Soft Computing (2021) 25:9925–9942
https://doi.org/10.1007/s00500-020-05428-z

METHODOLOGIES AND APPLICATION

Regression test optimization and prioritization using Honey Bee optimization algorithm with fuzzy rule base

Soumen Nayak[1,2] · Chiranjeev Kumar[1] · Sachin Tripathi[1] · Nirjharini Mohanty[3] · Vishal Baral[3]

Published online: 27 November 2020

Abstract

Regression testing is a maintenance level activity performed on a modified program to instill confidence in the software's reliability. Prioritization of test case arranges the regression test suite to detect the faults earlier in the testing process. The test cases necessary for validating the recent changes and finding the maximum faults in minimum time are selected. In this manuscript, an optimization algorithm (Bee Algorithm) based on the intelligent foraging behavior of honey bee swarm has been proposed that can enhance the rate of fault detection in test case prioritization. The bee algorithm, along with the fuzzy rule base, reduces the test cases' volume by selecting the test cases from the pre-existing test suite. The proposed algorithm developed for enhancing the fault detection rate in minimum time is inspired by the behavior of two types of worker bees, namely scout bees and forager bees. These worker bees are responsible for the maintenance, progress, and growth of the colony. The proposed approach is implemented on two projects. The prioritization result is quantified by using the average percentage of fault detection (APFD) metric. Compared with other existing prioritization techniques like no prioritization, reverse prioritization, random prioritization, and previous work, the proposed algorithm outperforms all in fault detection rate. The effectiveness of the proposed algorithm is represented by using the APFD graphs and charts.

Keywords Regression testing · Bee algorithms · Fuzzy logic · Test case prioritization

1 Introduction

Testing a program or application requires the most significant effort and is the most critical step in the software's life cycle (SDLC). According to Craig and Jaskiel (2002), "*Testing is a concurrent lifecycle process of engineering, using and maintaining testware (*i.e.*, testing artifacts) to measure and improve the quality of the software being tested.*" Regression testing is a maintenance level activity performed to check and validate the correctness of the modified software. It is defined in the IEEE software glossary (1990) as "*Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.*" Exhaustive regression testing consumes lots of time and cost for which it is not feasible to test every test case. Thus, effective testing, along with the prioritization techniques, is required to make the regression testing viable. Test prioritization or test case prioritization (Rothermel et al. 2001) is an extension of software testing done to decide the priority of the test cases following some

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00500-020-05428-z) contains supplementary material, which is available to authorized users.

✉ Soumen Nayak
soumen.nayak@gmail.com

Chiranjeev Kumar
kumar.c.cse@ismdhanbad.ac.in

Sachin Tripathi
var_1285@yahoo.com

Nirjharini Mohanty
nirjharini99@gmail.com

Vishal Baral
vishal.baral19@gmail.com

[1] Department of Computer Science and Engineering, IIT (ISM), Dhanbad 826004, India

[2] Department of Computer Science and Engineering, ITER, S'O'A Deemed to be University, Bhubaneswar, India

[3] Department of Computer Science and Information Technology, ITER, S'O'A Deemed to be University, Bhubaneswar, India

testing criteria. By performing test case prioritization, the testers and developers can reduce the computational resources like effort, cost, time, etc. and make sure that the launched product is of exceptional quality. Unlike other techniques like test case selection and test case minimization, the test case prioritization never discards any test cases (Rotherrmel et al. 1998).

A fuzzy rule-based framework is a rule-based structure in which fuzzy logic is used to represent various formats of knowledge that address the solution to the issues and the relations and interactions present among the parameters in the form of a model. This has been used extensively in modeling both human problem-solving practices and adaptive behavior. Here, a traditional method has found the usage of the "if–then" rule for representing the social concepts. If there are rule antecedents, then that leads to subsequent, i.e., action is properly executed. Fuzzy systems are an appreciable tool that can be used to model complicated systems in which the conventional models are not so effective due to the complexity and the precision (Malz et al. 2012). For the rule-based framework, the fuzzy logic provides a single inference computation model to deal with the various reasoning modes that are similar and approximate but not precise.

The bee algorithm is an optimization algorithm that is suggested in 2005 by Karaboga (2005). In the honey bees (*Apis mellifera*), Swarm Intelligence (SI) criteria such as self-organization and segregation of labor characteristics are distinctly and clearly shown. The bee algorithm is a classic instance of teamwork, practice, communication, synchronization, and alignment. The method in which they forage and supply their comb with the requisite amount of nectar to sustain the comb is astounding. This phenomenon gives rise to curiosity in regression test optimization using honey bees. In this manuscript, we seek to construct an approach for the optimization and prioritization of regression test cases using honey bee foraging algorithm (bee algorithm) and fuzzy logic. The bee algorithm's disadvantage is its slow converging rate, and there is every possibility that it may give local optima rather than a global one. The fuzzy rule-based system is used to alleviate those drawbacks in the algorithm. For our study, two projects have been considered. The proposed technique is contrasted with other prioritization strategies such as no prioritization (NP), reverse prioritization (REP), random prioritization (RP), and existing prioritization strategies available in the literature (Kavitha and Sureshkumar 2010; Tyagi and Malhotra 2015; Nayak et al. 2016). The efficiency of all the algorithms is measured by a metric known as the average percentage of faults detected (APFD) (Elbaum et al. 2000a, b). The result shows the suggested technique outperforms all other techniques.

The objective of this paper is to overcome the gap found in the current study. There are several research questions on which this study is focused as follows:

RQ1 Can the regression test prioritization techniques enhance the rate of fault detection of test cases in test suites?
RQ2 How the proposed algorithm enhances the fault detection rate and gives a solution for the test case prioritization problem?
RQ3 In what way the proposed algorithm is better than the other existing prioritization techniques?

The three questions that this paper answers are related to the computational cost and regression testing consistency. We also determine which algorithm to prioritize for optimum fault coverage in minimum time during the regression test.

This research proposes a new regression test prioritization algorithm, which simulates the bee colonies foraging behavior. The proposed method, however, is a bionic optimization algorithm that delivers a higher APFD value. The manuscript should add to the prevalent literature by conveying the following:

(1) A new algorithm that has never been previously implemented in any literature to solve the test case prioritization problem,
(2) A framework that considers several inputs to have a higher APFD value still,
(3) A better prioritization algorithm is built as opposed to conventional APFD value algorithms by taking into account the current problems that have been observed up to now, and
(4) The fault-revealing method is examined by test cases heavily dependent on the test suites.

The remnant of the paper is organized as follows. Section 2 presents the literature review that discusses various approaches in test case prioritization on test case prioritization. Section 3 discusses the background consisting of concepts and fundamentals required to understand the work in brief. In Sect. 4, the proposed methodology is presented in detail. Section 5 deals with the experimental evaluation and discussion. Section 6 deals with the analysis of the result and discussion. At last, Sect. 6 presents the conclusion and the future enhancements.

2 Literature review

During the past few years, several researchers have been reviewing many characteristics and methods for prioritizing test cases to improve the detection of faults and the efficiency of tests. One of the essential aspects of test case

prioritization is choosing the correct characteristics. TCP is a method of improvement testing implemented during the initial testing process mentioned by Erdogmus et al. (2005). Prioritization strategies schedule test cases to be performed in an order that aims to maximize their efficacy in achieving specific success objectives. Yoo and Harman (2007) surveyed different prioritization approaches, addressing open-end questions, and offering more study recommendations. Rothermel et al. (2001) develop various strategies using knowledge on test execution to order test cases for regression testing using the Siemens suite. These techniques are divided into three parts: test case order based on code coverage, previously uncovered test case order based on code coverage, and test case order based on fault detection ability. The result shows that each of the techniques increases the fault detection rate. Elbaum et al. (2000a, b) proposed a metric called the APFD metric to measure faults' detection rate. Malishevsky et al. (2006) have developed a new metric APFDc that also includes differences in test cases' cost and the frequency of TCP faults. The latter metric is an improvement over the former one's shortcomings.

Mukherjee and Patnaik (2019) have introduced a fuzzy model for TCP by analyzing 6 Siemens programs and 10 C programs. The result has shown that the prioritized order produces 2.01 times more savings than non-prioritized test suites if some techniques are combined. Korel et al. (2007) proposed a model-based prioritization approach that would take the system model's information and its TCP behavior into account. Experiments were ordered into a high priority and a low priority set. They characterized and analyzed various meanings of higher and lower priority test cases. Yet, the test case is assigned a higher priority if it applies to the model's change. Shrivathsan et al. (2019) have proposed two fuzzy-based clustering and prioritization techniques that use similarity coefficient and dominancy measure. On evaluating the data from Software-artifact Infrastructure Repository (SIR), there is an increase in the plausibility of determining more relevant test cases. Srikanth et al. (2016) implemented a program usage prioritization approach and merged them with the previous history to prioritize test cases.

