# Chapter 27
# A Study of Code Clone Detection Techniques in Software Systems

**Utkarsh Singh, Kuldeep Kumar, and DeepakKumar Gupta**

## 1 Introduction

In the information technology industries, software development is not performed under idle conditions. It is a period-bound activity and prerequisites from the stakeholders can change at random. To satisfy the stakeholder's changing requirements, developers are required to speed up and complete the product improvement in the given time limit [1]. Working under such conditions, the developers generally copy-paste the code in which there is either no modifications or they do some minor modifications to the code by including, erasing or updating the code statements. Doing it at a specific degree does not affect the product, but the extreme utilization of the copy-paste approach degrades the quality of the software systems [2].

Replicating existing code parts and pasting them with or without alterations into various areas of the source code of a software system is a very common practice in software development [2, 3]. The replicated code fragments are called code clones and the process is called software code cloning. This sort of reuse approach of the current code may lead to bug propagation. A fault arising in one part of the code may arise in all the replicated sections of the code. To mitigate this problem, it is

U. Singh (✉) · K. Kumar · D. Gupta
Department of Computer Science and Engineering, Dr. B R Ambedkar National Institute of Technology, Jalandhar, Punjab 144011, India
e-mail: utkarshs885@gmail.com

K. Kumar
e-mail: kumark@nitj.ac.in

D. Gupta
e-mail: guptadk@nitj.ac.in

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2021

M. Dave et al. (eds.), *Proceedings of the International Conference on Paradigms of Computing, Communication and Data Sciences*, Algorithms for Intelligent Systems,
https://doi.org/10.1007/978-981-15-7533-4_27

very essential to locate each related code pieces throughout the source code and for these, there is a requirement of software code clone detection techniques [4].

In this paper, after reviewing existing works on software clones, we gathered and summarized investigations in the area of software code clone detection. We explored different code clone detection techniques, provided a brief description of various clone terminologies, code clone evolution, clone detection process, and detailed description of code cloning with its pros and cons. It assists the users in understanding the clone detection process and choosing the appropriate techniques for detecting a possible type of clones. The detection and analysis of such clones can help in refactoring and maintenance processes [5].

The left part of the paper is sorted out as follows: Basic terminologies utilized in the area of code clones are clarified in Sect. 2. Section 3 talks about the literature review. Having talked about the points of advantages and disadvantages of code clones in Sect. 4 have a descriptive brief of the clone detection techniques is provided in Sect. 5. An overview of code clone evolution is discussed in Sect. 6. Section 7 concludes the paper with a detailed depiction on future directions.

# 2 Clone Terminologies

This section discusses different terminologies that are used during software code clone detection.

## 2.1 *Clone Relation Terminologies*

Code clone detection techniques produce results as clone classes, clone pairs or both. A couple of code fragments is known as *aclone pair* when they have significant similarities between them. For example, consider the three code fragments, *Me1*, *Me2*, and *Me3* as given in Table 1, we have five clone pairs, <*Me1(e)*, *Me2(e)*>,

**Table 1** An example illustrating clone pairs and clone class

| Fragment *Me1* | Fragment *Me2* | Fragment *Me3* |
|---|---|---|
| intsqrt = 0, i = 0, b = 0;<br>while(i < n) {b = i * i;<br>sqrt = sqrt * b; i++;} (e) | intsqrt = 0, i = 0, b = 0;<br>while(i < n){b = i * i; sqrt =<br>sqrt*b; i ++;} (e) | … |
| if(sqrt > 0){sqrt = n +<br>sqrt;}<br>else{sqrt = 0;} (f) | if(sqrt > 0){sqrt = n + sqrt;}<br>else{sqrt = 0;} (f) | if(res < 0){res = m+res;}<br>else{res = 0;} (e) |
| … | while(sqrt > n){if(sqrt > 0){sqrt<br>= sqrt/n;<br>sqrt = sqrt + n;}} (g) | while(res > m){if(res > 0){res =<br>res/m;<br>res = res + m;}} (f) |

*<Me1(f), Me2(f)>*, *<Me2(f), Me3(e)>* , *<Me2(g), Me3(f)>* and *<Me1(f), Me3(e)>*. The equivalence relation between the code fragments is shown by similarity relation among them [3].

