Finding repeated strings in code repositories and its applications to code-clone detection

Yoriyuki Yamagata*, Fabien Hervé†, Yuji Fujiwara*‡ Katsuro Inoue*‡
*National Institute of Advanced Science and Technology (AIST)
Email: yoriyuki.yamagata@aist.go.jp
†University of Nantes
Email: fabien.herve1@etu.univ-nantes.fr
‡Osaka University
Email: {inoue, y-fujiwr}@ist.osaka-u.ac.jp

***Abstract*—Although researchers have created many advanced code-clone detection techniques, more effort is required to realize wide adaptation of these techniques in the industry. One of the reasons behind this is the reliance of these advanced techniques on lexing and parsing programs. Modern programming languages have complex lexical conventions and grammar, which evolve constantly. Therefore, using advanced code-clone detection techniques requires substantial and continuous effort. This paper proposes a lightweight language-independent method to detect code clones by simply finding repeated strings in a code repository, relying on neither lexing nor parsing. The proposed method is based on an efficient technique developed in a bio-informatics context to find repeated strings. We refer to the repeated strings in the source-code as *weak Type-1 clones*. Because the proposed technique normalizes newlines, tabs, and white spaces into a single white space, it can find clones in which newline positions or indentations are changed, as often in the case when copy-pasting occurs. Although the proposed method only finds verbatim copies, it also makes interesting observations regarding repository structures. Many developers may prefer the proposed simple approach because it is easier to understand than other advanced techniques that use heuristics, approximation, and machine learning.**

I. Introduction

Finding *code clones* (similar code fragments) has important applications such as refactoring, fixing bugs, detecting copyright violations, and software traceability. The number of code clones is a measure of the code quality because as the number of code clones increases, the maintenance of software becomes more difficult.

Code clones are classified into four types: Type-1, Type-2, Type-3, and Type-4 clones [1]. Type-1 clones are those that match verbatim, for which only minor modifications, such as changing newline characters or removing comments, are allowed. Type-2 clones are those for which changes in identifiers and literals are also allowed. If identifiers are changed consistently (the same identifiers are modified to the same identifiers), the clone is called consistent Type-II. Otherwise, it is called blind Type-II. Type-3 clones are syntactically similar code snippets for which addition or deletion of statements is allowed. Type-4 clones refer to those that have functional similarity but may not have syntactical similarity. Code clones can also be classified based on their granularity (e.g., tokens, lines, statements, functions/methods, and files).

Most previous methods of finding code clones require lexing and paring of the source code. The syntax of a modern programming language is often complex and frequently updated. Therefore, developing and maintaining code-clone detection tools based on lexing and parsing require substantial and continuous effort. Furthermore, a lexer and a parser make a code-clone detection tool language-specific, rendering the deployment of such a tool inconvenient.

This paper proposes a lightweight code-clone detection method, which is easier to develop and deploy; it finds repeated strings, referred to as *weak Type-1 clones* in the code. Finding repeated substrings in a string has been widely studied in the bio-informatics context [2], [3], [4]. We adopted Barenbaum et al.'s technique [4] and developed a tool called *CodeRepeat* that can find all repeated strings in a software repository. Unlike previous language-independent approaches, the proposed method replaces all newlines, white spaces, and tabs into single spaces; thus, it does not rely on newline positions. Therefore, it can find a clone in which newline positions are changed or removed, which is common in copy-pasted code, while always outputting exact matches. Further, it does not rely on any approximation methods such as approximate substring matching or locality sensitive hashing. Avoiding approximation methods, heuristics, or machine learning improves developers' understanding of tools and reduces false positives.

To make the proposed technique strictly language independent, comments are not removed. Thus, the proposed method can only find a weak form of a Type-1 clone, but finding such clones has legitimate application potential. Weak Type-1 clones indicate simple copy-pasting; they are easier to refactor than Type-2 or Type-3 clones are. Finding verbatim copies, including comments, provides strong evidence of plagiarism.

We also showed that CodeRepeat can find interesting clones by applying the proposed method to large-scale open-source code repositories. We found that large-scale source-code repositories often contain multiple copies of third-party libraries, which are in different name spaces and have different versions. These multiple copies of libraries are clearly intended by developers and may be difficult to avoid, but they seriously affect software maintenance.

