Cloned Buggy Code Detection in Practice Using Normalized Compression Distance

Takashi Ishio*†, Naoto Maeda‡, Kensuke Shibuya‡, and Katsuro Inoue†
*Nara Institute of Science and Technology, Nara, Japan
†Osaka University, Osaka, Japan ‡NEC Corporation, Tokyo, Japan
Email: ishio@is.naist.jp, n-maeda@bp.jp.nec.com, k-shibuya@bq.jp.nec.com, inoue@ist.osaka-u.ac.jp

***Abstract*—Software developers often write similar source code fragments in a software product. Since such code fragments may include the same mistake, developers have to inspect code clones if they found a bug in their code. In this study, we developed a tool to detect clones of a faulty code fragment for a software company, since existing code clone detection tools do not fit the requirements of the company. The tool employs Normalized Compression Distance for source code comparison, because its definition is understandable for developers, and also it is easy to support multiple programming languages. We conducted two experiments using an existing research dataset and actual examples. Based on the evidence, the tool has been deployed in several projects in the company.**

I. Introduction

Software developers often write similar source code fragments in a software product. Those code fragments may have the same mistake [1], [2], [3]. Hence, developers have to inspect similar code fragments if they found a bug in their code. Code clone detection techniques are considered important in industry, because it enables to fix the same bug in multiple locations at once [4].

In this study, we developed a code clone detection tool for NEC Corporation, a software company that employs more than ten thousand developers. The tool detects clones of a given faulty code fragment in a software product. Although the company has already employed CCFinderX [5] to prevent unnecessary code cloning, it does not fit bug fix tasks in the company. This is because most of code clones are irrelevant to a faulty code fragment of interest to a developer. In addition, CCFinderX ignores code clones of a small faulty code fragment, e.g. a single line of code.

We designed our new tool based on the company requirements as follows.

- Usability: The tool has a simple, easy-to-use interface.
- Understandability: A result of the tool is easy to understand.
- Availability: The source code is publicly available.
- Accuracy: The tool detects clones of a faulty code fragment as much as possible with less false positives.
- Efficiency: The tool completes a search in a practical time.

The usability of the tool is important for the company so that all the developers in the company can use the tool without any special training. We adopt a `grep`-like command line interface. The tool takes a query code fragment and reports all the code fragments similar to the query in a product.

The understandability requirement is based on a past experience of the company on CCFinderX; a team who introduced it received many questions about the definition of code clones from users, because code fragments that look similar for the users were undetected as code clones. A similar observation has been reported in [6]. To address the understandability, we adopt Normalized Compression Distance [7] as a similarity metric for our tool. The distance regards two source code fragments as similar if they are highly compressed by a data compression algorithm; developers can easily analyze its property using common utilities such as `gzip` and `xz`. Despite of the simple definition, the distance is resilient to content changes such as renaming and reordering [8], [9], [10]. The resilience is important to detect code clones created by copy-and-edit operations [11].

The source code availability is required for risk control. An open-source tool can be improved by the company if necessary. The availability also enables the company to verify the safety of the tool. The safety is important for developers whose source code involves a trade secret. To achieve the availability, we implemented our tool using only open source components and opened a git repository of the tool[1]. It should be noted that the tool is accessible to their competing companies. They are considered potential contributors to keep the tool updated.

To evaluate the accuracy and efficiency of the tool, we have conducted an experiment using an existing benchmark [12]. The company also evaluated the tool with two actual examples of cloned bugs in their products. Based on the experiments, the company deployed the tool in several ongoing projects.

II. Related Work

The closest tool to our usage scenario is CBCD (Cloned Buggy Code Detector) [13]. It detects code clones of a small faulty code fragment by comparing context information of code fragments using program dependence analysis [14]. While the tool seems very effective for our purpose, it is hard to prepare for program dependence analysis for various programming languages used in the company including C, C++, Java, C#, and JavaScript.

