c_{|C|}\}$ sorted by their similarity (distance) values. We classify the clones in C into true positives and false positives by comparing with faulty code clones F in the benchmark. We consider a reported code clone $c_i \in C$ detects a faulty code clone $f \in F$ if c_i includes at least a single line of f. Since multiple code clones in C may detect the same faulty code clone f, we regard only the most similar (top-ranked) code clone detecting f as a true positive corresponding to f. More formally, a set of true positive clones $TP(F, C)$ is defined as follows.

$$
\begin{aligned}
TP(F, C) &= \bigcup_{f \in F} tp(f, C) \\
tp(f, C) &= \{c_i \in C \mid overlap(f, c_i) \\
&\qquad \wedge \forall k\ (1 \le k < i).\ \neg overlap(f, c_k)\}
\end{aligned}
$$

where $overlap(f, c)$ is a boolean function representing whether a line of f is included in c or not. In the definition, $tp(f, C)$ selects at most one code clone c_i corresponding to f. If c_i includes multiple faulty code clones, the same c_i is selected for those code clones. A set of faulty code clones detected by the true positives is represented as follows.

$$
Detected(F, C) = \{f \in F \mid \exists c \in C.\ overlap(f, c)\}
$$

Using $TP(F, C)$ and $Detected(F, C)$, we calculate precision and recall of the reported clones as follows.

$$
\begin{aligned}
Precision(F, C) &= \frac{|TP(F, C)|}{|C|} \\
Recall(F, C) &= \frac{|Detected(F, C)|}{|F|}
\end{aligned}
$$

In addition to precision and recall, we use mean average precision (MAP) to evaluate the effectiveness of LZJD as a ranking mechanism for C. MAP is the mean of average precision for all queries. Average precision is the mean of the precision scores obtained after each relevant document is retrieved, using zero as the precision for relevant documents that are not retrieved [28]. In our study, higher average precision values represent that true positive clones appear earlier in the list. Average precision is calculated as follows.

$$
AP(F, C) = \frac{1}{|F|} \sum_{f \in F} Precision(F, C_s(f))
$$

$$
C_s(f) = \begin{cases} \{c_k \in C \mid 1 \le k \le i\ \} & \text{if } tp(f, C) = \{c_i\} \\ \phi & \text{otherwise} \end{cases}
$$

$C_s(f)$ represents a sub-list of code clones $c_1, \cdots, c_i$ such that c_i is the true positive including f. For example, suppose there exists two faulty code clones ($F = \{f_1, f_2\}$) and our method reports three code clones ($C = \{c_1, c_2, c_3\}$). If c_1 and c_3 respectively include f_1 and f_2 (i.e., $tp(f_1, C) = \{c_1\}$ and $tp(f_2, C) = \{c_3\}$), we obtain $C_s(f_1) = \{c_1\}$, $C_s(f_2) = \{c_1, c_2, c_3\}$. The average precision is $AP(F, C) = \frac{1}{2}(\frac{1}{1} + \frac{2}{3}) = 0.833$.

4.2 Tool Configurations

We have implemented the proposed method as a new version of NCDSearch (v0.3.5). We have executed the proposed method using five configurations as follows:

P1 The proposed method with $\theta_l = 0.5$
P2 The proposed method with $\theta_l = 0.6$
P3 The code search with $\theta_l = 0.5$ without using the file selection step
P4 The code search with $\theta_l = 0.6$ without using the file selection step

[†]https://github.com/takashi-ishio/NCDSearch/#evaluation-dataset

Table 2 Performance of the tools (53 queries).