Test cases are organized and prioritized in these methods based on system specifications, which could play a vital role in identifying critical faults (Srikanth et al. 2005; Krishnamoorthi and Mary 2009). Kavitha and Sureshkumar (2010) have proposed a prioritization technique considering two factors like the rate of fault detection and fault impact. They have taken two industrial projects and proved that the prioritized orders are better than not prioritized test suite. Tyagi and Malhotra (2015) have considered three factors and shown better results than the work of Kavitha and Sureshkumar (2010). Nayak et al. (2016) have proposed a TCP technique that takes the weighted value of two parameters, namely fault rate and fault severity. The outcome of the paper is promising than the above two mentioned techniques. Nayak et al. (2017) proposed a prioritization approach based on multiple factors to enhance the fault exposure rate. Silva et al. (2019) have suggested a software-component approach for prioritization. Considering criticality, execution time, and history of faults, the test cases are prioritized, and the resulting test suites are analyzed to scrutinize their performance in detecting defects. The evaluation was conducted in eight applications. The findings suggest that the solution's fault detection efficiency was substantially higher than that of the unordered test suites and those obtained using a greedy process, achieving the optimum value to be tested where possible. At the design stage, Ray and Mohapatra (2014) defined a component's criticality level. They made a stronger test strategy such that the high critical components would be tested more carefully and rigorously than other less crucial components.

Li et al. (2007) proposed several TCP meta-heuristics, including a hill-climbing method, a greedy approach, an additional greedy algorithm, a genetic algorithm, and a two-optimal greedy algorithm. The greedy algorithms can produce sub-optimal results which are overcome by algorithms of meta-heuristic or evolutionary search. Their empirical study has taken six programs, and the work sheds light on the regression testing search space and points to its multi-modal existence. While the genetic algorithm performs well, greedy approaches win the race because of the landscape's multi-modal nature. Suri and Singhal (2011) have introduced an ACO dependent test case selection and prioritization strategy to overcome time constraints in the regression test. But they have not compared their result to any other method in this scenario. Öztürk (2018) has suggested a new technique of TCP inspired by bats behavior. The result reveals that it is better than ACO, PSO, and Greedy. Gao et al. (2015) prioritize test cases through an ACO algorithm that evaluates them along with the history and extent of the fault. Khatibsyarbini et al. (2019) proposed a prioritization technique based on the firefly algorithm. This technique is applied over three benchmark programs to conclude that this algorithm gives better APFD value. In software testing, bee colony optimization (BCO) or bee algorithm (Karaboga et al. 2013) has been used in many SDLC phases (Kaur and Goyal 2011). Mounika and Reddy (2015) have done path testing for test case selection using BCO. AdiSrikanth et al. (2011) proposed a test case optimization algorithm using BCO. The approach is

accredited by self-organization and reliability and focuses on constructing pathways derived from cyclomatic complexity. The result assures complete coverage of the path. The BCO algorithm is slow as compared to ACO (Sushant and Ranjan 2017) and also, it converges in the local optima. To avoid these two drawbacks, the fuzzy rule base is also been used along with the bee algorithm.

Relation-based TCP for regression testing is proposed by Chi et al. (2020). They are suggesting that the dynamic function call can guide TCP more effectively. For this, they have proposed a new approach called additional greedy method call sequence (AGC) that leverages dynamic relation-based coverage. They have applied it on eight real-world java open source projects. The result is compared with 22 TCP techniques in terms of bug detection ability. The proposed approach gives a better result than the rest. Alakeel (2014) offers a novel approach to prioritizing test cases during regression testing of systems that use fuzzy logic to make assumptions. Compared with untreated and randomly ordered test cases, the findings are positive. Malz et al. (2012) have designed a prioritization system that increases test effectiveness and fault detection rate in a prioritized order. Jeffrey and Gupta (2006) suggested a prioritization method using relevant slices.

An overview of different prioritization techniques is cited along with the purposes, strategies, and datasets used. From the survey, many issues related to BCO and prioritizations are searched. In the case of BCO, the algorithm is slow, and there is every possibility that the algorithm halts once local minima is obtained. The fuzzy rule base has been used to improve the algorithm's rate. The BCO algorithm is characterized by autonomy, self-organizing, and decentralized distributed functioning systems. From the literature, it is observed that still research will obtain a better performing TCP algorithm. This motivates the design of a new, improved nature-inspired algorithm that can converge appropriately giving a solution to the TCP problem.

3 Proposed methodology

In this section, the test case prioritization problem has been discussed, and then the proposed algorithm is mentioned that gives a solution to the prioritization problem.

3.1 Test case prioritization problem

Test case prioritization deals with the ordering of the test cases by following a test adequacy criterion. This maximizes the probability of the fault detection rate, faster code coverage, etc. Moreover, Rothermel et al. (2001) describe the problem of prioritization as such:

Definition 1 TCP_ProblemStatement

Given: A set of test cases, T, the permutation set of T, PT, and f as a function from PT to the set of real numbers.
Problem: Find $T' \in PT$ such that $((T'' \in PT)\ (T'' \neq T')\ [f(T') \geq f(T'')]$

As per the definition, all the possible arrangements (orderings) together in a T set are denoted by PT. Here, a function "f" returns a value known as "award value" for each order. (It is presumed in the definition that the superior value is desired more than that of the lower ones without the loss of generality).

From the above definition, the goals of prioritization like increasing the rate of fault detection, code coverage in the system under test (SUT) at a faster rate, enhancing SUT dependability at a quicker pace, detecting high-risk faults in advance in the testing process, and maximize the probability of disclosing flaws created due to the changes in the code earlier etc. can be stated quantitatively using the representation of "f." Based on the selection of "f" the prioritization problem may become a problem that is unmanageable or un-decidable. The TCP problem results will give a result that is effective for the traveling salesman problem (Jiang 2015). TCP is an NP-complete problem (Rothermel et al. 2001) and all NP-complete problems are NP-hard. Hence the bee algorithm and the fuzzy logic are implemented to solve the hard COPs (combinatorial optimization problems) (Pan et al. 2011).

3.2 Bee algorithm

The bee algorithm produces a random result or an initial population of size N, and N denotes the population's size or total food sources (the total number of Bees is equal to the total number of food sources). Food source X_i ($i = 1,\ldots, N$) are D-dimensional vectors, which correspond to one employed bee. $F(X_i)$ is either the fitness value of solution X_i or the quantity of nectar from the source of food. For each source, the iteration times are restricted to "limit," and for the entire population, max iteration is restricted to MAXpp. The steps are as follows:

1. Randomly produce afeasible solution to N.
   For $i = 1$ to N
   do

$$X_i^j = X_{\min}^j + \text{rand}(0,1)\left(X_{\max}^j - X_{\min}^j\right) \quad (1)$$

/*j = D-dimensional vector component*/

2. Employed bee is searching for a good solution in the neighboring food supply and measure the fitness value if the new source fitness is better than the old one.
   for i = 1 to N
   do

$$V_i^j = X_i^j + \varphi_i^j\left(X_i^j - X_k^j\right) \quad (2)$$

   /*$F(V_i)$ = new source fitness, if $F(X_i) < F(V_i)$, V_i replace X_i, if not then it is unchanged*/
   The φ_i^j is used to generate a random value between [− 1 to 1], according to the proposed method our $\varphi \geq 0$, as our solution lies between [0,1].
3. For follow-up bees, selecting one employed bee for neighborhood search according to the likelihood of a positive correlation with the employed bee fitness rating. The likelihood of food source i is chosen and n is the maximum number of test cases:

$$P_i = \frac{F(X_i)}{\sum_i^n X_i} \quad (3)$$

   /*The way follow-up bees check and choose the food source selected is identical to step 2*/
4. Of all food sources, if the search times exceed a certain limit (restrict) but still do not find a better solution for any food source, give up the source and create a new source:
   i = 1 to N
   do
   {
   limit = 0;
   m = 0;
   if $F(X_m) = F(X_{m-1})$ then $m = m + 1$;
   if m = restrict then
   {
   give up the food source and create a new source according to formula 1;
   restrict = 0;
   }
   }
5. Evaluate if the algorithm meets the final conditions or reaches the maximum MAXpp iteration limit. If necessary, output the optimal solution, otherwise, the algorithm will be in the next iteration.

3.3 Proposed work

The regression test case optimization and prioritization will be attained by combining the fuzzy rule base and bee algorithm. The optimization of the regression test cases is done by selecting the subset of test cases covering all the faults in minimum execution time from the available test suite. This proposed approach's information processing objective is to explore and exploit good food sources within the problem search space. The Scout bees are sent to search the problem space and randomly sample it to locate and identify better food sites. With the local search application, the better sites are exploited, where a few right areas are explored more often than the others. Better sites are continually being exploited, although many scout bees are sent in each iteration to search for other available sites.