A clone class is a maximal arrangement of cloned code fragments in which any couple of the two code sections is similar to each other. For example, as shown in Table 1, we get a clone class *<Me1(f), Me2(f), Me3(e)>* where this three code fragments *Me1(f)*, *Me2(f)* and *Me3(e)* make a clone pairs with each other, respectively, and as a result, there will produce three clone pairs, *<Me1(f), Me2(f)>*, *<Me2(f), Me3(e)>* and *<Me1(f), Me3(e)>*.

## 2.2 Types of Clones

On the basis of syntactic and semantic similarities between code fragments, code clones can be separated into four types: exact clones, renamed clones, near-miss clones, and semantic clones [2]. *Exact clones* (*type 1 clones*) are code fragments that are the same except the white space and comments. *Renamed clones* (*parameterized or type 2 clones*) are code fragments that are syntactically identical comparative aside changes in identifiers, literals, types. *Near-miss clones* (*type 3 clones*) are code fragments that have been duplicated with further modifications such as proclamation insertions/deletions in addition to the changes in identifiers, literals, types, and formats. As shown in Table 2, code fragments in columns A and B, A and C, A and D form exact, renamed, and near-miss code clones, respectively. *Semantic clones (type 4 clones)* are code fragments that need not be similar at the code-level but perform similar operations. Table 3 gives an illustrative example of semantically similar code clones.

**Table 2** Examples of code fragments illustrating different types of syntactic code similarities

| Code fragment (A) | Exact code clone (B) | Renamed code clone (C) | Near-miss code clone (D) |
|---|---|---|---|
| float sub(float j1, float j2){float subtotal = 0.0;<br>subtotal = j1 − j2;<br>return subtotal;} | float sub(float j1, float j2)<br>{float subtotal = 0.0;<br>subtotal = j1 − j2;<br>return subtotal;} | float sub(float x1, float x2)<br>{floattotal = 0.0;<br>total = x1 − x2;<br>return total;} | float sub(float x1, float x2)<br>float x4 = x2;<br>{Float total = 0.0;<br>total = x1 − x4;<br>return total;} |

**Table 3** An example illustrating semantic similarity between code fragments

| Code fragment (A) | Code fragment (B) |
|---|---|
| void fibonacci(intnum) {int t1 = 0, t2 = 1, t3 = 0;<br>for (intpj = 0; pj < num; pj ++) {cout ≪ t1 ≪ " ";<br>t3 = t1 + t2;<br>t1 = t2;<br>t2 = t3;}} | intfibo(int num1){<br>if (num1 <= 1)<br>return num1;<br>return fibo(num1-1) + fibo(num1-2);} |

## 3 Literature Survey

After some time, there has been a broad arrangement of research works in the territory of clone detection. Kamiya et al. [6] proposed a token-based clone detection tool CCFinder for distinguishing type 3 clones. In the initial step, the source code is converted into a token-sequence. From that point, clone sets/clone classes are extricated from the token-sequence utilizing a postfix-tree-based sub-string matching algorithm. Yang et al. [12] introduced an abstract syntax tree (AST)-based approach for clone detection that uses the Smith-Waterman algorithm for similarity comparisons. They evaluated their proposed approach on more than five open-source Java projects and achieved a significant value of precision and recall.

Roy and Cordy [9] introduced an AST-text based hybrid approach for distinguishing function clones in software systems. As a matter of fact, text-based techniques discover clones with high precision and recall, yet once in a while, the distinguished clones do not relate to proper syntactic units. On the other hand, AST-based techniques see syntactical clones however tend as more heavyweight because of the requirement for the full parser and sub-tree comparison algorithm. The experiments show that parser-based techniques produce low recall. So, they joined these two strategies to beat their restrictions and utilized their advantages. They evaluated their hybrid method on more than 15 open-source Java and C projects. They made a benchmark that can be utilized to confirm the results of other clone detection tools as they delivered the outcome for each project individually.

