Soft Computing (2022) 26:8203–8220
https://doi.org/10.1007/s00500-022-07236-z

Dominating set-based test prioritization algorithms for regression testing

Zafer Can Demir[1,2] · Şahin Emrah Amrahov[1]

Accepted: 18 May 2022 / Published online: 29 June 2022

Abstract
In this study, we consider test case prioritization (TCP) problem for regression testing. Regression tests are used to ensure that software updates do not affect other software functionality. In the process of regression tests, the tests previously used for software testing are run again. However, running all existing tests is a costly process as it will cause time and labor loss. Therefore, an optimization can be made by giving priority to the tests according to certain criteria and running the tests with high priority values. This reduces the burden of unnecessary tests and regression tests reach their goals more quickly. In this study, we show that TCP problem is equivalent to the problem of finding dominating sets for the bipartite graphs. We propose 3 algorithms based on dominating sets and then we compare experimental results of the proposed algorithms with the results of the existing algorithms.

Keywords Test case prioritization · Regression tests · Test case prioritization algorithms · Dominating set

1 Introduction

The software life cycle is a process of developing software and covering all the maintenance stages after the delivery of software to the customer. This process consists of planning, analysis, design, production, testing and maintenance phases. It is the testing phase that concerns us in this study. The testing phase is one of the important steps in the software life cycle. Software testing is the process of finding errors in software that can cause loss of money, time, trust or life, prevent the software from working normally or cause it to stop working at all. Today, the differentiation of software requirements and variation in features has made it almost impossible to produce error-free software. Errors in the software may reduce the customer's interest in the software and cause loss of trust as well as other costs. Sources of these errors can be requirement, design, software, inadequate testing, or other updates to the code. The more errors are detected at a later stage in the software development process, the more it will cost. For this reason, it is very important at what stage the errors are detected. For example, if the error is noticed at the requirement stage, the cost is not much, but it increases exponentially during the design, coding and usage stages. The goal of the software testers is to find bugs at the earliest possible stage and to make sure that these errors are eliminated. In previous years, while the testing phase was not given much importance in the software development process, software testing has now taken an important place in this cycle. The main reason for this is that the cost of presenting a product with errors to the customer is increasing. One of these costs is the labor and time required to find the source of the error and correct it. Another cost is the financial losses that occur as a result of the error that can be invoiced to the company on the user side. Apart from these, a cost is also incurred as a result of the brand image negatively affected.

The remainder of this paper is organized as follows. The research background is given in Sect. 2. Some related work is discussed and three proposed algorithms are explained in Sect. 3. The results of our empirical studies and experimental analysis are reported in Sect. 4. And the conclusion is presented in Sect. 5.

✉ Şahin Emrah Amrahov
emrah@eng.ankara.edu.tr

Zafer Can Demir
zcdeli@aselsan.com.tr

1 Computer Engineering Department, Ankara University, Ankara, Turkey

2 ASELSAN A. Ş, Ankara, Turkey

2 Background

When testing software, different methods are used according to the design and purpose of the software. These methods can generally be grouped under two headings. These are functional tests and non-functional tests (Luo 2001; Malik et al. 2013).

2.1 Non-functional tests

The following examples can be given to these tests.

- **Performance Test:** The application is tested with many data and many users.
- **Load Test:** When the application is tested with multiple users and data, it is checked whether it can respond in the specified time.
- **Stress Test:** When performance and load tests are performed, the accuracy of the returned results is checked.
- **Security Test:** These are the tests performed on the parts with safety risks.

2.2 Functional tests

The following examples can be given to these tests.

- **Unit Test:** It is the testing of software units. The software unit mentioned here is the smallest software component that can be tested and is named by names such as unit or module. Classes are examples of these units. A specific value is sent to the function under test and it is checked whether the returned result matches the expected result.
- **Integration Test:** It is the advanced version of unit test. While a single function is tested in the unit test, all interrelated functions are tested here.
- **Interface/UI-User Interface Test:** The interfaces of the application are tested. The aim is to provide visuality and functionality for users at the same time.
- **System Test:** Not only the application but also the hardware, software and services are fully tested.
- **Regression Test:** Testing whether a change made in software affects another part of the software.

2.3 Regression tests

Let’s explain the regression tests included in functional tests in some detail.