ID	Configuration	#Reported		#Detected		Precision		Recall		MAP	Time	
		Med.	Total	Med.	Total	Med.	Total	Med.	Total		Med. (sec.)	Total
P1	File Selection + LZJD $th = 0.5$	8	2570	1	82	0.158	0.032	1.000	0.932	0.749	34	26min 38sec
P2	File Selection + LZJD $th = 0.6$	49	9124	1	88	0.143	0.010	1.000	1.000	0.753	36	28min 12sec
P3	LZJD $th = 0.5$	11	3693	1	82	0.027	0.022	1.000	0.932	0.749	44	41min 30sec
P4	LZJD $th = 0.6$	112	48841	1	88	0.013	0.002	1.000	1.000	0.752	46	43min 11sec
P5	File Selection + Baseline 1	20	4914	1	88	0.083	0.018	1.000	1.000	0.757	31	52min 39sec
B1	NCD with Deflate $th = 0.5$ [11]	22	8068	1	88	0.053	0.011	1.000	1.000	0.757	618	12h 59min
B2	Normalized Levenshtein $th = 0.5$	363	711270	1	87	0.004	<0.001	1.000	0.989	0.754	65	2h 52min
B3	CCFinderX (50 tokens)	1	55	1	64*	0.500	0.782	1.000	0.750	N/A	1642	21h 30min
B4	CCFinderX ($\|q\|$ tokens)	6	367021	1	61	0.023	<0.001	1.000	0.753	N/A	539	6h 22min
B5	NiCad (Block, 3 lines)	0	19	0	13	0.000	0.684	0.000	0.148	N/A	1906	24h 53min
B6	NiCad (Block, 10 lines)	0	18	0	12	0.000	0.667	0.000	0.136	N/A	2149	27h24min
B7	NiCad (Functions, 3 lines)	0	21	0	14	0.000	0.667	0.000	0.160	N/A	2479	35h 53min
B8	NiCad (Functions, 10 lines)	0	20	0	13	0.000	0.650	0.000	0.148	N/A	2462	35h 14min
B9	CBCD (Estimated) [5], [14]	1	8408	1	82	1.000	0.010	1.000	0.932	N/A	N/A	N/A

* Multiple faulty code clones are detected as a single code clone.

P5 The file selection step combined with the normalized compression distance ($th = 0.5$)

P1 and P2 use the full version of the proposed method with different threshold values. P3 and P4 simply execute the code search step for all source files to analyze the effect of the file selection step. P5 is to analyze the effect of the code search. It combines the file selection step with the existing normalized compression distance in our previous work [11].

In the all configurations, we sorted reported code clones in ascending order of LZJD so that we can evaluate the correctness of LZJD as a source code similarity metric. We measured the execution time on Windows 10 running on Xeon E5-2690v3 processor, 64 GB DRAM, and a SSD. We use a single thread of control.

4.3 Baselines

We compare the proposed method with existing code clone detection methods as follows.

B1: NCD with Deflate ($th = 0.5$) is the best configuration in our previous work [11]. The tool compares two code fragments using Normalized Compression Distance as described in Section 2.2. We sort reported code clones in ascending order of NCD.

B2: Normalized Levenshtein Distance ($th = 0.5$) is another baseline that simply compares code fragments with a query. The normalized Levenshtein distance between code fragments (Levenshtein similarity [29]) is defined as follows.

$$NLD(q, s) = \frac{LD(q, s)}{max\{|q|, |s|\}}$$

where $LD(q, s)$ is the Levenshtein distance of two sequences q and s. It counts the number of edit operations including insertion, deletion, and modification of tokens between a query code q and a code fragment s. We remove redundant code clones using the same filtering step as the proposed method. The code clones are sorted in ascending order of NLD.

B3-B4: CCFinderX is one of the most popular code clone detection tools in Japan; it detects two sequences of tokens as a clone pair if they have the same sequence except for identifier names and constants. We executed the tool to detect code clones between the source code of a product and a file including a query code fragment, and then filtered out clone pairs that include no lines of a query code fragment. We consider a reported clone pair is correct if the pair includes at least a single line of a query code fragment and a single line of faulty code in the ground truth. B3 is the default configuration of CCFinderX. It detects code clones having at least 50 tokens by default. B4 is another configuration that extracts code clones whose length is the same as a query. Since CCFinderX does not complete a search for a low threshold, we implemented a specialized tool that directly compares code fragments obtained from the preprocessor of CCFinderX and reports only code clones whose size is $|q|$. These configurations result in lists of code clones without a ranking mechanism.

