Int. j. inf. tecnol.
https://doi.org/10.1007/s41870-024-01868-0

ORIGINAL RESEARCH

Test case prioritization based on fault sensitivity analysis using ranked NSGA-2

Kamal Garg[1,2] · Shashi Shekhar[2]

Received: 5 January 2024 / Accepted: 8 April 2024

Abstract This paper discusses regression testing in software maintenance, focusing on test case prioritization to verify modifications to software functionality efficiently. The primary goal is to rank test cases, prioritizing those covering more code or faults with minimal execution time. The challenge lies in prioritizing numerous test cases generated during development and maintenance. Various algorithms, including greedy approaches and meta-heuristic techniques, address this challenge. The paper introduces a ranking-based non-dominated sorting genetic algorithm (NSGA-2) for test case prioritization, emphasizing cases sensitive to faults caused by modifications or new functionality. Historical data is prioritized, with key objectives including the sensitive index, execution cost, and average percentage of fault detection (APFD). The proposed model is tested on handcrafted and benchmark Java-based applications, comparing its performance to state-of-the-art algorithms in test case prioritization.

Keywords Regression testing · Test case prioritization · Multi-objective optimization · NSGA-2

✉ Kamal Garg
kamal.garg_phd.cs21@gla.ac.in

Shashi Shekhar
Shashi.shekhar@gla.ac.in

[1] Tata Consultancy Services, Mumbai, India

[2] Department of Computer Engineering and Applications, GLA University, Mathura, India

1 Introduction

Testing is an integral part and plays a pivotal role in the software development process whether it is desktop applications or mobile applications [1]. The testing process enhances software reliability by eliminating faults and ensuring fault-free performance [2]. Software development modifications are a continuous process that needs regression testing that reports the effect on the software due to changes in one or more modules or adding additional functionalities. Generally, the software testing and maintenance budget is very high, so running the whole test suite every time is not desirable as it is expensive. The best way is to choose the most critical and practical subset of test cases for re-testing. Regression testing is generally done in three ways: test case selection [3], test case reductions [4], and test case prioritization. Test case prioritization (TCP) is widely acknowledged as the most favored approach for regression testing. Following this, two additional methods, test case selection, and test case reduction, are commonly employed. The prioritization factors for test case selection include total coverage, mutant coverage, and fault detection [5]. The literature extensively compared several TCP solutions, such as firefly [6], genetic algorithm [7], Ant colony optimization, integer linear programming [8], greedy, and particle swarm optimization as mentioned in Table 1.

The manuscript focuses on regression testing and test case prioritization, exploring various algorithms, including greedy, meta-heuristic, and optimization techniques. The regression testing delves into the following steps: efficient selection of test cases, reduction in numbers to avoid complexity, and prioritization to increase the rate of fault detection [9]. Multi-objective optimization techniques, particularly in multi-objective test case prioritization (MOTCP), aim to optimize multiple objectives, such as code coverage

Published online: 02 May 2024

Table 1 List of related work

References	Techniques applied	Dataset/subjects	Result
[10]	Learn to rank	Extended finite state Machine's Protocols	APFD (mean) is 0.884
[11]	Dependency structure	Elite, GSM, CRM, MET, CZT	APFD is 56–62%
[8]	Integer linear prog.	SIR Repository	Reduced execution time and APFD value improved
[12]	Lexicographical ordering	Ant, Galileo, Jmeter, Jtopas, NanoXML (Nano), XML-Security (XML)	Fault detection rate enhanced
[13]	NSGA-II, greedy and genetic algorithm	SIR Repository	The greedy approach performed better than the hybrid approach
[9]	Greedy method	Java open-source program	Achieve high mean APFD value and increased bug detection capabilities
[14]	Linear regression	Camel 1.6.1	Weight has been calculated based on the relation between bugs and OO metrics
[6]	Fire-fly	SIR Repository	APFD = 0.9517, and average time execution = 220 s
[7]	Gravitational-search and Genetic algorithm	UMD2005b	APFD = 0.9827, and minimized suite size

and execution time. While higher code coverage enhances fault detection, the paper emphasizes identifying test cases covering modified code or segments likely to impact functionality. The main contribution is prioritizing test cases based on identified target points highlighting fault-prone code areas. In addition to code complexity, the paper considers the historical behavior of test cases to determine their prioritized order.

