Journal of Information & Optimization Sciences
ISSN 0252-2667 (Print), ISSN 2169-0103 (Online)
Vol. 47 (2026), No. 7, pp. 2713–2722
DOI : 10.47974/JIOS-2326

Multiphase code clone detection using NSGA-II

Neha Saini [†]
Department of Computer Science
Government College, Chhachhrauli (Yamuna Nagar)
Chhachhrauli
Yamuna Nagar 135103
Haryana
India

Shalu *
Department of Computer Science & Technology
Manav Rachna University
Faridabad 121004
Haryana
India

Upasna Joshi [§]
School of Computer Science Engineering & Technology
Bennett University
Greater Noida 201310
Uttar Pradesh
India

Ankit Gambhir [‡]
Department of Computer Science & Engineering
Trinity Institute of Professional Studies
Guru Gobind Singh Indraprastha University
Greater Noida 201310
Uttar Pradesh
India

* *ORCID-ID:* **0000-0002-5516-0357**

[†] *E-mail:* **profnehasaini@gmail.com**
* *E-mail:* **singshalu2609@gmail.com (Corresponding Author)**
[§] *E-mail:* **upasnajoshi19@gmail.com**
[‡] *E-mail:* **er.ankit.gambhir@gmail.com**

Aniket Singh [@]
Department of Computer Science & Technology
Manav Rachna University
Faridabad 121004
Haryana
India

Mohd Anas Khan [#]
Department of Computer Engineering
Jamia Millia Islamia
Jamia Nagar 110025
New Delhi
India

Abstract

This duplication of source code is termed as code cloning. This is the most widely used source code reusing technique in software development. When one bug is discovered in a single section of a code, there is need to test all the other sections that are the same code to test the bug. Consequently, the given cloning technique can lead to the spread of some issues, which significantly affects maintenance expenses. Code clone detection (CCD) seems to be an active field of research. CCD method based on semantic similarity is applicable in software engineering (e.g., software evolution, software reuse, etc). The classical code clone detection systems emphasize semantic similarity to a lesser degree and more on the syntax similarity. This leads to ignoring of candidate code that has similar meanings in semantics. To address this issue semantic similarity-based code clone detection algorithm named as Multiphase Code Clone Detection using non-dominated sorting algorithm (NSGA- II) has been put forward which has given a type-1 clone detection of 97.37% and a type-2, type-3, and type-4 clones detection of 96.70%, 95% and 92.50% respectively.

Subject Classification (2020): *68N01, 68N30, 68Q25, 62P20.*

Keywords: *Code clone, NSGA, Clone type, Clone size software maintenance, Software metrics, CCD.*

1. Introduction

In computer programming, it is common to duplicate some code by coping and pasting it into a new section of code either unchanged or with few or no changes done. This procedure is termed code cloning, and the replicated code is designated as a code clone (CC). In case an error is discovered in a particular part of the code, then the identical error should be corrected in all the copied areas. Therefore, it is essential to identify all pertinent components inside the source code. The cloned code constitutes nearly 2050 percent of large software systems according to different research reports [1], [2]. Cloning has many advantages, but it is also harmful in many aspects. They also have the potential

[@] *E-mail:* **aniketsingh@mru.edu.in**
[#] *E-mail:* **anas.cse786@gmail.com**

to lead to the spread of bugs and this increases the cost of software maintenance to astronomical levels. The detection of the software clones became an active research topic due to all these maintenance problems. A variety of approaches and tools have been established to identify code clones. The most prevalent methods include text-based, token-based, tree-based, metric-based, semantic-based, and hybrid-based approaches [3], [4].

CCD is a crucial endeavour in software engineering for identifying the similarity of code snippets. The NSGA-II [5] is one of the widespread methods & multi-objective optimization approach that are used to detect code clones effectively using many criteria at the same time. NSGA-II algorithm [5] has its foundation on the principle of Pareto dominance that dictates that one solution is superiority over another solution in at least one objective and not inferior in any other objective. The algorithm randomly produces an initial set of solutions and then advances them by undergoing the process of selection, crossover, and mutation. These solutions are classified into various levels according to the Pareto dominance connections between the solutions and the most appropriate solutions in each level are picked to make the following generation. This procedure will be continued until a reasonable approximation of the pareto front is obtained [6].