To prioritize a regression test suite to maximize fault coverage, this algorithm has been mapped and inspired by the honey bees' food foraging behavior. The algorithm has taken few assumptions as follows:

1. Food sources are the test cases
2. Scout bees are in the same quantity as the test cases
3. Forager bees as the initial population
4. The number of faults identified within the test case execution times determines the test case's food site's consistency.

The basic flow of the proposed algorithm in the form of pseudocode is as follows:

i. The variables are first initialized with the values necessary for the desired execution of the pseudocode:
   Set TC = MAX, Covered_test = 0, Exec_Time = 0, k = 0
   Generate S1 and S2 = Test suite = {T_i, FC_i, NF_i, ET_i, RET_i, PT_i}
   Store all the dataset in S1 and S2.

TCP_Algo is the algorithm's start, where all the other parts of the algorithm have been called according to the requirement. The TCP_Algo continues executed till we do not get any new solution/path from the dataset. It returns the order of the solution based on the selection process.

Algorithm 1: TCP_Algo
Input: Fault matrix consisting of the test cases , the faults and the Execution time
Output : Prioritized Test Suite

```
1.  Initialize the data set list and other variables to zero;
2.  while(Solu != TNT){                    // TNT: Total Number of Test case
3.      if( Coverd_test != TNF){           // TNF: Total Number of faults
4.          Fault_rate();
5.          Remaining_ET();
6.          priority();
7.          P_max();
8.          C_Path();
9.      }
10.     else{
11.         Covered_test = 1 ;
12.         Keep the first selected constant and again search new path
13.     }
14. }
```

The Algorithm is divided into five modules, which is given as the following:

Module 1 This Fault_rate() is used to get the rate of fault cases covered by each test case.The Fault rate is one of the major factors of path decision as it shows total fault cases complete in a path.

Module 1: Calculate the fault rate

```
1.  Fault_rate()  //(μFC_Ti = FRi = Fault cases covered by the test case/total fault cases) – (4)
2.  {
3.    for( i=0;i<TNT;i++)
4.    {                          // NFi: Number of faults covered by the ith Test case
5.      FRi = NFi / TNF;         // FRi: Fault cover rate of the ith Test case
6.    }
7.  }
```

Module 2 The Remaining_ET() task is to calculate the remaining execution time for each test case provided in the dataset.

Module 2: Calculate the remaining execution time

```
1.  Remaining_ET()  //( μRET_Ti = ERi = Remaining execution time / total time of the problem) – (5)
2.  {
3.          for( i=0;i<TNT;i++)
4.          {// ERi: Remaining execution time rate of the ith Test case
5.              ERi = (ETi –TC)/ TC;   // ETi: Execution time of the ith Test case
6.          }
7.  }
```

Applying Fuzzy with Bee Algorithm (Sect. 3.2) makes the algorithm more efficient. The FIS used in the algorithm is the Mamdani System of inference, which is widely accepted and intuitive. In this paper, as for the application, FIS is used to find the priority of each test case, which is essential when it comes to the prioritization of a given test suite. The test cases all together make the test suite, and then one by one, the priority is found out using the inference system with a rule or a set of rules. The test cases' priority will help us choose the best tests among them while analyzing the test suite, sorting them so that the test case with the highest priority will be noted before every other test

case. The highest prioritized test case is combined with the rest of the test cases. The bee algorithm works along with the FIS to find the best test case as mentioned before. Here, the fuzzy rule (particularly Fuzzy Inference System) gives us assurance for enhancing our algorithm's efficiency.

Module 3 The priority() is assigned to give each path's priority value using Fault rates, Remaining Execution Time, and using the fuzzy rule.

This project's primary goal is to enhance the fault detection rate, increasing the fault detection rate in minimum time from the system under test. Working principle of the proposed approach is a population-based approach where every element of the test suite contains the probable result of the optimization problem. Each test case's fitness value is equated with the consistency or fitness of the colligated solution. Figure 1 shows an outline of the solution.

Module 3: Priority of each test case using fuzzy rule

```
1. priority()        //(FIS (μFC_Ti , μRET_Ti) =Pi = (μFC_Ti X μRET_Ti) / (μFC_Ti + μRET_Ti) (Fuzzy Formula))  - (6)
2.                                   // Priority_Ti = FIS (μFC_Ti , μRET_Ti)  -                (7)
3. {
4.    for(i=0;i<TNT;i++)
5.    {
6.       Pi = (ERi * FRi ) / (ERi + FRi );    // Pi: Priority of the ith Test case using fuzzy rule
7.    }
8. }
```

Module 4 The P_max() returns the max Priority path, which will help us cover all the fault cases with minimum time and minimum test cases.

4 Experimental evaluations

This section of the paper deals with experimental tests required to show the advantages and the efficiency of the present TCP algorithm. This section also dealt briefly with the empirical assessment of the tests.

Module 4: Find Max Priority test case

```
1. P_max()
2. {
3.        for(i=0;i<TNT;i++)
4.        {
5.              Find test case with max priority from Pi
6.        }
7.        Select(Test case with max Priority);
8. }
```

Module 5 The C_Path() work is to combine the paths until we cover all the fault cases. The combination is done with the highest priority selected path and remaining test cases to cover the fault cases.

Module 5: Combine test case

```
1. C_Path()
2. {
3.        let as assume Tj  have max priority
4. //Tj is add to Solu vector only when the Solu vector don't have Tj
5.        Solu.add(Tj );                          //K increases when the new test case is selected
6.        Exec_Time = Exec_Time +ETi ;      // Total time of execution
7.        Covered_test = Covered_test+NF;   // Increment for every selected Test case
8.        for(i=0;i<TNT;i++)
9.        {
10.             if(Tj != Ti)
11.             {
12.                   ETi  = ETj + ETi ;
13.                   FCi = FCi + FCj ;
14.             }
15.       }
16. }
```

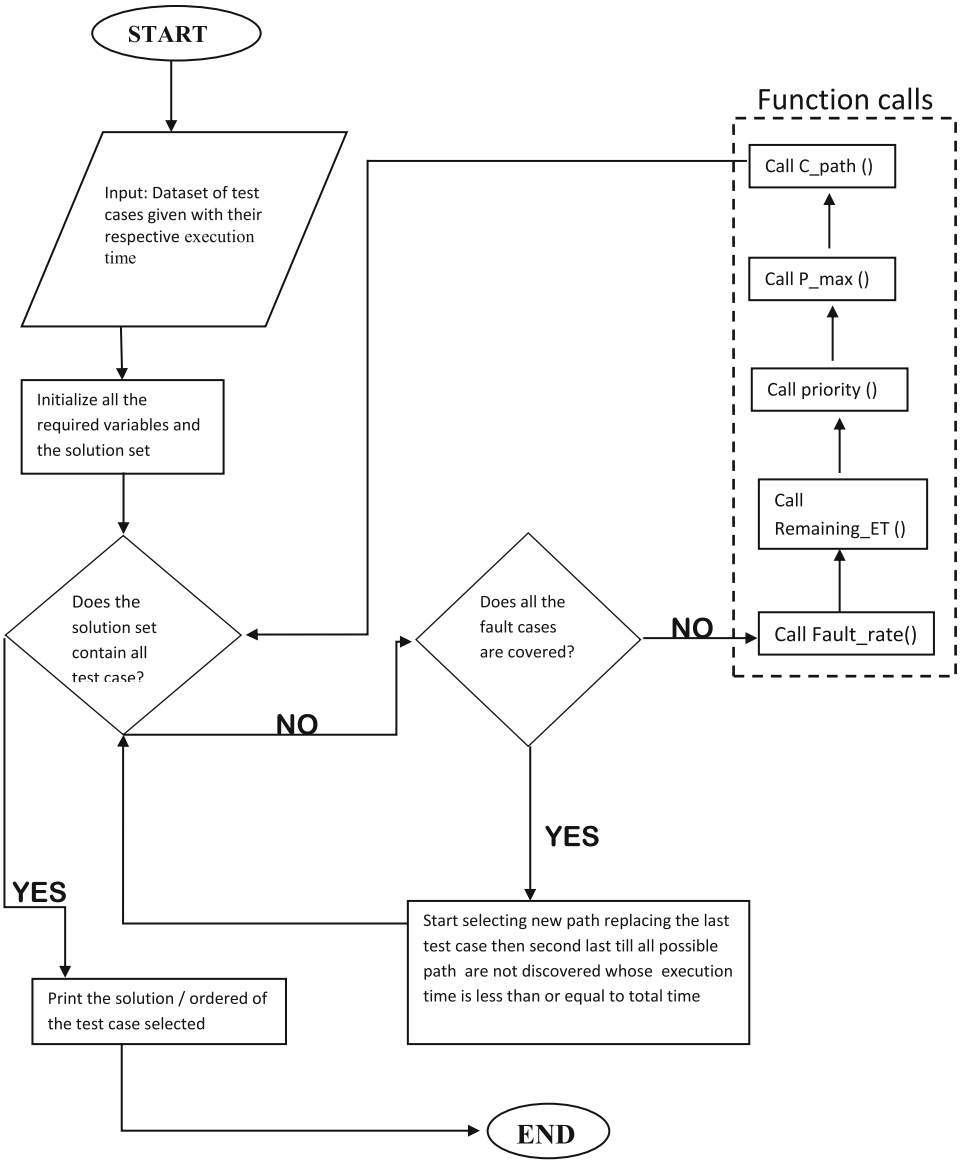

Fig. 1 Architecture or high-level flow of the algorithm

Table 1 Test cases as $T1 \ldots T10$ and $F1 \ldots F10$as faults, "χ" represent the faults addressed by the T