Mayrand et al. [16] introduced a tool, *Datrix* that utilizes metrics-based approach for detecting exact and near-miss function clones in large software systems. They used 21 function metrics grouped into four points of comparison–name, layout, expressions, and control flow–which helped in deciding the cloning levels. They validated their approach by applying on two telemonitoring systems. They additionally introduced the ordinary scale of eight cloning levels. The level range starts from the exact copy to the distinct functions. They cited that the level-1 clones have fewer rates of false-positives as compared to level-3 clones which get expanded substantially.

Basit and Jarzabek [17] presented a data-mining techniques for detecting high-level clones, called as structural clones. They characterized the structural clones as repeated configurations of lower-level contiguous cloned code fragments (they called them as simple clones). They introduced the tool named Clone Miner that detects structural clones by first detecting the simple clones, and then incrementally detecting the higher-level structural clones by utilizing the idea of the frequent-closed itemset mining technique.

Marcus and Maletic [18] utilized the latent semantic indexing on the syntactical representation of the source code to detect semantic similarities between program structures. Latent semantic indexing is a vector-based statistical technique which is used to represent the meaning of all the identifiers and comments of the source code.

They considered comments as one of the important factors in detecting semantic clones. Hence, when there are no proper comments in the code, the method fails to detect the clones.

Kodhai and Kanmani [4] proposed a hybrid approach that uses 12 different metrics and textual comparisons for detecting clones in a software system. The proposed method has been applied to seven different C and Java projects, and has high precision and recall. The approach also uses less time as compared to the other parallel tools. Table 4 presents a comparative analysis of selected clone detection techniques.

## 4 The Rationale for Code Duplication

There are different reasons that may prompt the nearness of code clones within software systems. Various factors influencing software development processes such as changes in technology, certain requirements changes, strain to complete the work in time-limits force the designers to go for open non-appreciable development practices. Such practices may lead to the introduction of clones in software systems. Further, reusing existing code with or without modifications is one of the popular and straightforward techniques in component reuse, which leads to the presence of code clones within software systems.

Sometimes, clones may be introduced by the programmers unintentionally [2]. The utilization of a specific API/library typically needs a progression of function calls as well as other arranged groupings of commands. Use of similar APIs/libraries can introduce clones in a software. It might also happen that two engineers were associated with actualizing a similar sort of rationale, and in the end come up with a similar solution, resulting in code clones in the software. Difficulty in understanding a large software system also leads to copying the logic and the functionalities.

### *4.1 Advantages and Disadvantages of Clones*

Sometimes, clones are introduced by the programmers intentionally in a software system [3, 19]. First, cloning is among the quickest and easiest strategies for addressing the change in requirements. Further, if a programmer wants to quickly enhance the functionality of a system, it has only left with one way, i.e., reuse or using the abstract mechanism. Code segments that are used by programmers multiple times show that they can be usable code segments. As a result, one should add these usable segments into a library for future use. However, due to the method calls overhead, sometimes, programmers have to increasing efficiency so they use code duplications.

Besides having advantages of having code duplication, clones have a serious impact on software systems. They can influence the product quality, maintenance cost, and can likewise influence product development [20]. Due to the cloned code, it is possible that it will put a strain on the resources. It is because cloning will