To implement NSGA-II for identifying code clones, the objectives must be established for optimisation. In this case, there are two goals to be set that are similarity and size. The similarity and the size objectives are to be minimized at the same time, the similarity is used to compute the level of similarity of two pieces of code and size is used to compute the size of the piece of code. One can measure similarity using the similarity objective, such as the token-based or tree-based similarity and measure the size using the size of tokens or lines of code.

Immediately the objectives have been completed, a pool of random solutions can be generated and their fitness evaluated in reference to the objectives as well as apply the NSGA-II algorithm will also be utilized in development of the solutions to obtain a pool of Pareto-optimal solutions. The algorithm will take the form of the result table with the Pareto-optimal solutions and the similarity and size value of the latter. The solutions in result table are ranked as per Pareto dominance relationships wherein the solution of the first row is the best solution that is not dominated by any other solution.

The paper will begin with the basic introduction of code clones, other methods and approaches of code clone detection and the description to NSGA II algorithm. Then, the entire NSGA II -based code clone detection framework is presented. This section is followed by the result analysis and the comparison of the implemented framework with the old methods to be used to detect software clone is given. The researchers conclude the research paper by giving the end, and the directions where the research may be carried out in the future are presented.

2. Literature Review

This subsection will converse the existing methodologies that have been involved in the detection of code clones. Ďuračík, Kršák, and Hrkút [17] study reveals that the majority of code plagiarism detection systems exhibit limits regarding the diversity of source codes they can effectively identify [7]. Moreover, CCD methods that depend on semantic code clones, often fail to provide heuristic solutions to the problems of rearrangement of statements, reversal of predicates of control, and the addition of extraneous statements, which may cause operational inefficiency [8]. As a way of overcoming the shortcomings, Tekchandani, Bhatia, and Singh [18] proposed a new method of using data flow analysis, liveness examination and reaching definitions, to detect syntactic clones in a program [9].

Wan et al. [10] proposed k-means clustering method as a successor to the threshold-based cutoff stage in clone identification. Scalability issues had already been solved in previous studies of clone detection (CD). Consequently, they advocate for the implementation of robust Type-3 clone detection techniques by practitioners in multiple software platforms. Their primary objectives are performance and usability improvement. In the initial configuration, k-means is employed to ascertain the anticipated number of clusters. The experimental results demonstrate that utilising the k-means algorithm produces a performance improvement of 13%.

Kumar, Reddy, and Govardhan [11] proposed a straightforward yet efficient method for differentiating between exact and near-miss replicas of source code, utilising Abstract Syntax Trees (ASTs). Code clone identification is not just useful in the development of more structured pieces of code but also in the identification of domain-specific concepts and how they should be idiomatically implemented.

3. Proposed Multiphase NSGA II Code Clone Detection Framework

The framework proposed to be used in code clone detection has 2 broad stages. The code is first translated into an intermediate form in the first phase. The result of the former phase serves as the input of the latter stage in identifying the code clones. The verification and validation of the framework is discussed in the next sub sections.

a. *Code Archive*

The installation of the given framework necessitated 551 open-source applications from several GitHub sources. The next phase involved the direct incorporation of category 1, 2, 3, and 4 CC within existing 551 codes. Four Java engineers introduced four deliberately injected CC of various lengths and logic into the system. The four programmers had extensive training prior to administering the clones. The evaluation findings of one programmer are compared with those of the other two programmers to ensure accuracy. To test the validity of the discovered clones, the programmer selected some of the clones placed randomly and tested whether they were identified by the

programmer. The programmer was notified in case they were. The whole procedure was completed in a year and a half [12].

b. *Phase 1: Intermediate Representation of Code.*

AST have been utilised to describe code files as intermediate representations due to its structural depiction of the code, encompassing all information from the source code. They can also recognise semantic clones.

c. *Phase 2: Detection of similarity.*

An adequate (large recall and accuracy) code similarity detection algorithm has been used after the codes to be analyzed are represented in an appropriate form. To the best of our understanding, we are the first to apply the non-dominated genetic sorting feature in ASTs with a matching score. The extensive codes might generate ASTs that require significant memory owing to the considerable depth of the resulting tree.