According to the software development life cycle, the test phase begins after the software is designed and the implementation phase is completed (Li et al. 2007). In this cycle, the testing and retesting processes are repeated over and over again to find errors as early as possible (Jeffrey and Gupta 2006). With the rapid development in information technologies, software testing has become more and more important as a software quality assurance (Jun et al. 2011). Regression test is one of the methods used in the test phase. While the existing tests are used in regression tests, in other test methods new tests are developed by the testers.

High-quality software cannot be developed without passing through careful testing stages. As the size and complexity of modern software increased, the importance of regression tests increased (Mohapatra and Prasad 2013). The purpose of regression testing is to ensure that software changes made for correction or improvement do not in any way affect other software functions that were previously working correctly (Ansari et al. 2016). Whenever a change is made in the software, a certain test group should be run to prove that this change does not affect other parts of the software. The previously run tests are rerun after the software change to make sure that the changes made do not cause new errors in the software. In this way, while solving existing errors, new errors are tried not to be created. Regression tests are required when the requirement changes and the code is updated accordingly, a new feature is added and an improvement is made regarding performance (Ansari et al. 2016). Regression tests are repeated throughout the software life cycle.

Regression tests check the following 4 situations (Navdeep Jain and Gambhir 2015).

- Errors
- Software updating
- Fix defects
- Performance issues

As mentioned above, regression tests check for errors, verify changes to be made with software updates, update of one part does not cause errors in another part and application performance does not decrease with updates.

Regression testing is a software development process, an important part of software maintenance and requires a significant budget. It ensures that the changes made to the software are correct and that fragments of the software that are not updated are not affected by other changes. It is not very reasonable to run every test for the software in this process. Because as the number of tests increases, the effort and time to be spent will also increase. Also, there may not be enough time and resources to run all tests. Some techniques have been developed to reduce this cost while running the tests.

The following methods can be used for regression tests (Srivastava 2008; Singh et al. 2010; Ansari et al. 2016).

- **Retest All:** In this method, all tests created for the software must be retested, that is, these tests must be re-run. The tests that cannot be applied to the updated software

are removed and all remaining tests are run again. However, it may take a long time to run all tests.
- **Regression Test Selection:** It is done by selecting a certain part of the tests instead of re-running all tests. Test cases are categorized as reusable and obsolete test cases. A subset of the test pool is selected so that only reusable tests can be used in subsequent regression processes.
- **Test Suite Reduction:** Test cases that become redundant as new functionalities are added to the software are removed. Unlike the regression test selection, the tests that are not needed in this technique are completely removed, while in the other technique, only the necessary tests are selected instead of removing the tests. The advantage of this is to reduce the costs of verification, running and maintenance of tests in future releases. As a disadvantage, it can be shown that the error detection capacity of the tests is reduced.
- **Test Case Prioritization:** It is the process of giving priority value to tests according to predetermined criteria suitable for a specific purpose. In this way, instead of leaving the tests out of use, less useful tests are used later. Tests with high priority are run first. The criteria mentioned are error detection rate, maximum code coverage or coverage of important features earlier, etc. If there is a limited time to run tests, the running time of the tests can also be added as a criterion.
- **Hybrid Approach:** It is the combination of test case prioritization and regression test selection methods.

2.4 Test case prioritization

Regression tests are used to verify that a changed software is working properly. Tests need to be prioritized as changing software is likely to come with new bugs and rerun all tests is costly in terms of time, effort and budget (Konsaard and Ramingwong 2015). In such cases, test case prioritization algorithms increase the effect of the regression test by putting existing tests in a certain order and ensure that the most important ones are run first. Thus, the error detection process can be done earlier, and feedback can be given about the quality of the system in the testing process. As a matter of fact, the purpose of regression tests is to determine whether the modified software contains an error that will affect the previous version and to verify that the changes were made without errors (Krishnamoorthi et al. 2009).