Test case/ Faults	*T*1	*T*2	*T*3	*T*4	*T*5	*T*6	*T*7	*T*8	*T*9	*T*10
*F*1								χ	χ	
*F*2		χ	χ		χ					
*F*3				χ		χ				χ
*F*4		χ	χ							
*F*5								χ		
*F*6								χ	χ	
*F*7				χ	χ		χ			
*F*8	χ					χ				
*F*9				χ	χ	χ				χ
*F*10	χ							χ		

4.1 Demonstration for project 1

The exploratory data needed for our study were drawn from an established work by Kavitha and Sureshkumar (2010). A 3 GHz Intel Pentium 4 PC with RAM of 8 GB is the experimental setup's hardware specification. This research was carried out at CCSQ, Chennai. The project is a VB project and was extensively analyzed manually and with an automated tool, QTP 9.5, by incorporating ten faults. The ten separate test cases are created using random numbers in the language "C." Finally, they have noticed the time taken by every test case to identify faults. The test suite T to be considered for this example exists of test cases = {*T*1, *T*2,…, *TC*9, *TC*10} and test cases = {*F*1, *F*2, …, *F*9, *F*10}. Table 1 displays the fault matrix consisting of the selected data, the number of faults noted in every test

Table 2 The number of faults detected and the time consumed by each test case

Test cases	No. of faults detected	Execution time (ms)	Remaining execution time (ms)
*T*1	2	9	34
*T*2	2	8	35
*T*3	2	14	29
*T*4	3	9	34
*T*5	2	12	31
*T*6	3	14	29
*T*7	1	11	32
*T*8	4	10	33
*T*9	2	10	33
*T*10	2	13	30

Table 3 Observations after the first iteration

Test case	Fault	ET	RET	μ_{FC_Ti}	μ_{RET_Ti}	Priority_T_i
T_1	2	9	34	0.2	0.7907	0.1596
T_2	2	8	35	0.2	0.8140	0.1606
T_3	2	14	29	0.2	0.6744	0.1543
T_4	3	9	34	0.3	0.7907	0.2175
T_5	2	12	31	0.2	0.7209	0.1566
T_6	3	14	29	0.3	0.6744	0.2075
T_7	1	11	32	0.1	0.7442	0.0882
T_8	4	10	33	0.4	0.7674	*0.2629*
T_9	2	10	33	0.2	0.7674	0.1587
T_{10}	2	13	30	0.2	0.6977	0.1554

Table 4 Observations after the second iteration

Test cases	Fault	ET	RET	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_8, T_1	5	19	24	0.5	0.5581	0.2637
T_8, T_2	6	18	25	0.6	0.5814	0.2953
T_8, T_3	6	24	19	0.6	0.4419	0.2545
T_8, T_4	7	19	24	0.7	0.5581	*0.3105*
T_8, T_5	7	22	21	0.7	0.4884	0.28757
T_8, T_6	7	24	19	0.7	0.4419	0.2709
T_8, T_7	5	21	22	0.5	0.5116	0.2529
T_8, T_9	4	20	23	0.4	0.5349	0.2289
T_8, T_{10}	6	23	20	0.6	0.4651	0.2620

Table 5 Observations after the third iteration

Test cases	FAULT	ET	RET	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_8, T_4, T_1	8	28	15	0.8	0.3488	0.2429
T_8, T_4, T_2	9	27	16	0.9	0.3721	*0.2633*
T_8, T_4, T_3	9	33	10	0.9	0.2326	0.1848
T_8, T_4, T_5	8	31	12	0.8	0.2791	0.2069
T_8, T_4, T_6	8	33	10	0.8	0.2326	0.1802
T_8, T_4, T_7	7	30	13	0.7	0.3023	0.2111
T_8, T_4, T_9	7	29	14	0.7	0.3256	0.2222
T_8, T_4, T_{10}	7	32	11	0.7	0.2558	0.1873

case, and the time needed to detect the faults. And this section will express the application of the proposed approach that uses the bee algorithm with fuzzy logic for prioritizing the test suite.

Table 2 shows the faults detected by the test case, the corresponding time taken in milliseconds to find the faults and the remaining execution time (in milliseconds) for the individual test cases.

The algorithm will be executed as long as it does not meet the stopping criterion: the total fault coverage and the time constraint to be maximum time 43 (in milliseconds). The sequence generated at the end of this process will be an optimal solution for this particular test suite.

The first iteration (Table 3) is carried out, and the values are computed using Eq. (5), (6), (7), and (8). The Priority_*Ti* column will give us the test case required to proceed by taking the test case with the highest priority.

As observed above, the test case with the highest priority is:

T8 with its priority as: Priority_T8 = 0.2629

Now further in the second iteration, the test case *T*8 will be combined with the rest of the tests [*T*1, *T*2, *T*3, *T*4, *T*5, *T*6, *T*7, *T*9, *T*10], resulting in a new test suite which is [(*T*8, *T*1), (*T*8, *T*2), (*T*8, *T*3), (*T*8, *T*4), (*T*8, *T*5), (*T*8, *T*6), (*T*8, *T*7), (*T*8, *T*9), (*T*8, *T*10)]. Table 4 shows the second iteration.

Just like before, the highest priority among all the test cases is highlighted:

T8, T4 with the priority as Priority_T8, T4 = 0.3105

So, taking the tuple (*T*8, *T*4), we proceed, i.e., couple it up with the remaining test case singularly excluding the test cases *T*8 and *T*4 as in Table 5. The algorithm proceeds

Table 6 Observations after the final iteration

Test cases	FAULT	ET	RET	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_8, T_4, T_2, T_1	10	36	7	1.0	0.1628	*0.1400*
T_8, T_4, T_2, T_3	9	41	2	0.9	0.0465	0.0442
T_8, T_4, T_2, T_5	9	39	4	0.9	0.0930	0.0843
T_8, T_4, T_2, T_6	10	41	2	1.0	0.0465	*0.0444*
T_8, T_4, T_2, T_7	9	38	5	0.9	0.1163	0.1030
T_8, T_4, T_2, T_9	9	37	6	0.9	0.1395	0.1208
T_8, T_4, T_2, T_{10}	9	40	3	0.9	0.0698	0.0648

Table 7 All the possible path generated

Test cases	FR	ET	RET	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_8, T_4, T_2, T_1	1.0	36	7	0.1	0.1628	0.1400
T_8, T_4, T_3, T_6	1.0	41	2	0.1	0.0465	0.0444
T_8, T_4, T_2, T_6	1.0	41	2	0.1	0.0465	0.0444
T_8, T_4, T_3, T_1	1.0	42	1	0.1	0.0253	0.0228
T_8, T_4, T_3, T_7	1.0	43	0	0.1	0	0

to the third iteration as the stopping criteria have not been met yet, as all the fault cases have not been covered.

In Table 6, this iteration proceeds as the stopping criterion has not yet been reached. So the test case with the highest priority will be selected, and the rest will be discarded. The selected tuple of the initial test cases will again be coupled with the others like the first column of Table 6. The test case with the highest priority gives:

T8, T4, T2with priority as Priority_T8, T4, T2 = 0.2633

Finally, the stopping criterion is met, and the iteration stops. Table 7 contains all the possible solutions to the problem. Those test cases are merged to give the optimal path for the forager bees to find the food source. These test cases have the full fault coverage as their fault rate is 1.0, depicting that all the faults are covered within the time constraint, which was set to be 43 ms.

The test case's order will be placed according to the test case's appearance in Table 7 as the first path is *T*8, *T*4, *T*2, *T*1, so that the order will be *T*8, *T*4, *T*2, *T*1. Then we select the second path *T*8, *T*4, *T*3, *T*6, as we have chosen *T*8, *T*4, in the solution. Therefore it will not be added again. So the order will become *T*8, *T*4, *T*2, *T*1, *T*3, *T*6. Continue this method until the last path is reached, as given in Table 7. If the test case is not present in Table 7, the remaining test case based on the descending order of Priority_Ti of the test cases is selected. Finally, the resulting order is *T*8, *T*4, *T*2, *T*1, *T*3, *T*6, *T*7, *T*9, *T*5, *T*10 for project 1.