Clones of the same code are applied in other sections of large code systems. Consequently, we resolved to reorganise one of the programs by rearranging related segments within the original file. Previous research has mainly focused on determining the similarity of code snippets on an individual basis. We introduce a new method of finding similarity of code in a code file and not a code snippet. Moreover, there are not many known strategies in detecting similarity of code files, but the proposed study is aimed at big code systems having an effective similarity detection strategy. Two subtrees are matched with the parameter matching score in equation 1. Following is the pseudo code of the proposed framework:

Matching Score: The resemblance of the two code segments pertaining to a subtree of the Abstract Syntax Tree, given two code files., e.g., cf1, cf2 consists of the following equation:

$$\text{Match (Subtree cf1, Subtree cf2)} = 2(Sh/(2Sh+Le+Ri) \quad (1)$$

Where Sh signifies the intensity of shared nodes in subtrees ST1 and ST2; [Le:[t1, t2, … tn] , Ri [t1, t2, …tn]

Pseudocode for NSGA II based Multiphase Code Clone Detection Framework

1. Clone identification input code files cf1 and cf2.
2. Set S.Thres = §
3. Given both cf1 and cf2 (l1, l2,……..,ln), produce token sequence TScf1 and TScf2.
4. Initialize Sim_Index =0
5. For (TEi, TEj) in (TScf1 and TScf2): Tree1= Construct.Tree (TEi) Tree2 = Construct.Tree(TEj) Sim = Compute Sim Index(Tree1,Tree2)
6. Find (If(Tree1=Tree2)) According to S.Thres i.e (Sim_Index> §) Return Clone Classes CCi (Where i=1 to n) Else Exit(): and Return No Clone Found
7. Apply non-dominating sorting algorithm to identify Pareto fronts of each Clone Class with the help of crowding distance to determine whether a solution exists which is better than this frontier.
8. When better solution identified: Update CCi.

4. Result and analysis

Code clone detection is conducted via NSGA II. The output of CCD using NSGA II is a collection of non-dominated solutions that illustrates the trade-offs among the objectives. The CCD objectives are number of detected clones, clone size, similarity of the clones and false positives/negatives. The NSGA II algorithm can be employed to identify the set of non-dominated solutions, wherein no other solution in the set outperforms in all objectives. The output of the CCD by means of NSGA II is measured by various performance measures.

The proposed framework is evaluated using 551 distinct Java code snippets sourced from the GitHub repository. Mean accuracy is obtained after testing all the 551 codes using a formula stated in the Equation 2. The framework exhibited an average accuracy of 95.37% in detecting all four types of clones, as illustrated in Figure 1. It is a remarkable CC detection accuracy.

$$MAcc = NCDet/TC \quad (2)$$

Where MAcc is the Mean Accuracy, NCDet is the count of detected clones and TC is the total count of clones present.

The run time of proposed algorithm is O(m/2) to represent the intermediate code and O(W/2) log(W/2) to match code where m is the number of lines of code in the code and w is the number of nodes in NDS-AST. The achieved accuracy of type-1 CD is 97. 37. Most type-1 clones can be well identified by the framework. Despite the fact that the proposed framework has demonstrated positive outcomes in the detection type-2 clone, the resulting accuracy of the type-2 CD is lower than that of type-1 CD. As it is possible to see in Figure 1, the accuracy with which it detects type-2 clones is 96.6%.

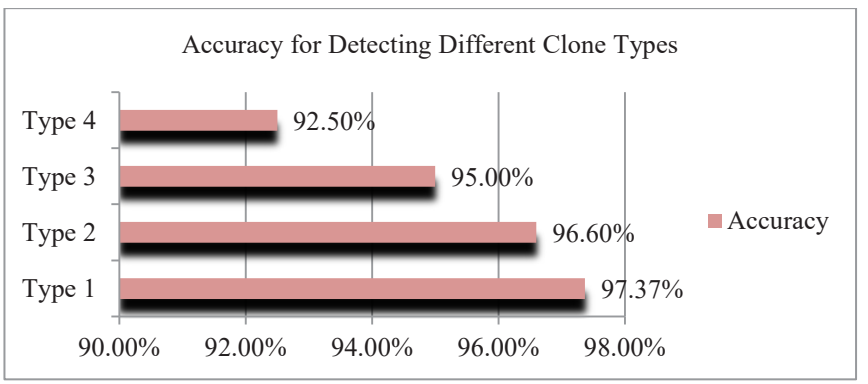

Figure 1
Accuracy in Identifying Various Categories of Clones

Type-3 clones are more challenging to identify than type-1 and type-2 clones. The identification accuracy is similarly inferior compared to type-1 and type-2 clones. Figure 1 demonstrates that the detection accuracy of type-3 clones is 95%. Type 4 clones, predicated on semantic code resemblance are the most

difficult to identify and sustain. Figure 1 illustrates that the suggested framework achieves an accuracy of 92.50% in detecting type-4 clones.