2 Proposed methodology

The MOTCP task utilizing NSGA-2 necessitates carefully balancing conflicting objectives, requiring thoughtful consideration. Our objective is to maximize the values of APFD and sensitivity index (SI), prioritizing test cases with a higher potential for fault detection and coverage of critical code areas. This approach enhances the effectiveness and comprehensiveness of regression testing, ultimately leading to improved software quality. In addition, we strive to minimize the execution cost to optimize resource utilization and reduce the time required for test case execution. Minimizing the execution cost ensures efficient allocation of testing resources and helps streamline the overall regression testing process.

We formulate and optimize three fitness functions: APFD, sensitivity index, and cost to accomplish these goals. Through a comprehensive discussion of these fitness functions and their formulation, we provide insights into how they contribute to the overarching objective of effective and efficient test case prioritization. To illustrate our proposed methodology and the formulation of fitness functions, we employ a small-scale project named Project P as a case study [15]. This project comprises five modules and seven classes. It also contains two tables: the first is the test case vs. fault metrics, and the second is the test cases vs. class matrix, illustrating the relationship between test cases with faults and project classes, respectively. This small project is used in further sections to calculate various parameters for the proposed methodology.

2.1 Average percentage of fault detection (APFD)

The primary objective function measures the fault detection rate (0 to 100) by organizing test cases. A higher APFD value indicates a better fault detection rate. Equation 1 calculates APFD, where TCi is the test case sequence, n is the number of test cases, and m is the total number of faults.

$$APFD = 1 - \frac{TC_1 + TC_2 + \cdots + TC_m}{nm} + \frac{1}{2n} \tag{1}$$

2.2 Sensitivity index (SI)

The objective of regression testing is to evaluate the impact of software modifications by giving priority to test cases that cover these changes. Fault sensitivity, assessed using weighted assignments, is critical in determining the priorities. Key target points in this process involve prioritizing test cases that successfully detect a significant number of faults, newly created test cases, test cases that cover modified or newly generated code, test cases with a history of high failure rates, test cases dependent on fault-prone areas of code complexity, and test cases that cover highly complex code or classes. These considerations significantly enhance the

effectiveness of regression testing and ensure comprehensive coverage of critical areas within the software.

This paper justifies the points mentioned above by introducing the sensitivity index (SI) based on two crucial parameters: the code complexity of classes and the history/status of test cases. By incorporating these parameters, we derive two weight metrics:W_c, a weight matrix for classes, and W_t, a weight matrix for test cases. Computing the second objective function, SI, involves calculating the area under the curve, which offers valuable insights into the prioritization process. The internal complexity of the code and its functional dependencies are utilized to identify critical test cases. As regression testing is conducted on Java applications, object-oriented metrics such as McCabe's Cyclomatic Complexity matrices and other custom metrics are employed to assess the code complexity [16–18].

2.2.1 Weight corresponding to classes (W_c)

For regression testing, the software tester assigns weights to code complexity properties based on Table 2, focusing on Project P [15]. W_c is calculated using this weight matrix. Class-wise code complexity matrix values are computed, outliers are detected, and if found, marked as 1; otherwise, marked as 0. The Weight for each class is then calculated accordingly.

Example: If the coupling between objects (CBO) has a value ranging from 0 to 47, and the acceptable range falls within 15% or lower and upper limits (i.e., 15–85%), such that values between 7.5 and 39 are accepted, we can examine an example scenario. Suppose the CBO value for class C2 is 32, and the CBO value for C1 is 4. In this case, C1 is considered an outlier because its CBO value is below 15% (7.5). Therefore, the CBO value for class C1 is 0, while the CBO value for C2 is 1. Table 3 shows the class complexity metrics (CCM) value of class C1 after outlier detection; the same process is also performed for other classes.

After identifying outliers, the Weight of a class is determined by multiplying the Weight specified by the tester with the calculated value after outlier identification, using Eq. 2. For class C1 in the given Project P, the calculation of W_c is presented below:

$$W_c = \sum_{m \in M} W_m \times V_m \tag{2}$$

W_c for C1 = 2×0 + 1×1 + 2×0 + 1×0 + 1×0 + 2×1 + 2×0 + 1×1 + 2×1 = 6.

The W_c values for all classes are calculated and summed up to generate the total Weight W_{tc} for the test case corresponding to the covered classes in this test case [15].