One of the domains where test case prioritization algorithms are required is run time. In particular in cases where the time determined for the test run is short, the tests that can be run within this period should be determined. The other domain is related to code coverage. Software testers may want to sort the tests to make the code coverage in a shorter time. Thus, 100% code coverage can be achieved earlier or before a specified time and at a higher speed (Krishnamoorthi et al. 2009). Another domain that can be used is the requirement coverage ratio. Requirements labeled as high priority by customer or changed should be covered before (Kavitha et al. 2010). If the tests are sorted according to the error coverage rate, it is understood that the first test to be run covers the most errors (Farooq and Nadeem 2017). If the criteria in the ranking is total statement coverage prioritization, the tests are ordered in descending order according to the number of cases covered. In additional statement coverage prioritization, tests are sorted by the number of cases that have not been covered before (Jeffrey and Gupta 2006). In some techniques, prioritization is made using the information obtained from the past runs of test cases (Kim and Porter 2002). By prioritizing test cases covering errors that cause new errors, interdependent or more important errors than others, problems that affect the operation of the software at the highest degree can be eliminated (Kayes 2011; Wang et al. 2015). Prioritization can also be made by considering the possibility of a test case failing in a version of the software to fail in a new software version (Lin et al. 2013). Apart from these, solutions can be provided with hybrid algorithms by considering more than one criteria (Kaur and Bhatt 2011). Code combinations coverage is a novel coverage criterion introduced by Huang et al. (2020), which combines the concepts of code coverage and combination coverage. Elbaum et al. (2002) compared several test prioritization techniques known at the time and Elbaum et al. (2004) investigated which types of prioritization techniques are appropriate and their eligibility conditions under certain test scenarios.

As a result of the test case prioritization, the tests are sorted according to certain criteria. The aim here is to reach the determined goal in a shorter time and with less cost than before the ordering (Ansari et al. 2016). Test case prioritization techniques increase the efficiency of the testing process by ranking existing tests so that the most useful ones run first (Srivastava 2008).

Thanks to test case prioritization;

- Error detection rate by the test team increases.
- High risk errors are detected earlier in the cycle.
- Regression errors due to a particular code snippet are detected early in the testing process.
- Code coverage speed is increased.
- The system becomes more reliable (Rothermel et al. 1999).

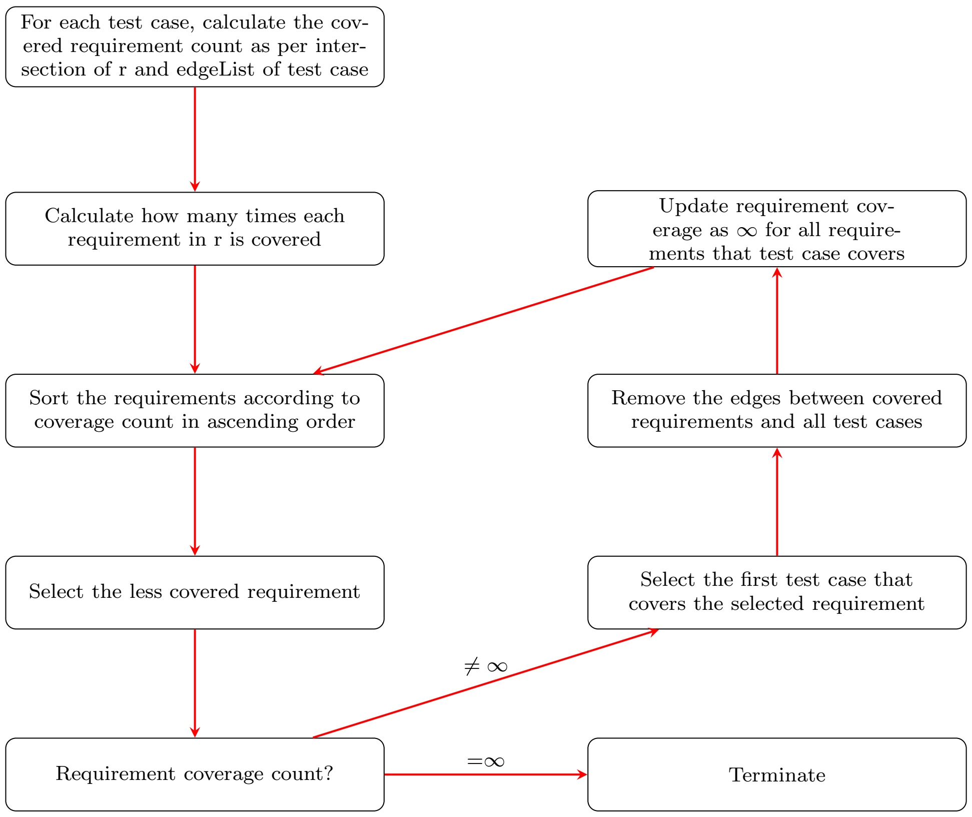
For each test case, calculate the covered requirement count as per intersection of r and edgeList of test case
Calculate how many times each requirement in r is covered
Sort the requirements according to coverage count in ascending order
Select the less covered requirement
Requirement coverage count?
≠ ∞
=∞
Terminate
Select the first test case that covers the selected requirement
Remove the edges between covered requirements and all test cases
Update requirement coverage as ∞ for all requirements that test case covers