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

ReDeBug [15] and CLORIFI [16] are also close to our usage scenario. They take a bug fix patch as input and report source code fragments where the patch has not been applied yet. However, those tools are also unsuitable to the company. The former tool detects source files including the entire patch content. Its conservative analysis does not detect modified code clones. The latter tool introduces constraints specialized for security patches that are unavailable for general bugs.

Balachandran [17] 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 tool uses a textual similarity, since clones of a buggy code fragment likely use similar functions and variables.

III. Clone Search Tool

Our tool takes a query code fragment q and a set of source files F to be analyzed. The tool uses a sliding window to extract a code fragment from F and then compares it with the query. If they are similar to each other, the code fragment is recognized as a code clone of the query code fragment. Conceptually, the tool extracts a set of code clones S with respect to a query q defined as follows.

$$S = \bigcup_{f \in F} \{s \in W(f,q) \mid NCD(q,s) \leq th\}$$

where $W(f,q)$ is a sliding window, $NCD(q,s)$ is a distance function, respectively. The tool filters overlapping code fragments in S and reports the resultant code fragments.

A. Sliding Window

A variable size sliding window $W(f,q)$ extracts code fragments from a source file f. It moves line by line within a file and extracts code fragments of different sizes. The size of the sliding window represents the number of tokens ignoring comments and white space. We experimentally determined the following 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 a query q.

B. Code Comparison

The tool compares two code fragments using Normalized Compression Distance defined as follows [7]:

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

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 NCD is a non-negative number $0 \leq r \leq 1 + \epsilon$ representing how different the two token sequences are.

To apply a data compression algorithm, we translate token sequences q and s into byte sequences by concatenating null-terminated strings. For example, a token sequence $\langle$`int`, `i`, `=`, `0`, `;`$\rangle$ is translated into a byte sequence including "`int`NUL`i`NUL`=`NUL`0`NUL`;`NUL", where NUL is the null character.

Our tool computes $C(qs)$, $C(q)$, and $C(s)$ using Deflate algorithm in `zlib` that is the most popular algorithm for `zip` and `gzip` utilities. The algorithm detects and eliminates duplicate strings in source code fragments to compute NCD.

C. Filtering

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

The tool reports code fragments in a CSV format. Each code fragment has four attributes: the file name, the first and last line numbers, and the distance from a query. A user can easily sort the reported code fragments by the locations and distances.

IV. Benchmark-based Evaluation

A. Benchmark

To evaluate the accuracy and efficiency of the tool, we employ a benchmark dataset for the CBCD tool [12]. 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 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. The longest query includes 14 lines of code.
- In case of 42 bugs, a single buggy clone is included in source code. The maximum number of buggy clones of a query is 13.
- 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 I shows the numbers of queries for each project and the median size of C/C++ files in the analyzed versions of the projects. The lines of code exclude comment and empty lines.

TABLE I
Cloned Buggy Code Detector Dataset [12]

Projects	#Queries	#Bugs	Median #Files	Median LOC
PostgreSQL	14	34	1,058	277,959
Git	5	8	261	67,028
Linux	34	39	22,181	6,931,715
Total	53	81	792,432	241,074,652

B. Tool Configurations

The normalized compression distance is dependent on a compression algorithm. To evaluate the effect, we use Deflate,

TABLE II
ACCURACY OF THE TOOLS

Configuration	#Report	Precision	Recall	MAP
NCD (Deflate, $th = 0.5$)	8107	0.010	1.000	0.741
NCD (zstd, $th = 0.5$)	368355	<0.001	1.000	0.721
NCD (xz, $th = 0.5$)	46757797	<0.001	1.000	0.722
Normalized LD ($th = 0.5$)	712087	<0.001	0.988	0.742
CCFinderX (50 tokens)	70	0.629	0.728	N/A
CCFinderX ($\|q\|$ tokens)	367021	<0.001	0.753	N/A
NiCad (Block, 3 lines)	19	0.632	0.3	N/A
NiCad (Block, 10 lines)	18	0.611	0.212	N/A
NiCad (Functions, 3 lines)	21	0.667	0.189	N/A
NiCad (Functions, 10 lines)	20	0.650	0.176	N/A