2.2.2 Weight corresponding to test cases (W_t)

Weighted test case (W_t) is determined by analyzing test case behaviour through historical and status data of the test case. The calculation involves modified class coverage, code coverage, dependency, faults, cost, new functionality, and status history. The calculation of W_t employs the same outlier method (Eq. 3), utilizing a threshold of 15% to identify outliers.

$$W_t = \sum_{m \in M} W_m \times V_m \tag{3}$$

Once the values of W_t and W_{tc} have been computed, their cumulative sums, CW_t and CW_{tc}, are calculated by the order of prioritized test cases. The area under the curve (AUC) between CW_t and CW_{tc}, is our sensitivity index [15]. The AUC is determined using the trapezoidal rule, as illustrated in Eq. 4.

$$\begin{aligned} \int_{x_1}^{x_n} f(x)dx = {} & (x_2 - x_1)\frac{f(x_1) + f(x_2)}{2} \\ & + (x_3 - x_2)\frac{f(x_2) + f(x_3)}{2} + \ldots \\ & + (x_n - x_{n-1})\frac{f(x_{n-1}) + f(x_n)}{2} \end{aligned} \tag{4}$$

2.3 Cost

The cost is the third objective function, defined as the execution time associated with the order of test cases. A straightforward cumulative sum is computed to determine the total cost of the test cases based on their execution order.

$$Cost = \sum_{i=1}^{n} CumSum(ET_i) \tag{5}$$

Table 2 Weights assigned to CCM by tester

Matrices	CBO	RFC	LCOM	DIT	NOC	MCN	WMC	LOC	CCD
Weights	2	1	2	1	1	2	2	1	2

Table 3 CCM value of class C1 after outlier detection

Matrices	CBO	RFC	LCOM	DIT	NOC	MCN	WMC	LOC	CCD
Value	0	1	0	0	0	1	0	1	1

Based on the provided order T_0 [15] of test cases and the corresponding execution times presented in Table 4, the total cost, computed using Eq. 5, amounts to 735 s.

3 Experimental assessment

The Experimental Assessment section aims to assess the effectiveness and performance of the proposed MOTCP approach. Our experimental procedure consists of a series of steps shown below to evaluate the effectiveness of our proposed methodology for test case prioritization, as depicted in Fig. 1.

Step 1: Dataset (subjects): Utilizing custom Java applications [15] and three open-source Java projects from SIR [19], we create different versions of each custom application and gather information about the size, lines of code, test cases, and faults for the open-source projects**.**

Step 2: Fault seeding: We employ fault seeding and mutation fault techniques to introduce artificial faults and simulate potential code mutations to evaluate our test case prioritization methods comprehensively.

Step 3: Test case generation: Test cases are explicitly generated and tailored for assessing module functionality, maintaining diverse test cases at two levels: test class and test method. No reduction or prioritization of test cases occurs during this phase.

Step 4: Software matrices generation: Various software metrics are generated, resulting in five data files: TestCases_Faults.csv, TestCases_Classes.csv, Class_Weights.csv, TestCases_Weights.csv, and TestCases.csv.

Step 5: Objective function optimization: Using the extracted data, we calculate three proposed objective functions (APFD, cost, and sensitivity index) and optimize them using the NSGA-2 algorithm. Parameters include an initial population size equal to the number of test cases, iterations twice the number of test cases, and crossover/mutation rates set to 0.5 and 0.25, respectively.

3.1 Evaluation of model

When assessing the performance of our proposed model, we rely on the APFD as the primary objective to determine the effectiveness of various models. APFD is a metric employed to gauge the efficacy of a software testing technique or strategy in fault detection. The evaluation determines the approach's ability in test case prioritization and its impact on regression testing. To guide our evaluation, we address the following research questions.

(a) How does the performance of the proposed MOTCP approach compare to existing methods in terms of prioritization effectiveness?
(b) What is the trade-off between maximizing APFD and SI and minimizing execution costs in MOTCP? How does this inclusion affect prioritization results?