**Table 4** Comparative analysis of selected clone detection techniques

| Author | Approach | Datasets | Tools/metrics | Targeted clone |
|---|---|---|---|---|
| Kamiya et al. [6] | Transformation of input source code into tokens followed by token-by-token comparison | C, C++, Java, COBOL (JDK, FreeBSD, NetBSD, Linux) | CCFinder | Syntactic code clones |
| Li et al. [7] | Based on frequent subsequence mining technique | Linux and FreeBSD | CP-Miner | Syntactic code clones |
| Lingxiao et al. [8] | Distinguishing similar sub-trees of the source code and with the help of this try to make a tree representation of the source code | Large codebases written in Java, C (Linux kernel, JDK) | Deckard | Syntactic code clones |
| Roy and Cordy [9] | A hybrid approach utilizing the advantages of Text-based and AST-based methods | Linux Kernel, Apache-httpd, and other open-source C and Java systems | Nicad | Exact clones, near-miss clones |
| Kodhai and Kanmani [4] | A lightweight hybrid approach combining textual analysis and metric-based comparison | Open-source C and Java projects | Clone manager | Function clones |
| Sajnani et al. [10] | Various code-blocks comparisons are expected to distinguish clones, as well as for token comparisons | Large inter-project repositories written in C, Java, and C# | SourcererCC | Exact clones, near-miss clones |
| Li et al. [11] | From the known code clone, we extract the tokens and non-clones to prepare a classifier, and after that utilizes the classifier to identify clones. | Large codebases written in C and Java | CCLearner | Code clones |

(continued)

**Table 4** (continued)

| Author | Approach | Datasets | Tools/metrics | Targeted clone |
|---|---|---|---|---|
| Yang et al. [12] | Tree-based detection techniques. The extracted abstract syntax tree is used for sub-tree comparison | Open-source projects developed in Java, Ant, Tomcat, JDK, ANTLR, DNSJava | – | Function clones |
| Wang et al. [13] | A specific matching index is generated for finding the similarity matching in the code uses an asymmetric similarity coefficient | Cook, Redis, PostgreSQL, Linux 1.0 in C projects. J DK, openNLP, Maven, Ant in Java project | CCAligner | Syntactic code clones |
| Luan et al. [14] | Use query code-snippet to search similar code | Java and Android-based applications | Aroma | Code search engine |
| Singh and Kumar [15] | Metric-based technique | Abyss v0.3, Bison 2.4, Apache-httpd 2.2.8 | – | Function clones |

increase resource usage and degrades the quality of the software. Further, the part of the code fragments which are copied may have a bug. Copying such buggy code fragments can lead to the probability of propagation of bugs in the software [7].

## 5 Clone Detection Process and Techniques

In this section, we discuss the clone detection process and various techniques for detecting code clones in detail. We discuss the distinguishing properties of different techniques thereafter.

### 5.1 *Clone Detection Process*

There are a few methodologies for identifying code clones; a few methodologies use source code directly, some apply a transformation to convert the source code into an appropriate form for the postprocessing of the output produced [2]. Although there are different detection techniques, Fig. 1 presents a general overview of the process followed by different clone detection techniques.

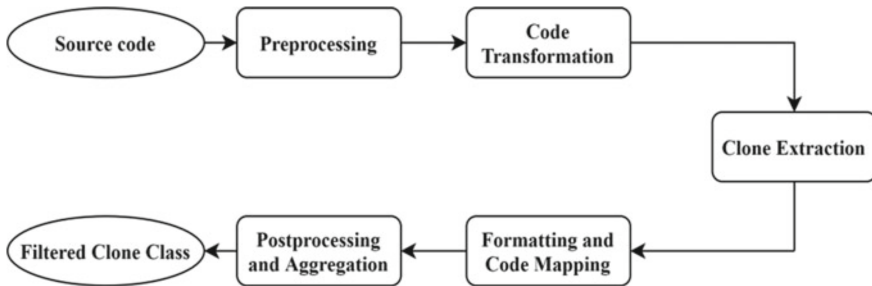


**Fig. 1** An overview of the clone detection process

**Preprocessing**. During the advancement of the software product, sometimes, developers uses remarks, whitespaces, comments and numerous other naming conventions which have nothing to do with the functioning of the product but they use for the better comprehension and intelligibility; it will work the same if these are not used [21]. In this step, such irrelevant parts and code artifacts that are of not any relevancy for clone detection are removed.

**Code Transformation**. The preprocessed source code is then changed into a suitable representation with a goal that the detection process can be applied to it and the machine can perform this detection process efficiently. Tokenization of source code, parsing of code, generating abstract syntax trees/program dependency graphs, calculating metrics, etc., are some of the transformation activities that can be applied to the preprocessed source code [2, 3].