Figure 2 depicts the precision and recall scores for identifying various types of clones. Figure 2 indicates that the proposed design achieved the maximum precision and recall in type 1 CD, followed by type 2, type 3, and type 4. Type-1 clone identification exhibits precision and recall rates of 97.6% and 95%, respectively, whereas Type-2 demonstrates rates of 96.5% and 93%, which are inferior to those of Type-1. The proposed model demonstrated a precision of 96% and a recall of 91% in type-3 CD. In type-4 clone identification, the precision was 95.5% and the recall was 88%.

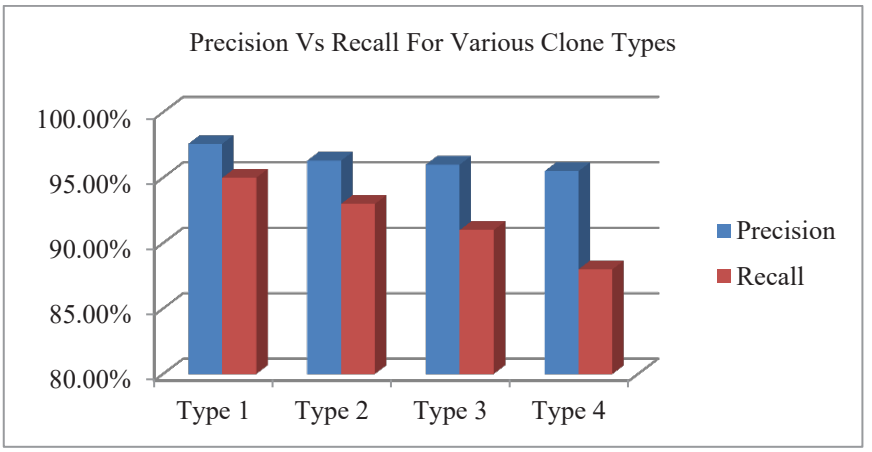

Figure 2
Precision Vs Recall for Different Types of Clones

4.1 *Comparative Analysis*

Table 1 delineates the results of the comparison between the proposed NSGA II algorithm and the current optimal approaches. A comparative table has been established to evaluate the performance indicators of the suggested algorithm against the prevailing models outlined in Table 2. The data indicated that the proposed model exhibits enhanced performance relative to the other models due to the performance metrics.

Table 1
Comparison in terms of clone detected

Benchmark	Deckard %age CD	Twin-Finder %age CD	SSA- HIAST %age CD	Multiphase CCD using NSGA II
bzip2	32.5	62.15	68.75	68.80
hmmer	27.19	59.55	62.24	63.00
Gzip	9.57	40.15	39.84	40.20
Man	14.74	49.08	55.47	55.55

Table 2
Evaluation of Precision and Recall

Study	KLOC	Precision (%)	Recall (%)
CCFinder [13]	24	98	92
Duploc [14]	25	89	86
DP matching [15]	29	75	84
Autoencode [16]	32	80	77
Proposed NSGA II algorithm	34	99	95

5. Conclusion

In recent years, CCD has been implemented using NSGA-II, a multi-objective optimisation technique. The approach will represent code clones as chromosomes and employ NSGA-II to identify the optimal solutions that simultaneously reduce the incidence of false positives and false negatives. Code clone detection using NSGA-II has been proved to be effective in various research. It has been demonstrated to exhibit superior accuracy in precision, recall, and F1-measure relative to conventional CD techniques. Conclusively, NSGA-II is a potential method of code clone detection that has a number of benefits over the traditional methods. The performance indicators of the developed model of code clone detection have been computed and compared with the already existing model provided by different authors in literature in this paper. The proposed approach in nearly all the cases performed considerably better compared to the current clone detection algorithms. In addition, the proposed solution has been tested using a total of 551 publicly available python codes on the GitHub repository and has shown to be effective. The outcomes have been measured in regard to the false positives and number of clones that have been discovered among other factors. The breadth of this study will be broadened to include additional computer languages such as Python, R, and C, as the existing framework is limited to the Java programming language. Along with identification and prioritisation of feasible clones, this framework may be strengthened further with the use of clone eradication strategies.

Conflicts of Interest: The authors declare no conflicts of interest.