B5-8: NiCad is a tool that has achieved both high precision and recall for various types of code clones [30]. The tool compares a pair of code blocks or functions, and reports them as a clone pair if they are similar. We filter out clone pairs that include no lines of a query code fragment. We consider a reported clone pair is correct if the pair includes both a query code fragment and a faulty code fragment in the ground truth. We try four configurations comprising two levels of code clones (block-level and function-level) and two thresholds (3 lines and 10 lines of code). While NiCad reports a list of code clones with their similarity values, we do not use the similarity values for sorting code clones because they do not represent the similarity of faulty source code fragments.

B9: CBCD (Estimated) is the baseline we have created using the CBCD method [5], [14]. Since the original implementation of CBCD is publicly unavailable, we have received the source code of the tool from the authors. Nevertheless, we could not execute the tool due to technical problems. Instead of the tool execution, we analyzed the implementation details described in the technical report [14]

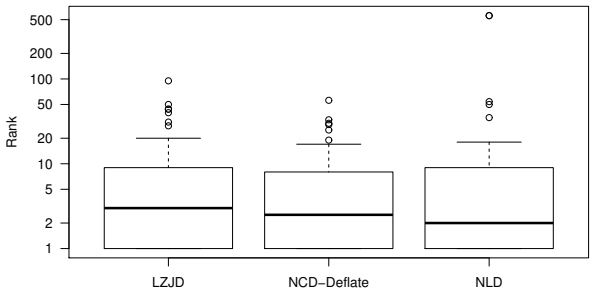


Fig. 3 Distributions of ranks of faulty code fragments

```
    for (var i=0; i < row.Cells.Count; i++)
    {
        if (row.Cells[i].Value == null)
        {
-           break;
+           continue;
        }
        ret.Add((string)row.Cells[i].Value);
    }
```

Fig. 4 An actual bug fix in the company (written in C#). The identifiers are anonymized. The bug fix replaced a `break` statement indicated by "-" with a `continue` statement indicated by "+". The same bug was found in a code clone of this code fragment.

and then estimated only the precision and recall. This was possible because the technical report classified queries into three groups:"N1: no false positives, no false negatives," "N2: no false positives, some false negatives," and "N3: some false positives, no false negatives." The report also describes that false positives are function calls when a query includes only a function call without data/control dependencies (i.e. statements) among them. For example, in case of Fig.1, CBCD reports all `perm_fmgr_info` function call sites in the program. Using the information, we counted the number of source code lines calling the functions in the N3 queries as false positives. The number of false negatives (six) is also obtained from the technical report. Using the numbers of false positives and negatives, we estimated the precision and recall of CBCD. The execution time is unavailable because we did not execute the tool.

4.4 Result

Table 2 shows the performance metrics of the tools: the number of reported clones, the number of detected faulty code clones, precision, recall, mean average precision (MAP), and the execution time. As the metrics vary by query, the "Med." columns show the medians of those values. The "Total" columns show total values for all queries.

The accuracy of the proposed method (P1 and P2) is comparable to the B1 configuration. P1 reported a smaller number of false positives but missed several clones than B1. P2 resulted in no false negatives but more false positives than B1. Compared with B2, the proposed method is more precise. The MAP column shows that cloned faulty code fragments are similarly ranked in P1, P2, B1 and B2. Fig.3 shows the distributions of ranks of faulty code fragments in three distances LZJD (P1 and P2), NCD (B1), and NLD (B2). The result shows that LZJD successfully worked as an approximation of NCD.