4 Result analysis

This section provides a brief overview of the outcomes obtained from our proposed methodology. Here, the performance of NSGA-2 is compared to various state-of-the-art algorithms and recent publications related to TCP. This includes additional algorithms such as greedy [20], 2 Opt [20], genetic algorithm (GA) [21], TCP using Honey Bee optimization (HB) [22], MOTCP using African buffalo optimization (MOBF) [23], and Analytic hierarchy process

Table 4 Execution cost of test cases used in projects P

Test cases	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Execution time	15	17	22	29	16	24	19	21

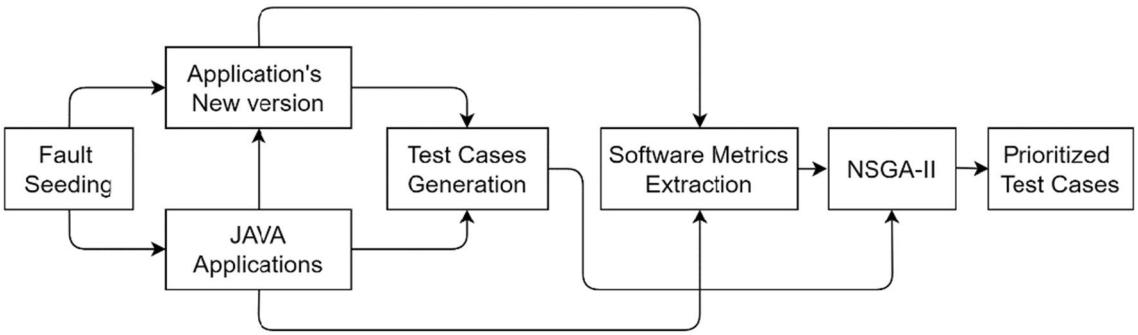

Fig. 1 Block diagram of proposed methodology

Table 5 Comparison of APFD value calculated from different algorithms

	NSGA-II	HB (APFD)	GA (APFD)	MOBF	2-Opt	AHP	Greedy
P1	97.69	97.62	97.44	97.56	96.68	96.61	96.48
P2	97.71	97.45	97.40	97.41	96.86	96.89	96.40
P3	97.91	97.67	97.62	97.6	97.14	96.80	96.74
P4	97.75	97.53	97.48	97.40	96.74	96.57	96.48
P5	98.14	97.92	97.88	97.65	97.48	97.37	97.31
Ant	93.27	89.46	90.24	87.65	86.74	87.54	84.76
Jmeter	93.84	91.37	90.96	89.26	89.17	88.19	85.26
Jtopas	94.14	92.21	91.86	91.14	90. 25	89. 34	86. 54

Table 6 Wilcoxon–Mann–Whitney statistical test result

Group1	Group2	p value (0.05)
NSGA-II	MOBF	0.0014
NSGA-II	GA (APFD)	0.0074
NSGA-II	HB (APFD)	0.013
NSGA-II	AHP	6.23E-05
NSGA-II	2-Opt	8.27E-05
NSGA-II	Greedy	1.34E-05

(AHP) based TCP [24]. These algorithms were applied to the provided dataset, as presented in the previous section.

Table 5 shows the comparison of various models corresponding to the dataset provided. We can see that our NSGA-2 perform better compared to other algorithms. If we compare NSGA-2 with the MOBF algorithm, it is found that MOBF lacks performance. The reason behind this is that NSGA-2 take care of diversity preservation and provides balanced exploration and exploitation compared to other multi-objective algorithms. We can also see that MOBF is lagging behind HB and GA but performs much better than 2-OptmAHP and Greedy. If we compare the single objective algorithm GA and HB with NSGA-2, it is found that HB performs better results for the medium type of problem when the number of tests is not significant and stuck in the local optimal solution for large test cases. Conversely, GA maintains its performance and gives good results for large problems compared to HB. If we compare 2-Opt, AHP and Greedy, it is found that Greedy performs worst while 2-Opt and AHP performance are mixed.

To conduct further statistical analysis on the performance of different algorithms, we performed the Wilcoxon–Mann–Whitney statistical test between NSGA-2 and other algorithms. The results of this analysis are presented in Table 6. In this test, a p value threshold of 0.05 was chosen. The test indicates significant differences between the performance of NSGA-2 and the other algorithms. In response to our first research question, it has been determined that NSGA-2 outperforms other algorithms. Additionally, it also discovered that the proposed approach demonstrates superior performance for more extensive data sizes.