Fig. 1 H1 - LessCoveredReqFirst

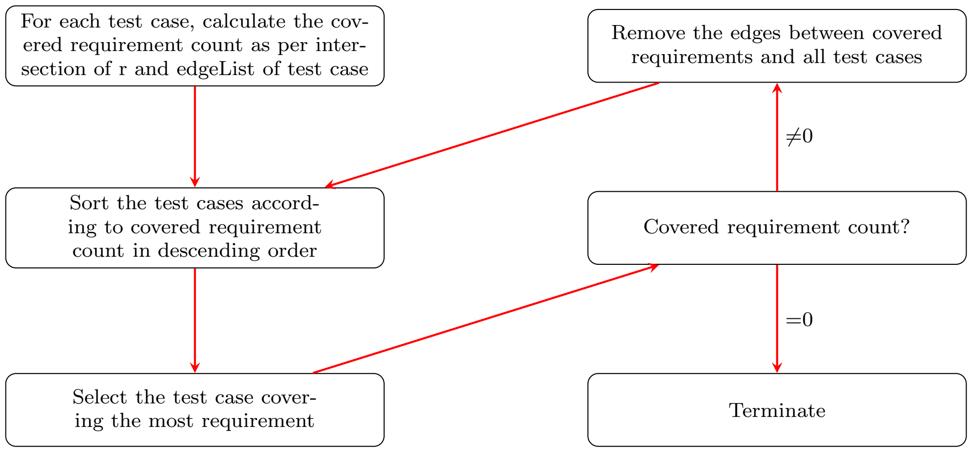
For each test case, calculate the covered requirement count as per intersection of r and edgeList of test case
Sort the test cases according to covered requirement count in descending order
Select the test case covering the most requirement
Covered requirement count?
≠0
=0
Terminate
Remove the edges between covered requirements and all test cases

Fig. 2 H2 - MostCoveringTestCaseFirst

2.4.1 Algorithms used in test case prioritization

Examples of algorithms used in test case prioritization are:

- Greedy Algorithm: The greedy approach is based on the thought of looking for the "next best." An example of these algorithms not working properly was given in the (Li et al. 2007) study.
- Additional Greedy Algorithm: Although it is similar to the greedy algorithm, it works with a different strategy. In this algorithm, the element with the highest weight is selected, which includes the cases not included as a result of previous selections. Therefore, with each selection result, the coverage status of the remaining elements is updated (Li et al. 2007).
- 2-Optimal Algorithm: This algorithm is a variation of the K-Optimal Greedy approach. In this algorithm, K elements that solve the biggest part of the problem are taken together. In K-Optimal Additional Greedy, K element that solves the remaining part of the problem is taken. K is chosen as 2 in the 2-Optimal Algorithm (Li et al. 2007).