This paper is organized as follows. Section II discusses related works. Section III presents the algorithm used, its implementation, and the evaluation methods. Section IV presents the evaluation results. Section V presents the conclusion of this study.

II. Related Works

Extensive studies on code clones, targeting each clone type, have been conducted. Duplo [5] and Simian [6] primarily target Type-1 clones by matching a hash value of each line. CCFinder [7] and its new implementation, CCFinderX [8], tokenize the source code and target Type-1 and Type-2 clones. NiCad [9] parses the source code and applies a transformation to a syntax tree. NiCad targets Type-1, Type-2, and Type-3 clones. SourcererCC [10] uses heuristics based on a bag-in-word approach to efficiently find Type-1, Type-2, and Type-3 clones. Sieamese [11] uses an n-gram-based method to quickly find clones of a given code. CCAligner [12] specializes in finding clones with large gaps. Oreo [13] uses a neural method to detect Type-3 and Type-4 code clones. For more complete reviews and benchmarks, please refer to [1], [14], [15].

Among these tools, we select Duplo, CCFinderX, and NiCad as baselines for comparison because they are mature and widely used. Duplo, CCFinderX, and NiCad are representative text-, token-, and syntax-based methods, respectively.

CodeRepeat has an advantage over hash-based approaches used by Duplo and Simian: they typically assume that the code is separated by newlines. In contrast, CopeRepeat replaces all newlines, tabs, and white spaces with single white spaces. Therefore, CodeRepeat can find clones in which newline locations or indentations are modified. Such modifications are common when code is copy-pasted. We did not select SourcererCC as a baseline because the Github version of SourcererCC only supports Java and Python and not C/C++, using which many open-source software are written. The lack of C/C++ support would suggest difficulties in supporting multiple languages even for a token-based approach. We did not compare CodeRepeat with CCAligner, Oreo, or Siamese because their goals are different from those of ours. For many developers, the behavior of a heuristic- or machine learning-based approach would be difficult to understand, whereas that of CodeRepeat would be easy to understand because it uses simple string matching. Further, we do not use any approximation method, such as locality sensitive hashing or approximate substring matching, because it can create many false positives, unless it is fine-tuned. Although the proposed method is simple, it found interesting clones in well-known open-source code repositories.

III. Method

A. Algorithm

We identify code clones with either *maximal repeats* or *super maximal repeats* [16] in the source code, depending on the purpose of code-clone detection. *Maximal repeats* and *super maximal repeats* have precise mathematical definitions.

Definition 1: Let S be a string. We write a substring of S from positions p_1 to p_2 (including the end) as $S[p_1{:}p_2]$.

- A *maximal pair* (p_1, p_2, l) is a pair such that $S[p_1{:}p_1+l] = S[p_2{:}p_2+l]$ but $S[p_1 : p_1+l+1] \neq S[p_2 : p_2+l+1]$.
- A *maximal repeat* is a substring R of S such that $R = S[p_1{:}p_1+l] = S[p_2{:}p_2+l]$, where (p_1, p_2, l) is a maximal pair.
- A *super maximal repeat* is a maximal repeat that is not a substring of any maximal repeat.

For example, in the string "$\underline{\mathrm{ab}}\mathrm{x}\underline{\mathrm{ab}}\mathrm{y}\underline{\underline{\mathrm{ab}}\mathrm{c}}\mathrm{z}\underline{\underline{\mathrm{ab}}\mathrm{c}}$," "ab" and "abc" are maximal repeats. "ab" is a maximal repeat because "abx" is not repeated; therefore, the first occurrence of "ab" cannot be extended. "abc" is the only super maximal repeat in the string.