References

[1] A. Sheneamer, S. Roy, and J. Kalita, "A detection framework for semantic code clones and obfuscated code," *Expert Systems with Applications*, vol. 97, pp. 129–143 (2018), doi: 10.1016/j.eswa.2017.12.040.

[2] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199 (2013), doi: 10.1016/j.infsof.2013.01.008.

[3] S. K. Panda, S. Mohapatra, and S. Das, "Artificial neural network-based virtual machine allocation in cloud computing," *Journal of Discrete Mathematical Sciences and Cryptography,* vol. 24, no. 6, pp. 1709–1722 (2021), doi: 10.1080/09720529.2021.1878626.

[4] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC and SourcererCC-I: Tools to detect clones in batch mode and during software development," in *Proc. 38th Int. Conf. Software Engineering Companion* (ICSE-C), pp. 441–444 (2016), doi: 10.1145/2889160.2889165.

[5] P. Vijai and P. Bagavathi Sivakumar, "A hybrid multi-objective optimization approach with NSGA-II for feature selection," *Decision Analytics Journal*, vol. 14, art. no. 100550 (2025), doi: 10.1016/j.dajour.2025.100550.

[6] E. Elabd, W. M. Ead, and Shalu, "An optimized differential private stochastic gradient descent (DP-SGD) approach to combat membership inference attacks in neural networks," *Int. J. Inf. Technol.,* vol. 17, no. 4, pp. 2369–2374 (2025), doi: 10.1007/s41870-025-02442-y.

[7] M. Karnalim and Simon, "Current trends in source code analysis, plagiarism detection and issues of analysis big datasets," *Procedia Computer Science,* vol. 124, pp. 329–338 (2017), doi: 10.1016/j.procs.2017.12.162.

[8] S. Singh and D. Singh, "A bio-inspired VM migration using re-initialization and decomposition based-whale optimization," *ICT Express,* vol. 9, no. 1, pp. 92–99 (2023), doi: 10.1016/j.icte.2022.02.003.

[9] S. K. Saini, H. Sajnani, and C. V. Lopes, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis," *The Journal of Supercomputing,* vol. 73, no. 11, pp. 4997–5026 (2017), doi: 10.1007/s11227-016-1832-6.

[10] B. Wan, S. Dong, J. Zhou, and Y. Qian, "SJBCD: A Java code clone detection method based on bytecode using Siamese neural network," *Applied Sciences*, vol. 13, no. 17, art. no. 9580 (2023), doi: 10.3390/app13179580.

[11] G. A. Kumar, D. C. R. K. Reddy, and D. A. Govardhan, "Software code clone detection using AST," *International Journal of P2P Network Trends and Technology,* vol. 9, no. 4, pp. 15–19 (2014), doi: 10.14445/22492615/IJPTT-V9P407.

[12] A. Yasmin, R. Venkatesan, and K. Gaverchand, "Enhancing the randomness of a symmetric cryptographic technique based on linear 1-D cellular automata," *Journal of Discrete Mathematical Sciences and Cryptography,* vol. 28, no. 1, pp. 185–204 (2025), doi: 10.47974/JDMSC-2133.

[13] 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), doi: 10.1109/TSE.2002.1019480.

[14] H. Liu and Z. Ma, "Detecting duplications in sequence diagrams based on suffix trees," in *Proc. Asia-Pacific Software Engineering Conf.,* pp. 269–276 (2008).

[15] M. Gabel, L. Jiang, and Z. Su, "Challenging cloning related problems with GPU-based algorithms," in *Proc. 4th Int. Workshop on Software Clones,* pp. 1–7 (2010), doi: 10.1145/1808901.1808905.

[16] W. Rahman, F. Pu, and X. Jia, "Clone detection on large Scala codebases," in *2020 IEEE 14th International Workshop on Software Clones* (IWSC), pp. 38–44 (2020), doi: 10.1109/IWSC50091.2020.9047640.

[17] M. Ďuračík, E. Kršák, and P. Hrkút, "Current trends in source code analysis, plagiarism detection and issues of analysis big datasets," *Procedia Engineering*, vol. 192, pp. 136–141 (2017), doi: 10.1016/j.proeng.2017.06.024.

[18] R. Tekchandani, R. Bhatia, and M. Singh, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis," *The Journal of Supercomputing*, vol. 74, no. 9, pp. 4199–4226 (Sep. 2018), doi: 10.1007/s11227-016-1832-6.

Received March, 2026