While keeping the accuracy, the proposed method is 20x faster than B1 according to the total time. In case of the configuration P1, the most time-consuming query took at most 83 seconds for detecting clones in Linux kernel. According to P3–P5, both of the file selection and code search steps effectively reduced the search time.

The default configuration of CCFinderX (B3) detected cloned faulty code fragments, if they are included in large code clones. While the result is precise, some faulty code fragments are missing because they have additional tokens (i.e. type-3 clones) that could not be handled by CCFinderX. Some other fragments are also missing because the preprocessor accidentally filtered out certain queries and their clones as "uninteresting" code fragments. A lower threshold (B4) does not improve a result. B4 missed some code clones detected by B3 because large code clones may accidentally include queries and their peers even if they are not directly corresponding to each other.

NiCad (B5–B8) reported a small number of code clones. This is because code clones in the dataset are neither block-level nor function-level.

The accuracy of the proposed method is also comparable to CBCD (B9). Although we did not use control and data dependence analysis, our textual similarity detected semantically similar code clones. This is probably because faulty code clones included similar identifiers to the query code fragments.

5. User Evaluation

We have deployed the tool for NEC Corporation. The software engineering group in the company has already used the B1 configuration of the tool in the group and several ongoing projects. The group implemented a GUI tool to execute a search, show the search result in a table, and highlight a selected code clone on a source code editor.

The software engineering group decided to use the P1 configuration as default. To decide the default configuration, they tested P1 and P2 configurations against two cases of faulty code clones found in the company (Fig.4 is one of the cases). As a result, they preferred P1 to P2 because P1 is more precise. They think that the code clone detection step is perceived as extra work for regular developers; they would like to encourage developers to quickly inspect code clones in their daily tasks.

The new tool is distributed on an internal website for software developers of the company. The tool is downloaded by 101 developers in the company after the first company-

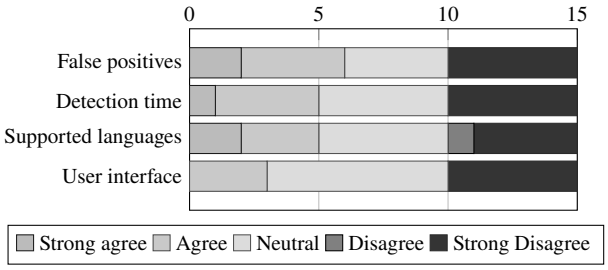


Fig. 5 Responses to the Questionnaire. We asked four questions: "Do you need the improvement on ..."

wide announcement. After a few months, we sent out questionnaires to those users. We have analyzed 15 responses from users. The questionnaires included the following questions:

1. What is your purpose of usage?
2. Do you need the improvement on the following aspects?
   - False positives
   - Detection time
   - Supported languages
   - User interface
3. Do you continue to use the tool?

The second question collected answers using a five-level Likert scale. "Strong Agree" and "Agree" answers indicate that the aspect is unsatisfactory for the users.

We found three purposes of usage: code clone search in multiple products, bug-fixing in a single product, and technical interest. 40% of the users answered that they would like to search the same bugs in related software products. 33% of the respondents answered that they would like to search source code when they have to review and fix multiple source code locations. The other users are simply interested in the new software development tool.

Fig.5 shows the answers to the second question on the necessity of further improvements. Regarding the number of false positives, 40% of users selected either "Strong Agree" or "Agree"; in other words, they requested further improvements. A possible future direction for reducing false positives is taking semantic aspects of source code into account because our method utilizes only a textual similarity.

Regarding the detection time, it is satisfactory for 33% of users who selected "Strong Disagree", while 33% of users requested further improvements. This result shows that the time efficiency is really important for industrial users. Our method is recognized as practical to a certain degree.

Supporting additional languages is also requested by 33% of users. This is probably because the tool did not support programming languages used in their projects; we believe that our implementation can support those languages by including lexers of the languages.