Many algorithms that efficiently find maximal pairs have been proposed in the bio-informatics context [2], [3], [4]. Barenbaum et al. [4] and Kulekci et al.'s algorithms [3] have the same asymptotic complexity ($O(n)$-space and $O(n \log(n))$-time), whereas Beller et al.'s algorithm [2] has $O(n)$-space and $O(n)$-time complexity. Barenbaum et al.'s algorithm uses a suffix array to find maximal and super maximal repeats. It is well known that we can find maximal and super maximal repeats using a suffix tree in linear time [16]. However, a suffix tree is not practical for a large input because of a large constant factor for space consumption. Barenbaum et al. replaced suffix trees with suffix arrays; efficient ($O(n \log(n))$-time) construction of these suffix arrays achieved lower memory requirement. Kulekci et al. used compressed data structures to achieve a further memory reduction. However, unlike genome sequences, comprising four "alphabets," the software code comprises bytes, with 256 characters. This makes compressed data structures less memory-efficient. Moreover, using compressed data significantly slows down the algorithm. We experimentally compared Kulekci et al. [3]'s and Barenbaum et al. [4]'s implementations, which were kindly provided by the authors. Although both algorithms have the same asymptotic complexity, Barenbaum et al.'s algorithm is significantly faster, and its memory consumption is comparable to that of Kulekci et al.'s algorithm. Therefore, we select Barenbaum et al.'s findrepset for our implementation.

B. Implementation

Figure 1 presents an overview of CodeRepeat. CodeRepeat treats files as pure byte strings (not Unicode characters). First, a target code repository is preprocessed. To handle a large number of files, CodeRepeat concatenates all files into a single file. Although Barenbaum et al.'s findrepset can simultaneously handle multiple files, the number of command-line arguments that can pass the subprocess call is limited. Therefore, we chose passing a single file to findrepset.

Because we need to recover files and lines in which clones appear, CodeRepeat generates "Charmap" and "Linemap," which provide mapping from locations in the concatenated file to file names and line numbers, respectively. CodeRepeat also replaces newline characters, tabs, and sequences of white spaces with single white spaces. We emphasize that the

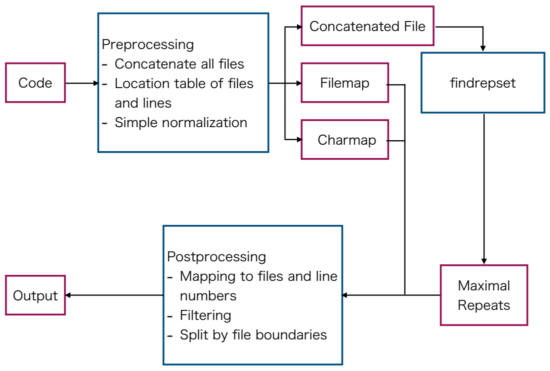

Fig. 1. Implementation

normalization performed by CodeRepeat is much simpler than lexing. A concatenated file is passed to findrepset. CodeRepeat can instruct findrepset to find maximal or super maximal repeats and the lower limits of length and frequency of repeats that it finds. The generated maximal or super maximal repeats are combined with information from Charmap and Linemap to obtain the locations of repeats in the source files during the postprocessing phase. Postprocessing splits a clone located across file boundaries into multiple clones. Postprocessing optionally removes clones that solely consist of blank characters or are null. The output uses a simple JSON format. Output compression is supported because the number and size of maximal repeats can be very large.

C. Evaluation

We performed two experiments to examine the usefulness of CodeRepeat. First, We perform the benchmark using BigCloneEval [17], which measures recall rates of different types of clones in BigCloneBench [14], Type-I, Type-II (blind and consistent), and Type-III with different degrees of syntactical similarities, very strong (90% - 100%), strong (70% - 90%), and moderate (50% - 70%). BigCloneBench provides a well-curated dataset of code clones extracted from open-source Java projects and is widely used to evaluate code-clone detection tools. We configured CodeRepeat to find small maximal repeats (40 bytes). We configured CCFinderX using default options. We configured NiCad and Duplo using default options but smaller minimum size (6 lines) to find small clones. The configurations used for BigCloneEval are summarized in Table I.

Next, we applied CodeRepeat, Duplo, CCFinderX, and NiCad to large-scale open-source code repositories, OpenSSL, Python, GCC, Firefox, and Android (Table II). Although we aim to show that CodeRepeat is reasonably fast and memory-efficient, we are more interested in demonstrating the stability and usefulness of the tools. Therefore, we measured the time and memory used by each tool, collected errors that occurred

TABLE I
CONFIGURATION OF TOOLS USED FOR BIGCLONEEVAL

Tools	Version	Configuration
CodeRepeat	Not applicable	Maximal repeats, min length 40 bytes
Duplo	24 Sep, 2020	Min length 6 lines
CCFinderX	10.2.7.4	Minimum length 50 tokens
NiCad	6.1	Min length 6 lines, max length 2500 lines, threshold 0.3

TABLE II
TARGETED OPEN-SOURCE REPOSITORIES

Software	Version	Number of Lines (KLoC)	Language[1]
Python	3.8.5	610	C/C++
OpenSSL	1.1.1h	624	C/C++
GCC	10.2.0	6311	C/C++
Firefox	80.0	11 205	C/C++
Android	11.0.0r3	20 823	Java

[1] For a project using multiple languages, the listed language was processed.

