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 tool1. 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.

1https://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 \( \{int, i, =, 0, ;\} \) is translated into a byte sequence including "\( \text{null} \text{null} \text{null} \text{null} \text{null} \)" , where \( \text{null} \) is the null character.

Our tool computes \( C(qs), C(q), \) and \( C(s) \) using Deflate algorithm in \texttt{zlib} that is the most popular algorithm for \texttt{zip} and \texttt{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++.

<table>
<thead>
<tr>
<th>Projects</th>
<th>#Queries</th>
<th>#Bugs</th>
<th>Median #Files</th>
<th>Median LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PostgreSQL</td>
<td>14</td>
<td>34</td>
<td>1,058</td>
<td>277,959</td>
</tr>
<tr>
<td>Git</td>
<td>5</td>
<td>8</td>
<td>261</td>
<td>67,028</td>
</tr>
<tr>
<td>Linux</td>
<td>34</td>
<td>39</td>
<td>22,181</td>
<td>6,931,715</td>
</tr>
<tr>
<td>Total</td>
<td>55</td>
<td>81</td>
<td>792,432</td>
<td>241,074,652</td>
</tr>
</tbody>
</table>

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.
precision, recall, and mean average precision (MAP) for each.

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 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 most similar code to the query (#2 in the table) is the source code that was not fixed in the product.
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

<table>
<thead>
<tr>
<th>#</th>
<th>NCD</th>
<th>Relevant?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.067</td>
<td>Yes</td>
<td>The query itself</td>
</tr>
<tr>
<td>2</td>
<td>0.182</td>
<td>Yes</td>
<td>The cloned faulty code fragment</td>
</tr>
<tr>
<td>3</td>
<td>0.367</td>
<td>Yes</td>
<td>A similar idiom correctly implemented</td>
</tr>
<tr>
<td>4</td>
<td>0.418</td>
<td>Yes</td>
<td>Another similar idiom correctly implemented</td>
</tr>
<tr>
<td>5</td>
<td>0.457</td>
<td>No</td>
<td>A similar loop using continue</td>
</tr>
<tr>
<td>6</td>
<td>0.455</td>
<td>No</td>
<td>A similar loop using continue</td>
</tr>
<tr>
<td>7</td>
<td>0.491</td>
<td>No</td>
<td>A loop structure without break statement</td>
</tr>
<tr>
<td>8</td>
<td>0.497</td>
<td>No</td>
<td>A loop structure without Add</td>
</tr>
<tr>
<td>9</td>
<td>0.497</td>
<td>No</td>
<td>A loop structure without (Same as #8)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>#</th>
<th>NCD</th>
<th>Relevant?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.052</td>
<td>Yes</td>
<td>The query itself</td>
</tr>
<tr>
<td>2</td>
<td>0.052</td>
<td>Yes</td>
<td>The cloned faulty code fragment</td>
</tr>
<tr>
<td>3</td>
<td>0.482</td>
<td>No</td>
<td>A similar loop without the outer if statement</td>
</tr>
</tbody>
</table>

REFERENCES

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.

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