4.2 Demonstration for project 2

The project "Student Admission System" took the experimental setup and the test suite for the research and analysis. The definition and specifications will be available on the website "http:/www.planet-source-code.com." A VB project has been chosen for our study and analysis purposes. The experiment is conducted on the Windows XP operating system with 2 GHz CPU (Pentium dual-core processor) and RAM of 2 GB. The selected application software is tested manually and using the automated tool QTP 9.2. The time taken to execute may vary, as it depends on the computing resources is used, like the speed of the processor, clock timing, size of RAM, microprogramming instructions, etc. There is a total of 10 faults seeded into the application at compile time by using compile-time injections. The provided test cases are executed on the program to detect all the flaws. We generate a random number using the "C" language. The random order is used to create eight different test cases. The generated test cases are implemented, and the time required in detecting each fault is recorded. Table 8 represents the fault matrix that contains the sample data with other data like the number of faults detected by every test case and the time required to cover faults.

The inputs and the stopping criterion would be the same as narrated in project 1. The above table depicts the faults covered by the test cases, the time taken to execute each of them (in milliseconds), and the remaining execution time (in milliseconds) for individual test cases. The optimum solution can be obtained by using bee optimization alongside the fuzzy law. Table 9 will display the algorithm

Table 8 Test cases as *T*1…*T*10 and *F*1…*F*10 as faults, "χ" represent the faults addressed by the *T*

Test cases/Faults	*T*1	*T*2	*T*3	*T*4	*T*5	*T*6	*T*7	*T*8
*F*1		χ	χ			χ		χ
*F*2	χ			χ				χ
*F*3		χ			χ		χ	
*F*4	χ			χ				
*F*5			χ					
*F*6					χ		χ	
*F*7	χ		χ			χ		
*F*8			χ				χ	
*F*9	χ			χ			χ	
*F*10					χ			
No. of faults detected	4	2	4	3	3	2	3	2
Execution time (ms)	7	4	5	4	4	3	4	2
Rem. Exec time	15	18	17	18	18	19	18	20

Table 9 Observations for the first iteration

Test cases	Fault	Execution time (ms)	Remaining E.T (ms)	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_1	4	7	15	0.4	0.6818	0.2521
T_2	2	4	18	0.2	0.8182	0.1607
T_3	4	5	17	0.4	0.7727	*0.2636*
T_4	3	4	18	0.3	0.8182	0.2195
T_5	3	4	18	0.3	0.8182	0.2195
T_6	2	3	19	0.2	0.8636	0.1624
T_7	3	4	18	0.3	0.8182	0.2195
T_8	2	2	20	0.2	0.9091	0.1639

progress in table format and cover all instances of fault as it is necessary to remember that the time constraint is also to be taken into consideration. The time constraint here is 22 ms.

The first iteration (Table 9) is carried out, and the values are computed using Eq. (5), (6), (7), and (8). The Priority_Ti column will give us the test case required to proceed by taking the test case with the highest priority.

The first iteration gives us the test case gives us the first highest prioritized test case.

The test case *T*3 has the highest priority among all test cases with a priority of **0.2636**. As the stopping criterion is not met, it will be carried on to the next iteration, where there will be new test cases with two test cases combined, as given in Table 10.

The second iteration gives another test case with the highest priority, which is: *T*3, *T*4.

With priority as **0.3204**, the next iteration will give us a new set of test suites combined with the before mentioned test case, as shown in Table 11.

The final iteration is the third one, which stops because all of the fault cases are covered. The selected test case is the one that has the highest priority, just like in every other iteration. So, here the fixed test case is *T*3, *T*4, *T*5 with the priority value is **0.2903**.

The iterations stop as both the criterion is met, the time (22 ms), and the coverage constraints. Table 12 shows all the possible solutions under the time constraint and covers all ten fault cases.

The final optimal path is found out using the same table that follows the path covered by the forager bees to get to the food source.

The order of the test case will be arranged according to the test case's appearance in Table 12. Like *T*3, *T*4, *T*5 are in the first case, the order for the solution will be *T*3, *T*4, *T*5; the second path is *T*3, *T*4, *T*8, *T*5, so we have take *T*3, *T*4, *T*5 the add *T*8 to the order *T*3, *T*4, *T*5, *T*8 and continue the method. So, the final possible solutions will be *T*3, *T*4, *T*5, *T*8, *T*7, *T*1, *T*6, *T*2.

This path should be covered by the forager bee to reach the food source efficiently.

5 Results and discussion

In this section, the analysis of the result has been discussed.

5.1 Measuring the effectiveness of a prioritized test suite

The metric APFD is used to get a fair comparison. This algorithm's primary goal is to speed up the rate of fault detection in regression testing. The quantification of the target can be accomplished by using the APFD metric, which measures the weighted average percentage of

Table 10 Observations for the second iteration

Test cases	Fault	Execution time (ms)	Remaining execution time (ms)	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_3, T_1	7	12	10	0.7	0.4545	0.2756
T_3, T_2	5	9	13	0.5	0.5909	0.2708
T_3, T_4	7	9	13	0.7	0.5909	*0.3204*
T_3, T_5	7	9	13	0.7	0.5909	0.3204
T_3, T_6	4	8	14	0.4	0.6364	0.2456
T_3, T_7	7	9	13	0.7	0.5909	0.3204
T_3, T_8	5	7	15	0.5	0.6818	0.2885

Table 11 Observations after the final iteration

Test cases	Fault	Execution time (ms)	Remaining E.T. (ms)	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_3, T_4, T_1	7	16	6	0.7	0.2727	0.1962
T_3, T_4, T_2	8	13	9	0.8	0.4091	0.2707
T_3, T_4, T_5	*10*	*13*	*9*	*1.0*	*0.4091*	*0.2903*
T_3, T_4, T_6	7	12	10	0.7	0.4545	0.2756
T_3, T_4, T_7	9	13	9	0.9	0.4091	0.2812
T_3, T_4, T_8	7	11	11	0.7	5	0.2917

Table 12 All the possible solutions

Test cases	Fault	Execution time (ms)	Remaining execution time (ms)	μ_{FC_Ti}	μ_{RET_Ti}	Priority_*Ti*
T_3, T_4, T_5	10	13	9	1.0	0.4091	0.2903
T_3, T_4, T_8, T_5	10	15	7	1.0	0.3182	0.2414
T_3, T_5, T_7, T_8	10	15	7	1.0	0.3182	0.2414
T_3, T_5, T_1	10	16	6	1.0	0.2727	0.2143
T_3, T_5, T_7, T_4	10	17	5	1.0	0.2273	0.1852
T_3, T_4, T_8, T_6, T_5	10	18	4	1.0	0.1818	0.1538
T_3, T_7, T_1, T_5	10	20	2	1.0	0.0909	0.0833

detected faults during execution. Most researchers have used this metric to solve the TCP problem, and it is suggested by Elbaum et al. (2000a, b). The value range is from 0 to 100, where the higher value indicates a higher degree of fault recognized. If the plot is drawn, the x-axis represents the percentage of the test suite running, and the y-axis represents the percentage of faults found. The area below the curve gives the value of the APFD. It is described mathematically, in Eq. 8, as shown below:

$$\text{APFD} = 1 - \left(\frac{\text{TF}_1 + \text{TF}_2 + \cdots + \text{TF}_m}{m * n}\right) + \left(\frac{1}{2 * n}\right) \quad (8)$$

where $\text{TF}_i \rightarrow$ Place of the first test in Test suite T showing fault i, $m \rightarrow$ Maximum number of defects found by T, $n \rightarrow$ Maximum number of test cases in a T

5.2 Existing prioritization techniques used in this paper

This paper's existing prioritization techniques to contrast the result with the proposed approach have been taken from the current literature. The description of the nine prioritization techniques that have been used in the work and discussion section is given below:

(*PT1*) No prioritization-NP (Rothermel et al. 2001). In this method, no prioritization approach has been used. The test cases in the test suite remain untreated throughout the testing process. This acts as a control for experimentation.

(*PT2*) Random prioritization-RP (Rothermel et al. 2001). The accomplishment of our testing goal depends upon the initial construction of the untreated test suite. In this case, the test cases are randomly ordered in a test suite. This acts as an additional control in our studies.

(*PT3*) Reverse prioritization-REP (Rothermel et al. 2001). The test cases' order is in reverse as the test cases are initially placed in the untreated test suite. This also acts as another control used for our study.

(*PT4*) Previous work on prioritization (Kavitha and Sureshkumar 2010). The prioritization method has considered the factors: rate of fault detection and fault impact to design the algorithm. This algorithm is based on the weighted value of the two factors that enhance the fault detection rate to some extent. The proposed method could not be able to obtain the optimal ordering.

(*PT5*) Previous work on prioritization (Tyagi and Malhotra, 2015). The three factors that influence the prioritization algorithm used in this paper: rate of fault detection, percentage of fault detected, and risk detection ability. This gives a better result than the previously mentioned prioritization method. Still, they have not considered the algorithmic complexity (NP-Complete) of the function in the problem statement section named "f" in this paper.