TABLE III
CONFIGURATION OF TOOLS APPLIED TO OPEN-SOURCE PROJECTS

Tools	Configuration
CodeRepeat	Super-maximal repeats, min length 400 bytes
Duplo	Min length 10 lines
CCFinderX	Minimum length 50 tokens, detect only Type-1
NiCad	Min length 10 lines, threshold 0.0, no custom contextual normalization

TABLE IV
RECALL (OVERALL)

Clone Class	CodeRepeat	Duplo	CCFinderX	NiCad
Type-1	100.0	38.2	98.6	99.9
Type-2	60.0	40.3	88.5	99.9
Type-2 (blind)	2.4	0.5	72.1	99.6
Type-2 (consistent)	65.2	44.0	90.0	99.7
Very-Strongly Type-3	11.1	0.5	27.1	99.4
Strongly Type-3	3.1	0.3	9.7	63.0
Moderately Type-3	0.1	0.4	0.5	0.4

during the execution of each tool. We configured each tool as described in Table III for open-source projects to find larger clones. For a fair comparison, we instructed CCFinderX and NiCad to only find Type-1 clones. To demonstrate that our method helps understand and restructure the source code repository, we analyzed the top 5 largest clones in each source code repository.

IV. RESULTS

A. BigCloneEval

Table IV shows the recall rate obtained for each tool by BigCloneEval. Although CodeRepeat was designed for weak Type-1 clones, it found almost all Type-1 clones and a moderate number of Type-2 clones. Because CodeRepeat outputs only exact matches, the precision of CodeRepeat should be 100%. Duplo performed poorly because it contains an integer overflow bug.

TABLE V
EXECUTION TIME (SECONDS)

Repositories	CodeRepeat	Duplo	CCFinderX	NiCad
Python	21	399	209	98
OpenSSL	28	219	219	77
GCC	261	2503	2503	2087
Firefox	694	10 696	4230	528
Android	1398	Not finished	6717	Failed

TABLE VI
MEMORY CONSUMPTION (MEGA BYTES)

Repositories	CodeRepeat	Duplo	CCFinderX	NiCad
Python	252	337	29	211
OpenSSL	316	32	32	134
GCC	178	558	558	211
Firefox	4778	2910	536	214
Android	9601	Not finished	684	Failed

B. Application to open source projects

Tables V and VI show the execution time and memory consumption of each tool, respectively. The tools ran on a machine with an Intel Xeon E5-1603 v4 2.80GHz Quad Core, 32 GB RAM. CCFinderX ran as a native Windows application, whereas the others ran in a Docker container using Ubuntu 20.0.4 LTS. CodeRepeat was equally as fast as baseline tools and handled a project with more than 20 000 K lines of code. It required more memory than baseline tools, but the memory required was within the amount of available memory in a modern personal computer.

CodeRepeat handled all projects without any modification, except for Android, in which a precompiled_kernal directory had to be removed because of an excessive number of nests of symbolic links. We applied same modification to the repository used for other tools because the issue is not tool-specific. Duplo required greater execution times for large projects, e.g., it did not finish for Android after running for 1 day. For CCFinderX, we needed to remove files with null bytes in GCC and manually extract Java files from Android projects. NiCad failed to generate any result for Android and could not parse many files, for example, 1251 files in GCC. It shows that CodeRepeat can handle a complex code repository without error, while other tools often cause errors when handling real-world projects. Being error-free is an important property in practice, arguably more so than accuracy, recall, or performance, which are properties that modern code-clone detection research is chasing.