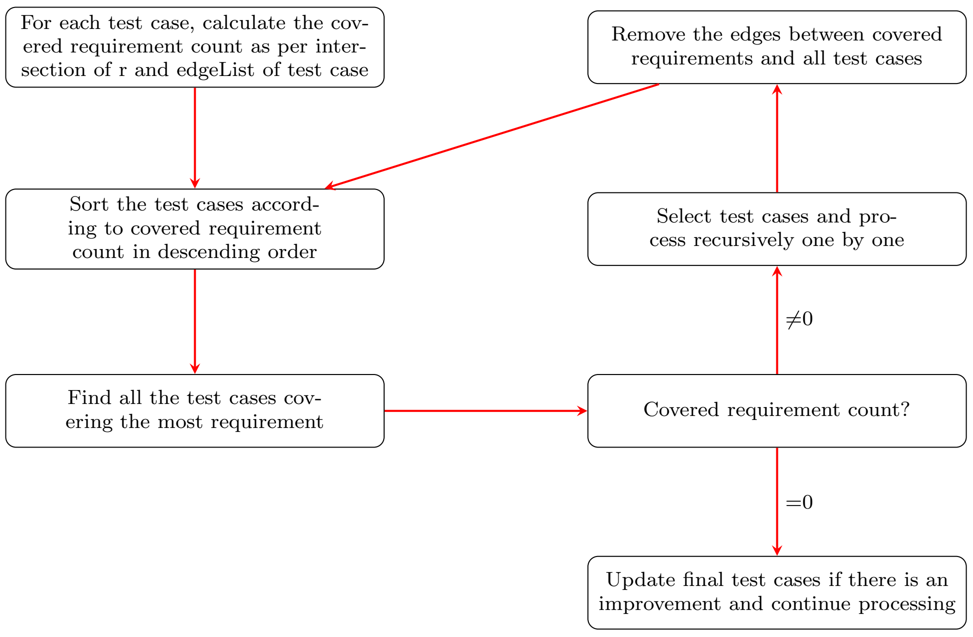
For each test case, calculate the covered requirement count as per intersection of r and edgeList of test case
Sort the test cases according to covered requirement count in descending order
Find all the test cases covering the most requirement
Covered requirement count?
≠0
=0
Update final test cases if there is an improvement and continue processing
Select test cases and process recursively one by one
Remove the edges between covered requirements and all test cases

Fig. 3 H3 - MostCoveringTestCaseFirst_Dyn

- Metaheuristic Algorithms: For example, genetic algorithm, ant colony algorithm, particle swarm optimization (Krishnamoorthi et al. 2009; Singh et al. 2010; Hla et al. 2008).
- K-Means Clustering : The purpose of this method is to detect test cases that are similar to each other. Using predetermined features and attributes to compare test cases, the similarities of test cases in the test suite are determined. These determined features are analyzed by using a matrix and similar features are removed in order not to cause overfitting problem. After the number of clusters is determined, K-means clustering is being used to cluster the test cases so that test cases with similar characteristics are in the same cluster. Then, the prioritization technique is applied to the clusters. Ranking is utilized for prioritizing the test cases inside every cluster formed. As a result of this ranking, some test cases will be prioritized (Gokilavani and Bharathi 2021).

There are also studies other than the algorithms mentioned above. In one of the first studies on test case prioritization, the problem is transformed into an objective function maximizing problem (Wong et al. 1997). Chen et al. (2018) proposed a clustering-based adaptive random sequence approach for object-oriented software. In a time-constrained environment, Panwar et al. (2018) introduced the Cuckoo Search (CS) algorithm followed by the Modified Ant Colony Optimization (M-ACO) approach to complete the test cases in an optimum sequence. Bian et al. (2017) proposed Epistatic Test Case Segment (ETS) for multiobjective search-based regression Test Case Prioritization (MoTCP), in view of epistasis theory which mirrors the gene correlation in evolution. Bagherzadeh et al. (2021) investigated the use of Reinforcement Learning (RL) to prioritize test cases in a Continuous Integration (CI) context.

When the algorithms mentioned above are compared, the algorithms have shown similar performances in trials with small number of test cases. As the number increases, performance differences have begun to occur. Since greedy algorithms are deterministic, they will give the same result when they are tried several times. But other algorithms (genetic algorithm, etc.) will give different results. According to the determined results, it can be said that the genetic algorithm performs better than other algorithms (Li et al. 2007).

3 Test case prioritization with the dominating set method

3.1 Dominating set

For a given graph G=(V,E) let D be a non-empty subset of the V. The set D is called a dominant set if each vertex from V is in D or it is adjacent to a vertex from D. The dominant set with the least number of elements is called a minimum dominant set. To find a minimum dominant set of a given graph is a NP-Hard problem and many algorithms have been developed for finding approximately (Liedlof 2008).

Within the scope of this study, it is assumed that the working time of the tests is equal, and the operations are made over the number of requirements that the tests trace. Accordingly, in a set of tests defined as X and set of requirements defined as Y, for any selected test x_1, if an arrow is drawn to y_1, y_2,

y_3 ... requirements that this test traces, a directed bipartite graph from set X to set Y will be formed. To solve the test case prioritization problem, it will be sufficient to find the minimum dominating set in this graph.