(*PT6*) Previous work on prioritization (Nayak et al. 2016). The algorithm used to prioritize the tests that take the weighted value of two parameters: fault rate and fault severity. Still, the technique is not able to give higher costs (optimal ordering). It remains an issue for this

algorithm that for every test suite, the combination of parameters needs to be changed to obtain the optimal order.

(*PT7*) *Optimal prioritization-OOP* The optimal ordering never performs worse than the orders that have been mentioned in the paper. If any algorithm achieves the optimal order, then that algorithm is efficient enough to detect the faults faster than other algorithms. The proposed technique in this paper gives the optimal ordering in two instances.

5.3 Results and comparative study

This section of the paper describes the comparative analysis between the suggested technique and various established techniques. Each algorithm's performance is determined for the option of a better performing algorithm.

5.3.1 Result analysis of project 1

The suggested method is contrasted with diverse prioritization approaches such as NP, REP, RP, and existing techniques contributed in the literature (Kavitha and Sureshkumar 2010; Tyagi and Malhotra 2015; Nayak et al. 2016). Such methods are compared for each methodology by computing APFD (average percentage of faults detected) values.

5.3.1.1 Comparison to earlier research work The proposed priority order in this section is contrasted with the preceding work of Kavitha and Sureshkumar (2010), Tyagi and Malhotra (2015), and Nayak et al. (2016). Table 13 indicates the current order of test cases and the priority order suggested by Kavitha and Sureshkumar (2010), Tyagi and Malhotra (2015), and Nayak et al. (2016) for the same dataset of test cases.

5.3.1.2 Comparison with other approaches for prioritization The suggested method is contrasted with other methods of prioritization, such as NP, REP, and RP. Table 14 reflects the ordering of test cases for various priority techniques.

Table 15 summarizes the APFD percentage for each of the prioritization strategies. The suggested technique's APFD percentage is the same as the OOP technique, and it is higher than the other methods used before. It proves the reliability and efficacy of detecting software faults in a minimum available time.

The APFD percentage for the above-introduced techniques is represented in Fig. 2a–e, respectively.

Table 14 Various prioritization techniques and their test case sequences

Prioritization techniques	Test case sequence
No prioritization	*T*1, *T*2, *T*3, *T*4, *T*5, *T*6, *T*7, *T*8, *T*9, *T*10
Random prioritization	*T*2, *T*9, *T*5, *T*7, *T*4, *T*6, *T*8, *T*10, *T*3, *T*1
Reverse prioritization	*T*10, *T*9, *T*8, *T*7, *T*6, *T*5, *T*4, *T*3, *T*2, *T*1
Proposed order	*T*8, *T*4, *T*2, *T*1, *T*3, *T*6, *T*7, *T*9, *T*5, *T*10

5.3.2 Result analysis of project 2

The suggested method contrasts with three current prioritization methods, such as the NP, RP, and REP. These strategies will be evaluated by measuring the per technique of APFD value. The comparative analysis is summarized in Table 16.

The APFD percentage for each prioritization techniques are abstracted in Table 17. The APFD percentage value of the proposed method is like to be the same as the OOP approach and is higher than the other prioritization techniques. This proves the efficiency and efficacy in detecting faults from the software in minimum time.

The APFD percentage for the above-introduced techniques is represented in Fig. 3a–d, respectively.

5.3.3 Computation of total fault coverage

The capacity for fault coverage of each prioritization strategy is extensively analyzed. It deals with the least number of test cases needed to identify of all potential faults by every prioritization method.

5.3.3.1 Result analysis for project 1 No prioritization strategy requires 80% of tests to identify all of the faults present in the application. Table 18 reveals that only 40% of the available test cases are included in the proposed

Table 13 Test cases ordering for previous work (Kavitha and Sureshkumar 2010; Tyagi and Malhotra 2015; Nayak et al. 2016) and proposed approach

Prioritization techniques	Test case sequence
Prioritized order by Kavitha and Sureshkumar (2010)	*T*8, *T*4, *T*9, *T*6, *T*5, *T*2, *T*1, *T*10, *T*3, *T*7
Prioritized order by Tyagi and Malhotra (2015)	*T*8, *T*4, *T*6, *T*9, *T*2, *T*1, *T*5, *T*10, *T*3, *T*7
Prioritized order by Nayak et al. (2016)	*T*8, *T*4, *T*9, *T*6, *T*2, *T*1, *T*5, *T*10, *T*3, *T*7
Proposed order	*T*8, *T*4, *T*2, *T*1, *T*3, *T*6, *T*7, *T*9, *T*5, *T*10

Table 15 APFD percentage for the various strategies of prioritization

Prioritization techniques	APFD percentage
No order	63
Random	66
Reverse	70
Prioritized order by Kavitha and Sureshkumar (2010)	78
Prioritized order by Tyagi and Malhotra (2015)	82
Prioritized order by Nayak et al. (2016)	81
Proposed work (Optimal)	**85**

method to identify all the software faults. The remaining prioritization strategies such as RP, REP, and prior work by Kavitha and Sureshkumar (2010), Tyagi and Malhotra (2015), and Nayak et al. (2016) require 70%, 80%, 60%, 50%, and 50% of tests, respectively.

5.3.3.2 Result analysis for project 2 A comparison is made of the effectiveness of each strategy, like prioritized and non-prioritized technique. Table 19 shows that the proposed approach takes just 37.5% of the available test cases to recognize all the software flaws or faults. The remaining prioritization methods, such as NP, RP, and REP, need 70%, 80%, 60%, 50%, and 50% of tests, respectively.

5.4 Discussion

The efficiency of the proposed prioritization technique and the algorithms mentioned above (both control and prior work techniques) can be measured by two factors: rate of fault detection and percentage of test case required to detect all the faults.

Rate of fault detection This factor deals with how fast an algorithm can detect all SUT defects (system under test). The defects may be introduced to the system in two ways: artificial (injected or seeded faults manually) or natural (present due to program errors committed by the programmers unknowingly, during the integration of modules, etc. beyond the knowledge of human expertise). Here, all the studies done in this paper are injecting the faults manually, and then the experiment is performed to check the algorithms' effectiveness. APFD metric has been used to measure the effectiveness of a prioritized test suite. The higher the value of this metric, the more the algorithm is efficient in identifying the faults.

For Project 1, the proposed method is compared with six other existing prioritized techniques. The obtained APFD percentage for no prioritization, random prioritization, reverse prioritization, prioritized order by Kavitha and Sureshkumar (2010), prioritized order by Tyagi et al. (2015), prioritized order by Nayak et al. (2016), and the proposed technique are 63%, 66%, 70%, 78%, 82%, 81%, and 85% respectively. The APFD score of the prioritized technique is better than the others, and also it gives the optimal ordering. For Project 2, the proposed method is compared with three other existing prioritized techniques. The obtained APFD percentage for no prioritization, random prioritization, reverse prioritization, and the proposed approach is 76%, 75%, 68%, and 82.5%. The proposed technique outclassed all other methods and gave an optimal ordering. The performance is enhanced when a fuzzy rule base is added with the BCO algorithm.

The proposed technique gives a better APFD score that can help identify the faults quicker from the above results. It saves time, cost, and effort of the software organization, and the tester can design more efficient feedback systems. This can lead to the development of robust software.

Percentage of test cases required to detect all the faults This factor deals with the minimum number of test cases necessary to recognize all possible faults by prioritizing techniques. This is also a factor in measuring the effectiveness of any prioritized regression test suite. If the percentage of test cases required to detect all the faults is less for any prioritized order, that technique is more efficient. And if the test case needed to catch all the flaws is more for any method, then that technique is not that efficient.