Among the detected clones in each repositories, the case for Firefox is particularly interesting, because we find that Firefox, a widely used web browser, contains many large scale code clones. Table VII shows the top five largest clones in Firefox. The "Path" and "Line" columns show the location in which a clone pair is located. The "Description" column presents the descriptions of a clone pair. As shown in the table, Firefox contains three versions of SQLite, namely versions 3.32.3, 3.31.1, and 3.29.0, as separate packages. Although it would be

TABLE VII
FIVE LARGEST CLONE PAIRS IN THE FIREFOX REPOSITORY FOUND BY CODEREPEAT

Path	Lines	Description
third_party/sqlite3/src/sqlite3.c	205312 – 224998	SQLite 3.32.3
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.c	203984 – 223670	SQLite 3.31.1
third_party/sqlite3/src/sqlite3.c	1201 – 7023	SQLite source
third_party/sqlite3/src/sqlite3.h	162 – 6034	SQLite header
third_party/sqlite3/src/sqlite3.c	1201 – 7023	SQLite source
third_party/sqlite3/src/sqlite3.h	162 – 6034	SQLite header
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.c	1204 – 6992	SQLite source
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.h	162 – 5950	SQLite header
security/nss/lib/sqlite/sqlite3.c	1206 – 6794	SQLite 3.29.0
security/nss/lib/sqlite/sqlite3.h	162 – 5750	SQLite 3.29.0

TABLE VIII
FIVE LARGEST CLONE PAIRS IN THE FIREFOX REPOSITORY FOUND BY CCFINDERX

Path	Lines	Description
third_party/sqlite3/src/sqlite3.c	200894 – 229013	SQLite 3.32.3
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.c	199568 – 227684	SQLite 3.31.1
third_party/sqlite3/src/sqlite3.c	782 – 25858	SQLite 3.32.3
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.c	785 – 25717	SQLite 3.31.1
security/nss/lib/sqlite/sqlite3.c	787 – 20792	SQLite 3.29.0
third_party/sqlite3/src/sqlite3.c	782 – 21461	SQLite 3.32.3
security/nss/lib/sqlite/sqlite3.c	787 – 20792	SQLite 3.29.0
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.c	785 – 21320	SQLite 3.31.1
security/nss/lib/sqlite/sqlite3.c	39571 – 49157	SQLite 3.29.0
third_party/rust/libsqlite3-sys/sqlite3/sqlite3.c	40317 – 49908	SQLite 3.31.1

intentional and may be difficult to avoid, this suggests that the developers must update each instance of the library whenever a serious bug is found in these instances. Another finding was that many declarations were shared between source files and header files, which would require simultaneous updating.

Although we do not present the details here, we also observed that Android "repackages" the same libraries to different namespaces.

CCFinderX reported clones (Table VIII) between each version of SQLite in Firefox, whereas CodeRepeat only reported clones between Versions 3.23.3 and 3.31.1 as the top 5 largest clones. This indicates that CCFinderX is more powerful than CodeRepeat in finding large-scale clones, although CCFinderX requires tokenizers. CCFinderX discards comments, declarations, and variable definitions under the default setting, which could have valuable information. NiCad reported clones (Table IX) in a small fragment, in which very few were more than

TABLE IX
FIVE LARGEST CLONE PAIRS IN THE FIREFOX REPOSITORY FOUND BY NICAD

Path	Lines	Description
security/nss/lib/zlib/inflate.c	625 – 1275	identical copy of zlib
modules/zlib/src/inflate.c	625 – 1275	identical copy of zlib
third_party/rust/libz-sys/src/zlib/inflate.c	625 – 1275	identical copy of zlib
modules/zlib/src/inflate.c	625 – 1275	identical copy of zlib
security/nss/lib/zlib/inflate.c	625 – 1275	identical copy of zlib
third_party/rust/libz-sys/src/zlib/inflate.c	625 – 1275	identical copy of zlib
third_party/rust/libz-sys/src/zlib/infback.c	256 – 629	identical copy of zlib
security/nss/lib/zlib/infback.c	256 – 629	identical copy of zlib
modules/zlib/src/inflate.c	256 – 629	identical copy of zlib
security/nss/lib/zlib/infback.c	256 – 629	identical copy of zlib

100 lines long, which was likely because NiCad respected the function boundaries and separated clones there. Therefore, it was difficult to see that there were large duplicated code blocks between files from NiCad outputs. Although it has been claimed that only clones that respect the function boundaries are useful [1], [14] for refactoring, clones with larger granularity can help find large-scale structures of repositories and restructure them. We did not analyze the output of Duplo because of integer overflow bugs.