Various studies have been carried out on the solution of the test case prioritization problem with the graph method. For example, in one study, a graph of test cases was obtained using code coverage, test run time, and test similarity. While the code coverage and test run time affect the node value, the test similarity has determined the values of the edges between the nodes. Then, the test cases were prioritized by traversing the graph (Azizi and Do 2018). In another study, after the code was instrumented, method calls have been converted to a graph format. Later, that graph was simplified under the name of call diagram. Different values were assigned to the edges between the nodes according to the type of method calls (test-test, test-source code and source code-source code) and operations were performed on them (Chi et al. 2020). In another study, test cases were converted to graph format using dissimilarity rates. Then, the problem was tried to be solved by using different sorting algorithms in this line (Murali et al. 2008). Within the scope of this study, the problem was tried to be solved by using the relationships between tests and requirements. Requirements can actually be thought of as the title of a particular piece of source code. Based on the requirements given as traceability within the test case, it can be understood which part of the source code will be covered. With this logic, assuming tests and requirements are nodes within the graph, a relationship is established by adding a directed link between the test cases and the requirements covered by these test cases. Then, the problem was tried to be solved by applying the dominating set method to this graph.

3.2 Proposed algorithms

Within the scope of this paper, 3 algorithms have been developed to prioritize test case using the dominating set method. These algorithms are:

- H1 - LessCoveredReqFirst (Fig. 1)
- H2 - MostCoveringTestCaseFirst (Fig. 2)
- H3 - MostCoveringTestCaseFirst_Dyn (Fig. 3)

The following abbreviations will be used in the algorithm explanations.

- Test cases pool: T
- Requirements: R
- Requirements to be covered: r

Algorithm 1 H1 - LessCoveredReqFirst

```
1: procedure MAIN
2:     Input: Test cases T = $t_1$, $t_2$, ...
3:     Input: Requirements R = $R_1$, $R_2$, ...
4:     Input: Requirements to be covered r = $r_1$, $r_2$ ...
5:     Initialize: TestcaseVsReqDict = Dictionary<int, List<int>>
6:     Initialize: CoveredReqCountDict = Dictionary<int, int>
7:     Initialize: SortedCoveredReqCountDict = Dictionary<int, int>
8:     Initialize: FinalTestCases = List<int>
9:     for test cases $t_1$, ... $t_i$ in T do
10:        edgeList = $t_i$.Edges
11:        coveredRequirements = Intersection of edgeList and r
12:        if coveredRequirements.Size > 0 then
13:            TestcaseVsReqDict[$t_i$] = coveredRequirements
14:            cumulCoveredRequirements += coveredRequirements
15:    for requirements $r_1$, ... $r_i$ in cumulCoveredRequirements do
16:        if CoveredReqCountDict containsKey $r_i$ then
17:            CoveredReqCountDict[$r_i$]++
18:        else
19:            CoveredReqCountDict[$r_i$] = 1
20:    SortReqDict()
21:    ProcessTestCases()

22: procedure SORTREQDICT
23:    SortedCoveredReqCountDict = Sort CoveredReqCountDict
       according to value in ascending order

24: procedure PROCESSTESTCASES
25:    for requirements $r_1$, ... $r_i$ in SortedCoveredReqCountDict do
26:        element = SortedCoveredReqCountDict.First
27:        if element.Value $\neq \infty$ then
28:            req = element.Key
29:            testcase = SelectTestCase(req)
30:            if testcase $\neq$ -1 then
31:                UpdateGraph(testcase)
32:                SortReqDict()

33: procedure SELECTTESTCASE($req$)
34:    testcase = -1
35:    for test cases $t_1$, ... $t_i$ in TestcaseVsReqDict do
36:        if $t_i$.Value contains req then
37:            Insert $t_i$ to FinalTestCases
38:            testcase = $t_i$.Key
39:            break
40:    return testcase

41: procedure UPDATEGRAPH($testcase$)
42:    reqs = TestcaseVsReqDict[testcase]
43:    for test cases $t_1$, ... $t_i$ in TestcaseVsReqDict do
44:        remove edges between reqs and $t_i$
45:    for req $r_1$, ... $r_i$ in reqs do
46:        CoveredReqCountDict[$r_i$] = $\infty$
```

3.2.1 Explanation of algorithms on example

Below is a sample test and requirement table where the algorithms will be applied. In this table, the requirement covered by each test is marked (Table 1).