**Clone Extraction**. After performing the transformation, the source code is in an appropriate form on which a detection algorithm can be applied. In this step, the transformed units are compared to all other transformed units to find the cloned matches. For better performance and to detect clones closely, transformed units are merged into bigger units so that the precision of the techniques can be improved. The output can be in any form, i.e., clone pair or clone classes per the detection technique followed.

**Formatting and Code Mapping**. After getting a list of clone pairs, it is mapped on the original source code. Line numbers or proper references are provided on the original source code with reference to the clone pairs.

**Postprocessing and Aggregation**. Since there are no automated verification methods available, manual verifications are important to discover false-positive clones. In this step, verification of detected clones is carried out, and after filtering out the false-positives, it is represented using proper visualization technique so that the output is easy to understand and can be visualized easily [3]. In aggregation, we reduce the data amount for analysis. Clone sets are accumulated into clone clusters, classes, or clone groups.

### *5.2 Clone Detection Techniques*

In large software systems, to identify code clones, there should be a need for detailed knowledge of the orientation and its internal structure such as programming language, file extensions, etc. Likewise, to discover all the code clones, it is required to compare each code part and all other accessible code pieces, which is a costly process in terms of the calculation performed by the system to accomplish this [2]. So, there are different types of techniques depending on the alternative ways to deal with detecting code clones in a software system.

**Text-based Detection Techniques**. It deals with sequences of code or strings used. In code fragments, each statement is termed as a sequence of text/string [2]. To detect code clones, two code parts are matched based on the similarities of text/string sequences. After the detection, the results can be returned as a clone sets or clone class. Sometimes, the original source code is not in an appropriate form for comparison, so as to make it suitable for comparison, we have to apply some transformations/changes or filtering to the source code [21].

**Token-based Techniques**. In this technique, the whole source code is changed to sequences of tokens using reasonable parser or various transformations. Looking at the whole, text/strings can be expensive, which gets improved in the token-based technique since it changes the whole source code into tokens. It makes it robust and simple for comparison. After tokenization, similar token-subsequences are identified by using various algorithms. These similar token-subsequences correspond to clone pairs or clone classes.

**Tree-based Techniques**. In this technique, the source code is converted into a tree-like structure where nodes represent to program entities (such as code fragments, methods, etc.) and edges represent connections among program entities. During detection, a similar sub-tree is looked into the whole tree with some suitable tree-searching algorithms. Postprocessing is applied to return the clone pairs or clone classes on the detected similar sub-trees [3].

**Program Dependency Graph-based Techniques**. In this approach, semantic information is represented in the forms of data flow and control flow among the components of the software system. For detecting code clones, appropriate sub-graph matching algorithms are used and isomorphic diagrams are searched [2]. It has several benefits in terms of any statements addition, deletion, or re-ordering.

**Metrics-based Techniques**. In this technique, different metrics such as lines of codes, numbers of edges/vertices in the control flow graph representation, cyclomatic complexity, etc., are calculated for the program entity (a unit of comparison used for clone detection). At that point, program entities having similar metrics are returned as clone pairs/classes.

### 5.3 Discussion

Collectively, we analyze different clone detection techniques with respect to their distinguishing properties.

During transformation, string-based approaches remove whitespaces and comments (and sometimes, it uses normalizations), token-based approaches transform source code to tokens, tree-based approaches parse the source code to AST, PDG-based approaches convert the code to a PDG, and metric-based approach generates metrics values. In code representation, string-based approaches generate filtered or normalized source code, token-based approaches generate a sequence of tokens, tree-based approaches generate abstract syntax trees of the program dependent on its structure and the code test, PDG-based approaches generate a set of PDGs for the procedure of program and metrics-based generate set of metrics values [20]. During comparison granularity, string-based approaches compare lines or tokens of line, token-based approaches compare only tokens, tree-based approaches compare tree node, PDG-based approaches compare PDG node, and metrics-based approaches compare metrics values use for each method/block.