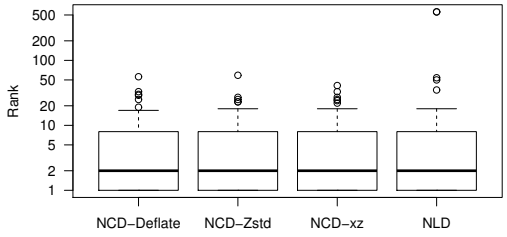

Fig. 1. Distributions of ranks of faulty code fragments

zstd, and xz with the tool. We sorted reported code fragments by the ascending order of NCD.

As a baseline that simply compares code fragments with a query, we use normalized Levenshtein distance between code fragments (Levenshtein similarity [18]) defined as follows.

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

where $LD(q, s)$ is the Levenshtein distance. 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 also use CCFinderX and NiCad as baselines of code clone detection techniques. The tools detect code clones between the source code of a product and a file including a query code fragment. We 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. CCFinderX detects code clones having at least 50 tokens by default. To analyze the effect of the default threshold, we also extract 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. In case of NiCad, we tried four configurations comprising two levels of code clones (block-level and function-level) and two thresholds (3 lines and 10 lines of code).

C. Result

Table II summarizes the numbers of reported clones, precision, recall, and mean average precision (MAP) for each configuration. CCFinderX and NiCad have no MAP values because they have no ranking features.

Our tool identified all the cloned faulty code fragments, while it reports a large number of false positives. The MAP column shows that cloned faulty code fragments are similarly ranked, while the numbers of reported clones are different. Fig.1 shows the distributions of ranks of faulty code fragments in the results. It shows that the top-20 code fragments include most of cloned bugs. Although the result of normalized Levenshtein distance (NLD in Fig.1) is comparable to NCD, it missed buggy clones using different variable names.

The default configuration of CCFinderX detected cloned faulty code fragments, if they are included in large code clones. A lower threshold does not improve a result. 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 missing because the preprocessor accidentally filtered out certain queries and their clones as "uninteresting" code fragments.

NiCad reported a small number of code clones, while its precision is similar to CCFinderX. This is because code clones in the dataset are neither block-level nor function-level.

TABLE III
PERFORMANCE OF THE TOOLS

Configuration	Median Time (seconds)			Total Time
	Postgres	Git	Linux	
NCD (Deflate, $th = 0.5$)	48	7	1,130	12h 26min
NCD (zstd, $th = 0.5$)	20	3	430	4h 35min
NCD (xz, $th = 0.5$)	830	224	20,170	176h 38min
Normalized LD ($th = 0.5$)	7	2	165	2h 45min
CCFinderX (50 tokens)	69	29	2,234	21h 30min
CCFinder ($\|q\|$ tokens)	29	10	678	6h 22min
NiCad (Block, 3 lines)	117	59	2,498	24h 53min
NiCad (Block, 10 lines)	133	43	3,116	27h 24min
NiCad (Functions, 3 lines)	111	55	3,582	35h 53min
NiCad (Functions, 10 lines)	118	56	3,581	35h 14min

Table III shows the time performance to process all the queries by a tool. The time has been measured on Windows 10 running on Xeon E5-2690v3 processor, 64 GB DRAM, and a SSD. Each tool uses a single thread of control. The result shows that our tool is comparable to existing clone detection tools. The time is acceptable for the company.

V. EXAMPLE-BASED EVALUATION

The software engineering group in the company evaluated the tool using two actual cases of cloned bugs. In those cases, developers fixed faulty code fragments but missed their clones at that time. The developers spent extra effort to fix the clones later.