- H1 LessCoveredReqFirst Algorithm

Algorithm 2 H2 - MostCoveringTestCaseFirst

```
1: procedure MAIN
2:     Input: Test cases T = $t_1, t_2, ...$
3:     Input: Requirements R = $R_1, R_2, ...$
4:     Input: Requirements to be covered r = $r_1, r_2$ ...
5:     Initialize: TestcaseVsReqDict = Dictionary<int, List<int>>
6:     Initialize: SortedTestcaseVsReqDict = Dictionary<int,List<int>>
7:     Initialize: FinalTestCases = List<int>
8:     for test cases $t_1, ... t_i$ in T do
9:         edgeList = $t_i$.Edges
10:        coveredRequirements = Intersection of edgeList and r
11:        if coveredRequirements.Size > 0 then
12:            TestcaseVsReqDict[$t_i$] = coveredRequirements
13:    SortTestCaseDict()
14:    ProcessTestCases()

15: procedure SORTTESTCASEDICT
16:    SortedTestcaseVsReqDict = Sort TestcaseVsReqDict according to value count in descending order

17: procedure PROCESSTESTCASES
18:    for i = 0 to SortedTestcaseVsReqTableDict.Count do
19:        element = SortedTestcaseVsReqDict.First
20:        if element.Value.Count != 0 then
21:            testcase = element.Key
22:            Insert testcase to FinalTestCases
23:            UpdateGraph(testcase)
24:            SortTestCaseDict()

25: procedure UPDATEGRAPH($testcase$)
26:    reqs = TestcaseVsReqDict[testcase]
27:    for test cases $t_1, ... t_i$ in TestcaseVsReqDict do
28:        remove edges between reqs and $t_i$
```

- The number of times each requirement is covered by the test cases is determined (Table 2).
- Requirements are sorted in ascending order according to total coverage number (Table 3).
- The first test case, T2, that covers the first element R1 is selected.

Table 1 Example requirement coverage table

	R1	R2	R3	R4
T1		X	X	
T2	X	X		
T3			X	X

Table 2 Algorithm H1 - Traceability numbers of requirements

Requirement	Traceability count
R1	1
R2	2
R3	2
R4	1

Algorithm 3 H3 - MostCoveringTestCaseFirst_Dyn

```
1: procedure MAIN
2:     Input: Test cases T = $t_1, t_2, ...$
3:     Input: Requirements R = $R_1, R_2, ...$
4:     Input: Requirements to be covered r = $r_1, r_2$ ...
5:     Initialize: TestcaseVsReqDict = Dictionary<int, List<int>>
6:     Initialize: SortedTestcaseVsReqDict = Dictionary<int, List<int>>
7:     Initialize: GlobalFinalTestCases = List<int>
8:     Initialize: GlobalCount = $\infty$
9:     for test cases $t_1, ... t_i$ in T do
10:        edgeList = $t_i$.Edges
11:        coveredRequirements = Intersection of edgeList and r
12:        if coveredRequirements.Size > 0 then
13:            TestcaseVsReqDict[$t_i$] = coveredRequirements
14:    Initialize localTestCases = List<int>
15:    ProcessTestCases(TestcaseVsReqDict, localTestCases)

16: procedure PROCESSTESTCASES($tcVsReqDict, localTestCases$)
17:    sortedCoveringDict = GetSortedTestCaseDict(tcVsReqDict)
18:    firstElement = sortedCoveringDict.First
19:    if firstElement.Value.Count == 0 then
20:        if localTestCases.Count < GlobalCount then
21:            GlobalCount = localTestCases.Count
22:            GlobalFinalTestCases = localTestCases
23:        return
24:    maxCoveringElements = KeyValuePair<int, List<int>>
25:    if firstElement.Value.Count == 1 then
26:        Insert firstElement to maxCoveringElements
27:    else
28:        Insert firstElement to maxCoveringElements
29:        for test cases $t_1, ... t_i$ in sortedCoveringDict do
30:            if firstElement.Value.Count == newElement.Value.Count then
31:                if newElement has any edge other than firstElement then
32:                    Insert newElement to maxCoveringElements
33:    for test cases $t_1, ... t_i$ in maxCoveringElements do
34:        duplicateLocalTestCases = localTestCases
35:        duplicateTcVsReqDict = tcVsReqDict
36:        testcase = $t_i$.Key
37:        value = $t_i$.Value.Count
38:        Insert testcase to duplicateLocalTestCases
39:        UpdateGraph(duplicateTcVsReqDict, testcase)
40:        ProcessTestCases(duplicateTcVsReqDict, duplicateLocalTestCases)

41: procedure GETSORTEDTESTCASEDICT($tcVsReqDict$)
42:    sorted = Sort tcVsReqDict according to value count in descending order
43:    return sorted

44: procedure UPDATEGRAPH($tcVsReqDict, testcase$)
45:    reqs = tcVsReqDict[testcase]
46:    for test cases $t_1, ... t_i$ in tcVsReqDict do
47:        remove edges between reqs and $t_i$
```