Regarding the user interface, only 20% of users selected "Disagree". No users selected "Strong Disagree." The result shows that a simple user interface that takes as a query and reports a list of source code location is satisfactory for most of users.

For the last question, 73% (95% CI [52.2, 92.8]) of the respondents answered that they will continue to use the tool for their software development tasks. The result shows that the overall performance of the tool is acceptable for the users. One respondent stated: "*I often execute Visual Studio's search function several times for a bug-fixing task. I think I can search such code fragments at once.*"

The respondents also gave us comments for future improvements. One respondent requested a semantic search: "*I would like to search a particular type of code fragments that are syntactically similar but semantically different. For example, I had to search* `final` *non-*`static` *fields in an actual bug-fixing task. Supporting such a search would be beneficial.*" Another respondent requested to support natural language text included in comments and document files.

6. Conclusion

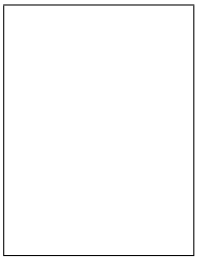
In this study, we proposed a code clone detection method that efficiently uses Lempel-Ziv Jaccard Distance. While the proposed method keeps accuracy, the method is 20x faster than the previous method based on Normalized Compression Distance. The tool has been deployed in the NEC Corporation; the user evaluation shows that the performance is acceptable for industrial developers.

In future work, we would like to take semantic aspects of source code into account to improve effectiveness, while keeping the efficiency. Another direction is monitoring the long-term effect in projects so that we can identify best practices to use the tool.

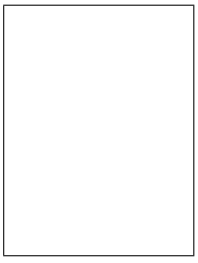
References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp.73–88, 2001.
[2] N.H. Pham, T.T. Nguyen, H.A. Nguyen, and T.N. Nguyen, "Detection of recurring software vulnerabilities," Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp.447–456, 2010.
[3] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution, pp.422–432, 2017.
[4] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, "Transferring code-clone detection and analysis to practice," 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, pp.53–62, 2017.
[5] J. Li and M.D. Ernst, "CBCD: Cloned buggy code detector," Proceedings of the 34th IEEE/ACM International Conference on Software Engineering, pp.310–320, 2012.
[6] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and Systems, vol.12, no.1, pp.26–60, 1990.
[7] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, "The similarity metric," IEEE Transactions on Information Theory, vol.50, no.12, pp.3250–3264, 2004.
[8] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, "Shared information and program plagiarism detection," IEEE Transactions on Information Theory, vol.50, no.7, pp.1545–1551, 2004.
[9] L. Zhang, Y. ting Zhuang, and Z. ming Yuan, "A program plagia-