4.1 Three-point analysis

In addressing the second research question concerning the trade-off between different objective functions in the context of MOTCP, which explores the impact of their inclusion on prioritization results, we generated two tables. Table 7 illustrates the effect on the performance of our model when each objective function is removed individually. It was observed that the proposed MOTCP exhibits the poorest performance when APFD is removed as an objective function, whereas the removal of Cost and SI yields comparatively better results.

Table 7 APFD comparison while removing one objective at a time

	NSGA-2 (APFD + COST + SI)	NSGA-2 (COST + APFD)	NSGA-2 (SI + APFD)	NSGA-2 (COST + SI)
P1	97.69	97.14	97.54	91.292
P2	97.71	96.60	97.66	90.165
P3	97.91	97.12	97.82	93.110
P4	97.75	96.92	97.53	93.477
P5	98.14	97.08	97.73	94.268
Ant	93.27	89.84	90.30	85.974
Jmeter	93.84	90.92	91.02	87.972
Jtopas	94.14	91.62	92.11	89.762

Table 8 APFD comparison considering one objective at a time

	GA (APFD)	GA (COST)	GA (SI)
P1	97.44	72.652	89.542
P2	97.40	74.392	88.145
P3	97.62	78.108	91.012
P4	97.48	76.915	91.447
P5	97.88	79.003	93.893
Ant	90.24	74.541	81.859
Jmeter	90.96	76.189	83.764
Jtopas	91.86	74.847	86.321

The response is mixed when removing SI as one of the objective functions. This is because having a high SI value does not necessarily correspond to a high APFD value. The purpose of including SI is to ensure adequate coverage of target points for effective regression testing. It is possible that test cases covering target points may not contain any faults to detect, resulting in their execution being delayed in the sequence. Consequently, for the same SI values, multiple APFD values can be obtained.

It can be observed that removing cost as an objective function has the most negligible impact on the proposed MOTCP. This is because there is very little likelihood that a test case with a lower execution cost would have a high APFD score. It is important to note that this cost refers to the cumulative sum of execution time rather than the total execution time. This cumulative sum is entirely dependent on the order of test cases. Table 8 also reveals that incorporating Cost or SI alongside APFD improves the model's performance. There are instances where single-objective approaches get trapped in local optimization problems, and the inclusion of cost or SI acts as a catalyst to overcome such issues. Furthermore, to ascertain the individual importance of each objective function, we conducted GA on each objective independently, as depicted in Table 8.

5 Conclusion and future scope

This paper employs NSGA-2 for multi-objective test case prioritization, emphasizing APFD, SI, and Cost as objective functions. SI, calculated with a focus on fault generation and identification, significantly contributes to our approach. Comparative analysis with state-of-the-art algorithms, using APFD as the criterion, demonstrates the effectiveness of our methodology. NSGA-2 outperforms other algorithms, providing balanced solutions across all objectives, and proves valuable in regression testing scenarios. However, future work should explore integrating SI and APFD into a unified objective to enhance control in test case prioritization.

Funding No funding was obtained for this study.

Data availability Data can be made available on reasonable request.

Declarations

Conflict of interest There is no conflict of interest.

References

1. Samet S, Ishraque MT, Ghadamyari M, Kakadiya K, Mistry Y, Nakkabi Y (2019) TouchMetric: a machine learning based continuous authentication feature testing mobile application. Int J Inf Technol 11(4):625–631. https://doi.org/10.1007/s41870-019-00306-w
2. Saraf I, Iqbal J (2019) Generalized software fault detection and correction modeling framework through imperfect debugging, error generation and change point. Int J Inf Technol 11(4):751–757. https://doi.org/10.1007/s41870-019-00321-x
3. Verma AS, Choudhary A, Tiwari S (2023) A novel chaotic Archimedes optimization algorithm and its application for efficient selection of regression test cases. Int J Inf Technol 15(2):1055–1068. https://doi.org/10.1007/s41870-022-01031-7
4. Rehman Khan SU, Lee SP, Javaid N, Abdul W (2018) A systematic review on test suite reduction: approaches, experiment's quality evaluation, and guidelines. IEEE Access 6:11816–11841. https://doi.org/10.1109/ACCESS.2018.2809600
5. Mishra DB, Panda N, Mishra R, Acharya AA (2019) Total fault exposing potential based test case prioritization using genetic algorithm. Int J Inf Technol 11(4):633–637. https://doi.org/10.1007/s41870-018-0117-0
6. Khatibsyarbini M, Isa MA, Jawawi DNA, Hamed HNA, Mohamed Suffian MD (2019) Test case prioritization using firefly algorithm for software testing. IEEE Access 7:132360–132373. https://doi.org/10.1109/ACCESS.2019.2940620
7. Bajaj A, Sangwan OP (2021) Discrete and combinatorial gravitational search algorithms for test case prioritization and minimization. Int J Inf Technol 13(2):817–823. https://doi.org/10.1007/s41870-021-00628-8
8. Hao D, Zhang L, Zang L, Wang Y, Wu X, Xie T (2016) To be optimal or not in test-case prioritization. IEEE Trans Softw Eng 42(5):490–505. https://doi.org/10.1109/TSE.2015.2496939
9. Chi J et al (2020) Relation-based test case prioritization for regression testing. J Syst Softw 163:110539. https://doi.org/10.1016/j.jss.2020.110539
10. Huang Y, Shu T, Ding Z (2021) A learn-to-rank method for model-based regression test case prioritization. IEEE Access 9:16365–16382. https://doi.org/10.1109/ACCESS.2021.3053163