For Project 1, the percentage of test cases required to detect all the faults for seven prioritization techniques, namely no prioritization, random prioritization, reverse prioritization, prioritized order by Kavitha and Sureshkumar (2010), prioritized order by Tyagi and Malhotra (2015), prioritized order by Nayak et al. (2016), and the proposed technique is 80%, 70%, 80%, 60%, 50%, 50%, and 40% respectively. The percentage of test cases required to detect all the proposed order faults is less than the other techniques. Therefore, the proposed technique is more efficient than others in detecting faults from the SUT in minimum time. For Project 2, the percentage of test cases required to detect all the faults for four prioritization techniques, namely, no prioritization, random prioritization, reverse prioritization, and the proposed order are 62.5%, 50%, 75%, and 37.5%, respectively. The proposed

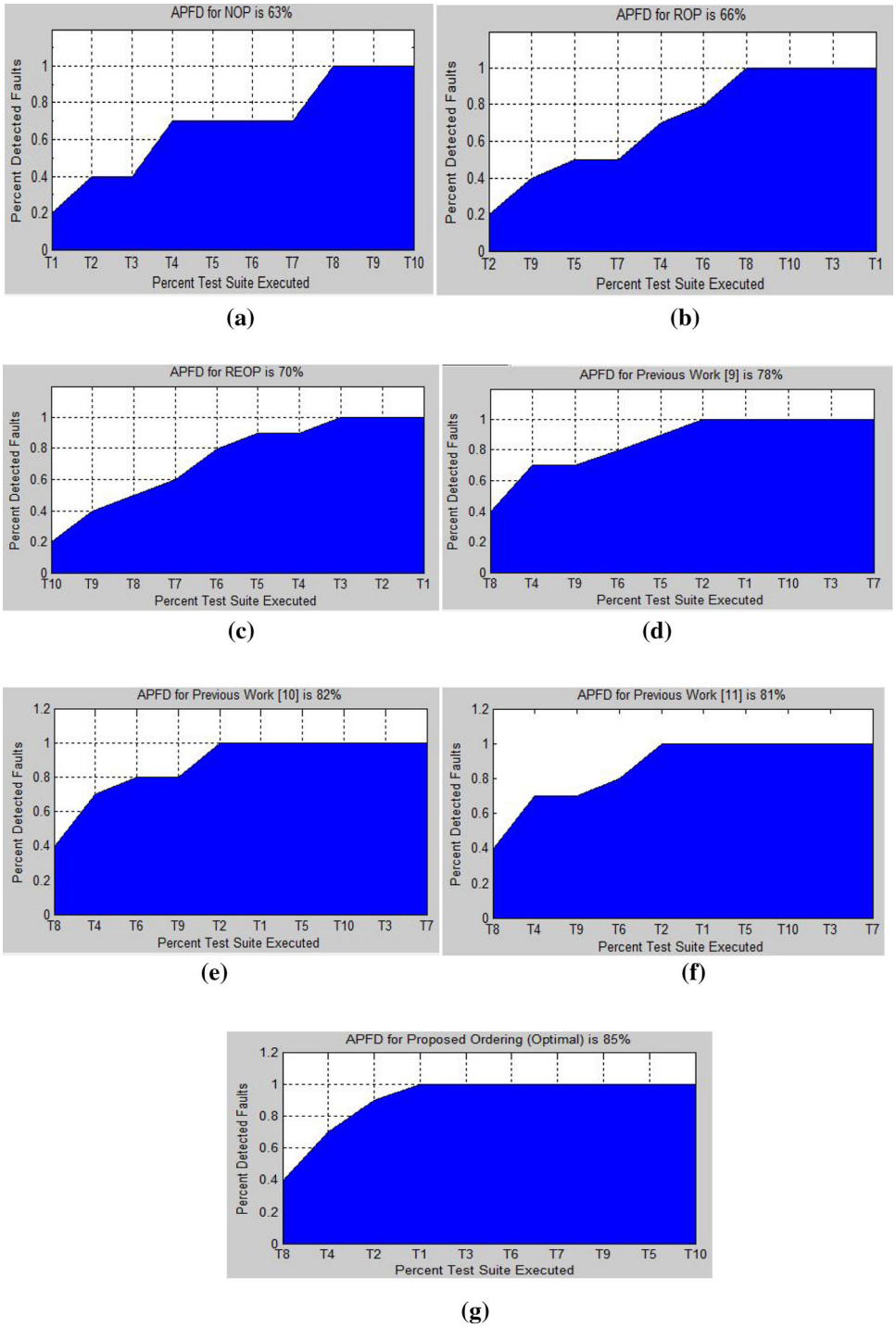

Fig. 2 Resemblance of various prioritization methods: **a** No order, **b** Random order, **c** Reverse order, **d** Prioritized order by Kavitha and Sureshkumar (2010), **e** Prioritized order by Tyagi and Malhotra (2015), **f** Prioritized order by Nayak et al. (2016), and **g** Proposed work

technique is far better than the others while locating faults in less time.

5.4.1 Discussion on research questions

This section addresses the paper's investigative questions raised in Sect. 1.

RQ1:: There are several limitations in the test case prioritization approaches developed to date. First, they take excessive prioritization times, as observed in the experiment. If the faults are detected earlier, a robust feedback system can be developed to save valuable computing resources. The seven prioritization techniques have been used in this paper, and the effectiveness of each method involving prioritized and non-prioritized

Table 16 Comparative study of various prioritization techniques

Prioritization techniques	Test case sequence
No prioritization	*T*1, *T*2, *T*3, *T*4, *T*5, *T*6, *T*7, *T*8
Random prioritization	*T*5, *T*7, *T*1, *T*3, *T*6, *T*2, *T*8, *T*4
Reverse prioritization	*T*8, *T*7, *T*6, *T*5, *T*4, *T*3, *T*2, *T*1
Proposed order	*T*3, *T*4, *T*5, *T*8, *T*7, *T*1, *T*6, *T*2

Table 17 APFD percentage for the various strategies of prioritization

Prioritization techniques	APFD%
No prioritization	76
ROP	75
REOP	68
Proposed work (Optimal)	**82.5**

approach is contrasted. In Project 1, Table 15 summarizes the APFD percentage for all prioritization methods, like the no prioritization technique, the reverse order prioritization technique, the random order prioritization technique, the previous work (Kavitha and Sureshkumar 2010; Tyagi and Malhotra 2015; Nayak et al. 2016) and the suggested technique. It is observed that the test prioritization techniques are more efficient in fault detection than if the test suite is not prioritized. In Project 2, the test cases are so arranged that the no prioritization technique is better than the random prioritization and the reverse prioritization but is less than the proposed approach in terms of APFD values. So, it is related to the structure of the test suite that decides whether a prioritization technique should be implemented or not. It is advisable to use prioritization techniques rather than no prioritization.

RQ2:: The approach suggested in this paper increases the rate of covering the faults. Using APFD metrics, the fault detection rate is quantified. The higher the APFD value, the faster the algorithm can cover the flaws in a lesser amount of time. The APFD values of various prioritization techniques are listed in Tables 15, and 17, for Project 1, and Project 2. The more the cost, the quicker

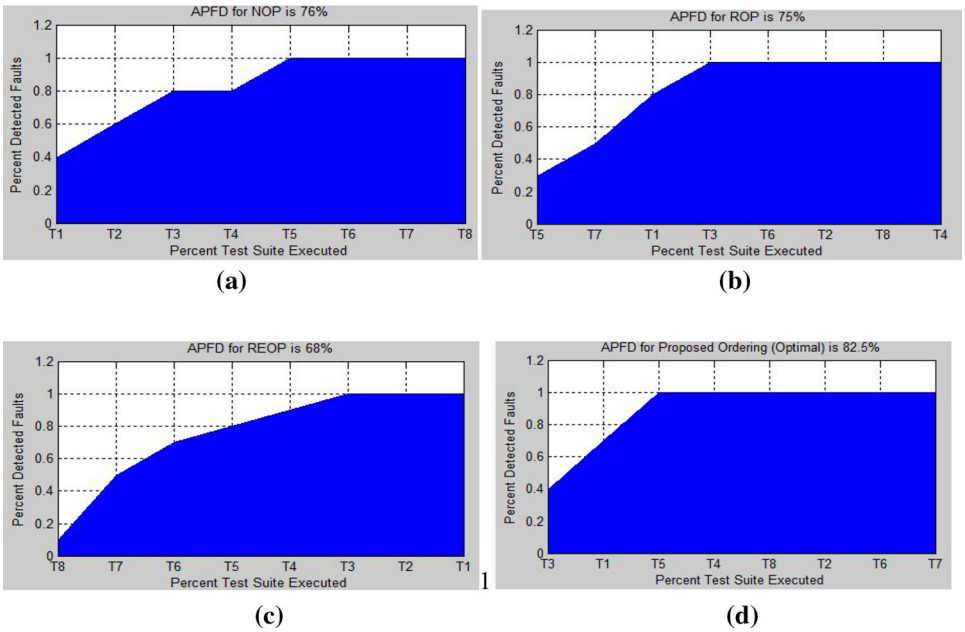

Fig. 3 Resemblance of various prioritization methods: **a** No order, **b** Random order, **c** Reverse order, and **d** Suggested work

Table 18 Percentage of test cases conducted to identify all faults linked to different prioritization techniques

Prioritization Techniques	% of tests executed to detect all faults
No	80
Random	70
Reverse	80
Prioritized order by Kavitha and Sureshkumar (2010)	60
Prioritized order by Tyagi and Malhotra (2015)	50
Prioritized order by Nayak et al. (2016)	50
Proposed order	**40**

Table 19 Percentage of test cases conducted to identify all faults linked to different prioritization techniques

Prioritization Techniques	% of tests executed to detect all faults
No prioritization	62.5
Random prioritization	50
Reverse prioritization	75
Proposed order	**37.5**

the algorithm is in detecting the faults. By the APFD values, the proposed algorithm enhances the rate of fault detection and gives a solution to the problem statement mentioned in Sect. 3 of this paper.