V. CONCLUSION AND FUTURE WORKS

In this paper, we introduced a new method for code-clone detection using an efficient algorithm of finding repeated substrings in a string, and we implemented a tool called CodeRepeat. Because our method does not involve either parsing or lexical analysis, it is language independent, easy to implement, and robust against complexities in the source code. Further, our method normalizes newlines, tabs, and sequences of white spaces into single white spaces, and it can detect clones with different newline positions or different indentation, which commonly arise when copy-pasting occurs. Our method always outputs exact matches and does not use any approximation, heuristics, or machine-learning based method. Its simplicity would improve developers' understanding of tool behaviors.

We applied BigCloneEval to CodeRepeat together with a baseline tool, Duplo, CCFinderX, and NiCad. The results showed that CodeRepeat found almost all Type-1 clones and a moderate number of Type-2 clones, although it is based on finding exact matches.

By applying large-scale open source repositories, we showed that our method is as fast as existing methods and that its memory consumption is within the capacity of a modern personal computer. Our tools are robust in real-world settings, whereas other tools often fail to process these repositories correctly. Furthermore, we demonstrated that our tool can find multiple instances of different versions of the same library, duplicated macro definitions, declarations, and table definitions. Such information is useful for maintaining, updating, and understanding large-scale code repositories.

Our study has several limitations. First, we did not measure the precision of CodeRepeat. Theoretically, the precision of CodeRepeat should be 100%, but a potential bug may lead to a reduction in this precision. Second, we did not fine-tune the configuration of baseline tools. For example, CCFinderX has an option for controlling the number of token types in the reported code clones. We observed that this option allows CCFinderX to find variable-definition level clones. Third, we only collected errors that appeared in the error messages. Internal errors that did not appear in the error messages may have led to incorrect processing. Finally, Duplo had a serious bug due to which wrong line numbers were given as output, thereby making its comparison to other tools could be unfair.

In future, we want to improve the algorithm to find the maximal repeats. Beller et al.'s algorithm [2] is asymptotically faster than Barenbaum et al.'s algorithm [4], and Beller et al.'s implementation reportedly consumes less memory. Using Beller et al.'s algorithm may allow us to tackle larger software repositories, such as the entire software packages of a Linux distribution. The output of CodeRepeat has a JSON format, which simplifies machine processing. However, the sheer number of repeats reported makes it difficult for developers to analyze. Thus, we need tools to analyze the output from CodeRepeat to help developers understand the structure of code repositories. One possible direction is a tool to find only inter-project clones with large granularity, which would help to track code evolution and source code traceability.

REFERENCES

[1] 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.

[2] T. Beller, K. Berger, and E. Ohlebusch, "Space-efficient computation of maximal and supermaximal repeats in genome sequences," *SPIRE 2012*, vol. 7608 LNCS, pp. 99–110, 2012.

[3] M. O. Kulekci, J. S. Vitter, and B. Xu, "Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 2, pp. 421–429, 2012.

[4] P. Barenbaum, V. Becher, M. H. A. Deymonnaz, and P. Heiber, "Efficient repeat finding in sets of strings via suffix arrays," *Discrete Mathematics & Theoretical Computer Science*, vol. 15, 2013.

[5] [Online]. Available: https://github.com/dlidstrom/Duplo

[6] S. Harris, 2018. [Online]. Available: https://www.harukizaemon.com/simian/

[7] 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.

[8] [Online]. Available: http://www.ccfinder.net

[9] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *ICPC 2011*, 2011, pp. 219–220.

[10] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *ICSE 2016*, 2016, pp. 1157–1168.

[11] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2236–2284, 2019.

[12] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: a token based large-gap clone detector," in *ICSE 2018*, 2018, pp. 1066–1077.

[13] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *ESEC/FSE 2018*, 2018, pp. 354–365.

[14] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," *ICSME 2015*, pp. 131–140, 2015.

[15] ——, "A survey on the evaluation of clone detection performance and benchmarking," *arXiv preprint arXiv:2006.15682*, 2020.

[16] D. Gusfield, "Algorithms on stings, trees, and sequences: Computer science and computational biology," *Acm Sigact News*, vol. 28, no. 4, pp. 41–60, 1997.

[17] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *ICSME 2016*, 2016, pp. 596–600.