11. Haidry S-Z, Miller T (2013) Using dependency structures for prioritization of functional test suites. IEEE Trans Softw Eng 39(2):258–275. https://doi.org/10.1109/TSE.2012.26
12. Eghbali S, Tahvildari L (2016) Test case prioritization using lexicographical ordering. IEEE Trans Softw Eng 42(12):1178–1195. https://doi.org/10.1109/TSE.2016.2550441
13. Yoo S, Harman M (2007) Pareto efficient multi-objective test case selection. In: Proceedings of the 2007 international symposium on software testing and analysis. ACM, London, pp 140–150. https://doi.org/10.1145/1273463.1273483
14. Taneja D, Singh R, Singh A, Malik H (2020) A Novel technique for test case minimization in object oriented testing. Proc Comput Sci 167:2221–2228. https://doi.org/10.1016/j.procs.2020.03.274
15. GitHub. Build software better, together. https://github.com/CodeReformer/MOTCP. Accessed 20 Jan 2024
16. Chhillar RS, Gahlot S (2017) An evolution of software metrics: a review. In: Proceedings of the international conference on advances in image processing. ACM, Bangkok, pp 139–143. https://doi.org/10.1145/3133264.3133297
17. Debbarma MK, Debbarma S, Debbarma N, Chakma K, Jamatia A (2013) A review and analysis of software complexity metrics in structural testing. Int J Comput Commun Eng. https://doi.org/10.7763/IJCCE.2013.V2.154
18. Lincke R, Lundberg J, Löwe W (2008) Comparing software metrics tools. In: Proceedings of the 2008 international symposium on Software testing and analysis, Seattle. ACM, pp 131–142. https://doi.org/10.1145/1390630.1390648
19. Software-artifact Infrastructure Repository: Home. [Online]. https://sir.csc.ncsu.edu/portal/index.php. Accessed 20 Jan 2024
20. Khanna M, Chaudhary A, Toofani A, Pawar A (2019) Performance comparison of multi-objective algorithms for test case prioritization during web application testing. Arab J Sci Eng 44(11):9599–9625. https://doi.org/10.1007/s13369-019-03817-7
21. Huang Y-C, Peng K-L, Huang C-Y (2012) A history-based cost-cognizant test case prioritization technique in regression testing. J Syst Softw 85(3):626–637. https://doi.org/10.1016/j.jss.2011.09.063
22. Nayak S, Kumar C, Tripathi S, Mohanty N, Baral V (2021) Regression test optimization and prioritization using Honey Bee optimization algorithm with fuzzy rule base. Soft Comput 25(15):9925–9942. https://doi.org/10.1007/s00500-020-05428-z
23. Singhal S, Suri B (2020) Multi objective test case selection and prioritization using African buffalo optimization. J Inf Optim Sci 41(7):1705–1713. https://doi.org/10.1080/02522667.2020.1799514
24. Nayak S, Kumar C, Tripathi S (2022) Analytic hierarchy process-based regression test case prioritization technique enhancing the fault detection rate. Soft Comput 26(15):6953–6968. https://doi.org/10.1007/s00500-022-07174-w