Text-based techniques are lightweight and can distinguish accurate clones with high recall. Token-based techniques are quick in detecting a huge number of clones with high recall yet flopped precision. Parser-based methods are commendable in detecting syntactic clones with high precision. Regardless, they give low recall but the detected candidates can be used by the developers in refactoring for the clone management [3]. Metric-based methods are extremely efficient in detecting both syntactic and semantic clones. PDG-based methods can discover progressively semantic clones. These restrictions in existing strategies give a way to examine mixture or combination of the detection techniques so as to defeat them [20].

## 6 Code Clone Evolution

During the evolution of software systems, designers regularly roll out certain improvements in existing code and use them directly. So, if a model/tool can recognize all such code parts, at that point, it will be very useful in the maintenance process. Since a large software system is being followed for a decade, several versions have been launched over time. Different analysts have worked out to distinguish how the clones evolve in various versions of a software system.

Machine learning models such as autoregressive integrated moving average model (ARIMA), back propagation neural network, and multi-objective genetic algorithm neural networks (MOGA-NN) have been applied to find the advancement of code clones across different versions of a software system [22]. Aside from the machine learning models, there are different strategies pursued in the past to predict clone evolution [23, 24, 25].

Kim et al. [23] used code snippets' text and locations to analyze clone evolution. Code text shows an internal description of the code which a clone detector uses for comparison purpose. The code location is utilized to follow the code snippets across all versions of software systems. To inspect how much the content of code snippets has changed over the variants, they have utilized a text similarity functions that compute the textual similarity between the writings of code snippets across the versions. Thummalapenta et al. [24] recognized all clone classes in a single version, and then they discovered all the code clones groups that change across all versions of a software system.

A methodology that uses information from a unique AST to find clone evolution was proposed by Bakota et al. [25]. Before matching code fragments across the versions of a software system, they first eliminated all possible matches of code fragments whose AST sub-tree representations have different types of the root node. For each remaining pair of the code fragments, they calculated a similarity metric that is an aggregation of five weighted metric values.

## 7 Conclusions and Future Scope

This paper puts a light on all the types of semantic and syntactic clones and various clone detection techniques for detecting the clones. There are a lot of factors that influence software development processes such as changes in technologies, certain requirements changes, and strain to complete the work in time-limits force the designers to go for open non-appreciable development practices. Such practices may lead to the introduction of clones in software systems. Clones have a serious impact on software systems; they can influence the product quality, maintenance cost, and can likewise influence product development. Their detection can help in decreasing maintenance costs, improving project comprehension, and controlling code modifications. Since there are various individual clone detection techniques with certain advantages and disadvantages in their calculation, another way for improving this calculation is by combining different clone detection techniques. It produces an outcome with higher precision and recall. This area has still a great deal of future scope for specialists to take a shot at code clone family, examining potential clones from the actually detected clones, recognizing type 4 (semantic) clones with more precision and accuracy, refactoring of clones, and breaking down the significance of clone detection in maintenance which is the most expensive phase of software development life cycle.

**Acknowledgements** This work has been supported by a research grant from the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India under the Early Career Research Award Scheme.