RQ3:: The proposed method's efficiency is contrasted with the other current prioritization strategies by taking two factors, as mentioned in the previous section. Using the extensible used metrics APFD, the fault detection rate quantification should achieve the least possible time. Tables 15, and 17 illustrates the algorithm's ability to detect faults faster than the other techniques. In this second criterion, the least likely number of test cases is considered that has the potential to recognize all the possible faults contain in the program. It is noticed in Tables 18 and 19, for Project 1 and Project 2, respectively.

5.5 Limitations of our work

There are several fallibilities in our work. Those fallibilities are described below:

- APFD metric apprehends so little aspects of ordering performance. It does not discuss other critical criteria, such as the expense of faults and test cases. While it is a commonly used metric for quantifying the rate of error detection, other metrics can also be used to determine its precision.
- To solve massive data inputs, the proposed algorithm suffers from a chaotic situation (improper exploitation).
- Because the paths are formed at random, we may often not obtain the optimal solution.

6 Conclusion and future remarks

In this paper, a new test case optimization and prioritization algorithm is proposed by considering bees' foraging behavior and a fuzzy rule base. The bee algorithm and the fuzzy rule base reduce the test cases' volume by selecting the test cases from the pre-existing test suite. The proposed approach is implemented on two projects. The two project data have been extracted from the literature and the second project is a student project designed by the undergraduate students as a final year project. Promising results have been obtained after applying the method on these two projects.

The proposed approach yielded the highest APFD results among the comparison algorithms. It is detected in the experiment that the proposed method is better than the existing prioritization techniques based on fault identification rate. For Project 1, the obtained APFD percentage for no prioritization, random prioritization, reverse prioritization, prioritized order by Kavitha and Sureshkumar (2010), prioritized order by Tyagi and Malhotra (2015), prioritized order by Nayak et al. (2016), and the proposed technique are 63%, 66%, 70%, 78%, 82%, 81%, and 85%, respectively. For Project 2, the obtained APFD percentage for no prioritization, random prioritization, reverse prioritization, and the proposed approach is 76%, 75%, 68%, and 82.5%. It also requires the least number of tests to expose all the faults present in the software.

To date, this study is the first to use the honey bees foraging actions with a fuzzy rule base in prioritizing regression tests. This algorithm will be checked on the data collected from the repositories like real-world projects coded with different programming languages to allow the proposed approach changes.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Alakeel AM (2014) Using fuzzy logic in test case prioritization for regression testing programs with assertions. Sci World J 5:1–9

Chi J, Qu Y, Zheng Q, Yang Z, Jin W, Cui D, Liu T (2020) Relation-based test case prioritization for regression testing. J Syst Softw 163:1–18

Craig RD, Jaskiel SP (2002) Systematic software testing. Artech House Publishers, Boston

Elbaum S, Malishevsky A, Rothermel G (2000) Prioritizing test cases for regression testing. In: Proceedings the 2000 ACM SIGSOFT international symposium on software testing and analysis, Portland, Oregon, USA, pp 102–112

Elbaum S, Malishevsky AG, Rothermel G (2000b) Test case prioritization: a family of empirical studies. IEEE Trans Softw Eng 28(2):159–182

Erdogmus H, Morisio M, Torchiano M (2005) On the effectiveness of the test-first approach to programming. IEEE Trans Softw Eng 31(3):226–237

Gao D, Guo X, Zhao L (2015) Test case prioritization for regression testing based on ant colony optimization. In: 6th IEEE international conference on software engineering and service science (ICSESS), Beijing, pp 275–279

ISG of Software Engg Terminology (1990) IEEE standards collection, IEEE std 610.12-1990

Jeffrey D, Gupta N (2006) Test case prioritization using relevant slices. In: Proceedings of the 30th annual international computer software and applications conference (COMPSAC 2006), Washington, DC, USA, IEEE Computer Society, pp 411–420

Jiang H (2015) Artificial bee colony algorithm for traveling salesman problem. In: 4th international conference on mechatronics, materials, chemistry and computer engineering (ICMMCCE 2015), pp 468–472

Karaboga D (2005) An idea based on honey bee swarm for numerical optimization. Technical Report TR06, Erciyes University, Engineering Faculty, Computer Engineering Department

Karaboga D, Gorkemli B, Ozturk C, Karaboga N (2013) A comprehensive survey: artificial bee colony (ABC) algorithm and applications. Artif Intell Rev 4:1–37

Kaur A, Goyal S (2011) A Bee colony optimization algorithm for fault coverage based regression test suite prioritization. Int J Adv Sci Technol 29:17–29

Kavitha R, Sureshkumar N (2010) Test case prioritization for regression testing based on severity of fault. Int J Comput Sci Eng 2(5):1462–1466

Khatibsyarbini M, Isa M, Jawani D, Hamed H, Suffian M (2019) Test case prioritization using firefly algorithm for software testing. IEEE Access 7:132360–132373

Korel B, Koutsogiannakis G, Tahat LH (2007) Model-based test prioritization heuristic methods and their evaluation. In: Proceeding A-MOST'07 proceedings of the 3rd international workshop on advances in model-based testing, ACM, pp 34–43

Krishnamoorthi R, Mary SA (2009) Factor oriented requirement coverage based system test case prioritization of new and regression test cases. Inf Softw Technol 51(4):799–808

Li Z, Harman M, Hierons (2007) Search algorithms for regression test case prioritization. IEEE Trans Softw Eng 33(4):225–237

Malishevsky A, Ruthruff JR, Rothermel G, Elbaum S (2006) Cost-cognizant test case prioritization. Technical report TR-UNL-CSE-2006-004, Department of CSE, University of Nebraska-Lincoln, USA

Malz C, Jazdi N, Göhner P (2012) Prioritization of test cases using software agents and fuzzy logic. In: Proceedings of fifth international conference on software testing, verification and validation, IEEE, pp 483–486

Mounika M, Reddy DV (2015) Test case selection for path testing using bee colony optimization. Elysium J 2(1):1–7

Mukherjee R, Patnaik S (2019) Introducing a fuzzy model for cost cognizant software test case prioritization. In: IEEE Region 10 Conference (TENCON), pp 504–509

Nayak S, Kumar C, Tripathi S (2016) Effectiveness of prioritization of test cases based on faults. In: Proceedings of third international conference on recent advances in information technology (RAIT-2016), IEEE

Nayak S, Kumar C, Tripathi S (2017) Enhancing efficiency of the test case prioritization technique by improving the rate of fault detection. Arab J Sci Eng 42(8):3307–3323

Rotherrmel G, Harrold MJ, Ostrin, J, Hong C (1998) An emprical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of the international conference on software maintenance, pp 34–43

Öztürk M (2018) A bat-inspired algorithm for prioritizing test cases. Vietnam J Comput Sci 5(1):45–57

Pan Q-K, Fatih TM, Suganthan PN, Chua TJ (2011) A discrete artificial bee colony algorithm for the lot-streaming flow shop scheduling problem. Inf Sci 181(12):2455–2468

Ray M, Mohapatra DP (2014) Multi-objectve test prioritization via a genetic algorithm. Innov Syst Softw Eng 10:261–270

Rothermel G, Untch R, Chu C, Harrold M (2001) Prioritizing test cases for regression testing. IEEE Trans Software Eng 27(10):929–948

Shrivathsan A, Ravichandran K, Krishankumar R, Sangeetha V, Kar S, Ziemba P, Jankowski J (2019) Novel fuzzy clustering methods for test case prioritization in software projects. Symmetry MDPI 11(11):1–22

Silva DS, Rabelo R, Neto PS, Britto R, Oliveira PA (2019) A test case prioritization approach based on software component metrics. In: IEEE international conference on systems, man and cybernetics (SMC), pp 2939–2945

Srikanth H, Williams L, Osborne J (2005) System test case prioritization of new and regression test cases. In: International symposium on empirical software engineering (ISESE 2005), Noosa Heads, Australia, pp 64–73

Srikanth A, Kulkarni NJ, Naveen KV, Singh P, Srivastava PR (2011) Test case optimization using artificial bee colony algorithm. In: Communications in computer and information science (CCIS-2011), pp 570–579

Srikanth H, Cashman M, Cohen MB (2016) Test case prioritization of build acceptance tests for an enterprise cloud application: an industrial case study. J Syst Softw 119:122–135

Suri B, Singhal S (2011) Implementing ant colony optimization for test case selection and prioritization. Proc Int J Comput Sci Eng 3(5):1924–1932

Sushant K, Ranjan P (2017) ACO based test case prioritization for fault detection in maintenance phase. Int J Appl Eng Res 12(16):5578–5586

Tyagi M, Malhotra S (2015) An approach for test case prioritization based on three factors. Int J Inf Technol Comput Sci 4:79–86

Yoo S, Harman M (2007) Regression testing minimisation, selection and prioritisation: a survey. Softw Test Verif Reliabil 2:1–60