Fig.2 shows one of the examples. The identifiers in the source code are anonymized. Given the original code of Fig.2, the tool reported nine code fragments shown in Table IV. The group confirmed that the most similar code to the query (#2 in the table) is the source code that was not fixed in the product. The group also confirmed that the #3 and #4 code fragments are also relevant to the bug fix task, while the fragments are correctly implemented.

```
    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. 2. An actual bug fix in a product (written in C#). The identifiers are anonymized. The bug fix replaced a `break` statement indicated by "-" with a `continue` statement indicated by "+".

TABLE IV
A SEARCH RESULT OF OUR TOOL FOR THE CODE FRAGMENT IN FIG.2

#	NCD	Relevant?	Description
1	0.067	Yes	The query itself
2	0.182	Yes	The cloned faulty code fragment
3	0.367	Yes	A similar idiom correctly implemented
4	0.418	Yes	Another similar idiom correctly implemented
5	0.455	No	A similar loop using `continue`
6	0.455	No	A similar loop using `continue`
7	0.491	No	A loop structure without `break` statement
8	0.497	No	A loop structure without `Add`
9	0.497	No	A loop structure without (Same as #8)

Fig.3 shows the other bug fix example in another product. The identifiers in the code are also anonymized. Given the query, our tool reported three code fragments shown in Table V. The most similar source code fragment to the query is the clone that was not fixed in the product.

Based on the evaluation results, the software engineering group recognized that the tool meets their requirements. They decided to deploy the tool in the group themselves (around 23 developers working in two countries) and several ongoing projects. They emphasized the usability of the tool. They implemented a GUI in prior to the deployment so that users can quickly execute the tool.

The group is also interested in deployment of the tool to their standard development environment used by more than ten thousand developers. Toward the deployment, they still have concerns about best practices to use the tool, for example:

- How many lines of code should be included in a query?
- What is an appropriate threshold value?

Identifying best practices is crucial for developers to effectively use the tool in their daily work. Since such aspects have not been investigated in the experiments, our future work is a long-term field study with the early adopters in the company.

VI. CONCLUSION

In this study, we developed a `grep`-like code clone detection tool for a software company using Normalized Compression Distance. The tool has been deployed in several projects after the evaluation.

In future work, we will monitor the long-term effect in projects so that the company can identify best practices to use the tool. We are also planning to extend the tool to search documents in addition to source files, as requested by the early adopters.

```
    if ((*list)->count > 0) {
        iCount = (*list)->count - 1;
        for (; iCount >= 0; iCount--) {
            if ((*list)->filename[iCount] != NULL) {
                free((*list)->filename[iCount]);
            }
        }
-       free(*list);
    }
+   free(*list);
```

Fig. 3. Another bug fix example in another product (written in Visual C++). The identifiers are anonymized. The bug fix moved a function call `free` in order to fix a memory leak.

TABLE V
A SEARCH RESULT OF OUR TOOL FOR THE CODE FRAGMENT IN FIG.3

#	NCD	Relevant?	Description
1	0.052	Yes	The query itself
2	0.052	Yes	The cloned faulty code fragment
3	0.482	No	A similar loop without the outer `if` statement

REFERENCES

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. SOSP*, 2001.

[2] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proc. ASE*, 2010.

[3] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *Proc. ICSME*, 2017.

[4] Y. Dang, D. Zhang, S. Ge, R. Huang, C. Chu, and T. Xie, "Transferring code-clone detection and analysis to practice," in *Proc. ICSE*, 2017.

[5] 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.

[6] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[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 plagiarism detection model based on information distance and clustering," in *Proc. IPC*, 2007.

[10] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of source code in the presence of pervasive modifications," in *Proc. SCAM*, 2016.

[11] 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.

[12] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," University of Washington, techreport UW-CSE-11-05-02, 2012. [Online]. Available: https://homes.cs.washington.edu/~mernst/pubs/buggy-clones-tr110502.pdf

[13] ——, "CBCD: Cloned buggy code detector," in *Proc. ICSE*, 2012.

[14] 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.

[15] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proc. IEEESP*, 2012.

[16] 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.

[17] V. Balachandran, "Query by example in large-scale code repositories," in *Proc. ICSME*, Sept 2015, pp. 467–476.

[18] M. M. Deza and E. Deza, *Encyclopedia of Distances*, 4th ed. Springer Berlin Heidelberg, 2016.