## References


1. Nurmuliani N, Zowghi D, Powell S (2004) Analysis of requirements volatility during software development life cycle. In: 2004 Australian software engineering conference. IEEE, pp 28–37. https://doi.org/10.1109/ASWEC.2004.1290455
2. Roy CK, Cordy JR (2007) A survey on software clone detection research, vol 541, issue no. 115. Technical Report 541, Queen's University at Kingston, pp 64–68
3. Rattan D, Bhatia R, Singh M (2013) Software clone detection: a systematic review. Inf Softw Technol 55(7):1165–1199
4. Kodhai E, Kanmani S (2014) Method-level code clone detection through LWH (Light Weight Hybrid) approach. J Softw Eng Res Dev 2(1):1–29. https://doi.org/10.1186/s40411-014-0012-8
5. Tsantalis N, Krishnan GP (2013) Refactoring clones: a new perspective. In: 7th international workshop on software clones. IEEE, pp 12–13. https://doi.org/10.1109/IWSC.2013.6613035
6. Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670. https://doi.org/10.1109/TSE.2002.1019480
7. Li Z, Lu S, Myagmar S, Zhou Y (2006) CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE Trans Softw Eng 32(3):176–192. https://doi.org/10.1109/TSE.2006.28
8. Lingxiao J, Misherghi G, Zhendong S, Glondu S (2007) DECKARD: scalable and accurate tree-based detection of code clones. In: 29th international conference on software engineering (ICSE), pp 96–105. https://doi.org/10.1109/ICSE.2007.30
9. Roy CK, Cordy JR (2009) Near-miss function clones in open source software: an empirical study. J Softw Maintenance Evolu Res Pract 22(3):165–189. https://doi.org/10.1002/smr.416
10. Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV (2016) SourcererCC: scaling code clone detection to big-code. In: Proceedings of the 38th international conference on software engineering. ACM Press, New York, New York, USA, pp 1157–1168. https://doi.org/10.1145/2884781.2884877
11. Li L, Feng H, Zhuang W, Meng N, Ryder B (2017) CCLearner: a deep learning-based clone detection approach. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 249–260. https://doi.org/10.1109/ICSME.2017.46
12. Yang Y, Ren Z, Chen X, Jiang H (2018) Structural function based code clone detection using a new hybrid technique. In: 42nd Annual computer software and applications conference, vol 1. IEEE, pp 286–291. https://doi.org/10.1109/COMPSAC.2018.00045
13. Wang P, Svajlenko J, Wu Y, Xu Y, Roy CK (2018) CCAligner. In: Proceedings of the 40th international conference on software engineering. ACM Press, New York, New York, USA, pp 1066–1077. https://doi.org/10.1145/3180155.3180179
14. Luan S, Yang D, Barnaby C, Sen K, Chandra S (2019) Aroma: code recommendation via structural code search. Proc ACM Program Lang 3:1–28. https://doi.org/10.1145/3360578
15. Singh MK, Kumar K (2020) Scalable and accurate detection of function clones in software using multithreading. In: Jarzabek S, Poniszewska-Marańda A, Madeyski L (eds) Integrating research and practice in software engineering. Springer International Publishing, Cham, pp 31–41. https://doi.org/10.1007/978-3-030-26574-8_3
16. Mayrand J, Leblanc C, Merlo E (1996) Experiment on the automatic detection of function clones in a software system using metrics. In: International conference on software maintenance, vol 96. IEEE, pp 244–253. https://doi.org/10.1109/ICSM.1996.565012
17. Basit HA, Jarzabek S (2009) A data mining approach for detecting higher-level clones in software. IEEE Trans Softw Eng 35(4):497–514. https://doi.org/10.1109/tse.2009.16
18. Marcus A, Maletic JI (2001) Identification of high-level concept clones in source code. In: Proceedings of 16th annual international conference on automated software engineering. IEEE, pp 107–114. https://doi.org/10.1109/ASE.2001.989796
19. Kapser CJ, Godfrey MW (2006) Supporting the analysis of clones in software systems. J Softw Maintenance Evol Res Pract 18(2):61–82. https://doi.org/10.1002/smr.327

20. Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591. https://doi.org/10.1109/TSE.2007.70725
21. Koschke R (2007) Survey of research on software clones, Internat. Begegnungs-und Forschungszentrum für Informatik
22. Pati J, Kumar B, Manjhi D, Shukla KK (2017) A comparison among ARIMA, BP-NN, and Moga-NN for software clone evolution prediction. IEEE Access 5:11841–11851. https://doi.org/10.1109/ACCESS.2017.2707539
23. Kim M, Sazawal V, Notkin D (2005) An empirical study of code clone genealogies. In: European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13). ACM, pp 187–196. https://doi.org/10.1145/1081706.1081737
24. Thummalapenta S, Cerulo L, Aversano L, Di Penta M (2010) An empirical study on the maintenance of source code clones. Empirical Softw Eng 15(1):1–34. https://doi.org/10.1007/s10664-009-9108-x
25. Bakota T, Ferenc R, Gyimóthy T (2007) Clone smells in software evolution. In: IEEE international conference on software maintenance (ICSM). IEEE, pp 24–33. https://doi.org/10.1109/ICSM.2007.4362615