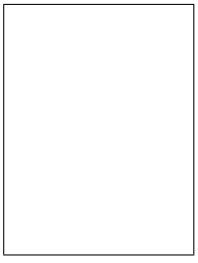
rism detection model based on information distance and clustering," Proceedings of the 2007 International Conference on Intelligent Pervasive Computing, pp.431–436, 2007.
[10] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of source code in the presence of pervasive modifications," Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation, pp.117–126, 2016.
[11] T. Ishio, N. Maeda, K. Shibuya, and K. Inoue, "Cloned Buggy Code Detection in Practice Using Normalized Compression Distance," Proceedings of the 2018 IEEE 34th International Conference on Software Maintenance and Evolution, pp.591–594, 2018.
[12] E. Raff and C. Nicholas, "An alternative to ncd for large sequences, lempel-ziv jaccard distance," Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.1007–1015, 2017.
[13] E. Raff and C. Nicholas, "Lempel-Ziv Jaccard Distance, an effective alternative to ssdeep and sdhash," Digital Investigation, vol.24, pp.34–49, 2018.
[14] J. Li and M.D. Ernst, "CBCD: Cloned buggy code detector," techreport UW-CSE-11-05-02, University of Washington, 2012.
[15] M. Fowler, Refactoring - Improving the Design of Existing Code, Addison Wesley object technology series, Addison-Wesley, 1999.
[16] 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.
[17] 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.
[18] C.K. Roy and J.R. Cordy, "An Empirical Study of Function Clones in Open Source Software," Proceedings of the 2008 15th Working Conference on Reverse Engineering, pp.81–90, 2008.
[19] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?," Proceedings of the 9th Working Conference on Mining Software Repositories, pp.74–77, 2012.
[20] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," Proceedings of the 2012 IEEE Symposium on Security and Privacy, pp.48–62, 2012.
[21] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," Proceedings of the 2017 IEEE Symposium on Security and Privacy, pp.595–614, 2017.
[22] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: software vulnerability discovery using code clone verification," Concurrency and Computation: Practice and Experience, vol.28, no.6, pp.1900–1917, 2015.
[23] V. Balachandran, "Query by example in large-scale code repositories," Proceedings of the IEEE International Conference on Software Maintenance and Evolution, pp.467–476, 2015.
[24] C. Ragkhitwetsagul and J. Krinke, "Siamese: Scalable and incremental code clone search via multiple code representations," Empirical Software Engineering, vol.24, no.4, p.2236–2284, 2019.
[25] K. Inoue, Y. Miyamoto, D.M. German, and T. Ishio, "Finding Code-Clone Snippets in Large Source-Code Collection by ccgrep," in Open Source Systems (Proc. IFIP International Conference on Open Source Systems), pp.28–41, 2021.
[26] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," IEEE Transactions on Information Theory, vol.24, no.5, pp.530–536, 1978.
[27] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE Transactions on Information Theory, vol.23, no.3, pp.337–343, 1977.
[28] C. Buckley and E.M. Voorhees, "Evaluating evaluation measure stability," Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp.33–40, 2000.
[29] M.M. Deza and E. Deza, Encyclopedia of Distances, 4 ed., Springer Berlin Heidelberg, 2016.
[30] F. Farmahinifarahani, V. Saini, D. Yang, H. Sajnani, and C.V. Lopes, "On precision of code clone detection tools," Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, pp.84–94, 2019.



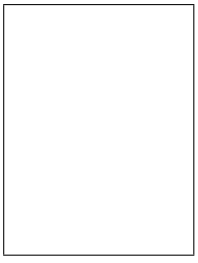
Takashi Ishio received the Ph.D degree in information science and technology from Osaka University in 2006. He was a JSPS Research Fellow from 2006-2007. He was an assistant professor at Osaka University from 2007-2017. He is now an associate professor of Nara Institute of Science and Technology. His research interests include program analysis, program comprehension, and software reuse.



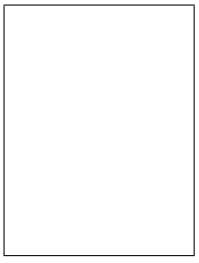
Naoto Maeda received his master's degree in informatics from Waseda University in 1999. He is an AI engineering manager at NEC Corporation, leading R&D of medical imaging AI technologies.



Kensuke Shibuya received his master's degree in informatics from Waseda University in 2008. After graduation he joined NEC Corporation. He is currently working on improving software development by adopting software engineering technologies.



Kenho Iwamoto received his master's degree in mechanical engineering from Chiba University in 2013. After graduation he joined NEC Corporation. He is currently working on improving software development by adopting software engineering technologies.



Katsuro Inoue received his Ph.D. from Osaka University in 1984. He was an associate professor of the University of Hawaii at Manoa from 1984 to 1986. After becoming an assistant professor of Osaka University in 1986, he has been a professor since 1995. His research interests include software engineering, especially software maintenance, software reuse, empirical approach, program analysis, code clone detection, and software license/copyright analysis.