- All the edges between T2 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T2 is added to the selected test cases.

  - Edges between the covered requirements (R1 and R2) and other test cases giving traceability to those requirements are deleted.
  - The coverage number of remaining requirements is determined and these requirements are sorted in ascending order (Table 4).
  - The first test case, T3, that covers the first element R4 is selected.
  - All the edges between T3 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T3 is added to the selected test cases.
  - Edges between the covered requirements (R3 and R4) and other test cases giving traceability to those requirements are deleted.
  - The algorithm ends as all requirements are covered.
  - Selected Test Cases: T2, T3
- H2 MostCoveringTestCaseFirst Algorithm
  - Test cases are ordered in ascending order according to requirement coverage number (Table 5).
  - The first element, T1 is selected.
  - All the edges between T1 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T1 is added to the selected test cases.
    dummy
  - Edges between the covered requirements (R2 and R3) and other test cases giving traceability to those requirements are deleted.
  - Using uncovered requirements (R1 and R4), the coverage numbers of test cases are determined and sorted in descending order (Table 6).
  - The first element, T2, is selected.
  - All the edges between T2 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T2 is added to the selected test cases.
  - Edges between the covered requirements (R1) and other test cases that give traceability to those requirements are deleted. Since there is no edge except R1 and T2, proceed to the next step.
  - Using uncovered requirements (R4), the coverage numbers of test cases are determined and sorted in descending order (Table 7).
  - The first element, T3, is selected.
  - All the edges between T3 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T3 is added to the selected test cases.
  - Edges between the covered requirements (R4) and other test cases that give traceability to those requirements are deleted. Since there is no edge except R4 and T3, proceed to the next step.
  - The algorithm ends as all requirements are covered.
  - Selected Test Cases: T1, T2, T3
- H3 - MostCoveringTestCaseFirst_Dyn
  - F (List to keep the final test cases), GlobalCount = ∞ (Variable to keep the number of the final test cases) and maxCoveringElements list (list to keep test cases that cover the same number of requirements as the first element) are defined.
  - Test cases are ordered in ascending order according to requirement coverage number (Table 8).
  - The first element, T1, is selected.
  - Test cases that have equal coverage number as T1 are determined and these are added to the maxCoveringElements list along with T1. (Now, maxCoveringElements contains T1, T2 and T3)
  - The following operations are done for each element in this list.

Table 3 Algorithm H1 - Sorted traceability numbers of requirements

Requirement	Traceability count
R1	1
R4	1
R2	2
R3	2

Table 4 Algorithm H1 - Sorted traceability numbers of remaining requirements

Requirement	Traceability count
R4	1
R3	2

Table 5 Algorithm H2 - Coverage numbers of test cases

Test case	Coverage count
T1	2
T2	2
T3	2

Table 6 Algorithm H2 - Coverage numbers of remaining test cases

Test case	Coverage count
T2	1
T3	1

Table 7 Algorithm H2 - Coverage number of last remaining test case

Test case	Coverage count
T3	1

Table 8 Algorithm H3 - Coverage numbers of test cases

Test case	Coverage count
T1	2
T2	2
T3	2

- **Iteration 1 (for T1):** T1 is selected.
- **Iteration 1 (for T1):** All the edges between T1 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T1 is added to the selected test cases.
- **Iteration 1 (for T1):** Edges between the covered requirements (R2 and R3) and other test cases giving traceability to those requirements are deleted.
- **Iteration 1 (for T1):** Using uncovered requirements (R1 and R4), the coverage numbers of test cases are determined and sorted in descending order (Table 9).
- **Iteration 1 (for T1):** The first element, T2, is selected.
- **Iteration 1 (for T1):** All the edges between T2 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T2 is added to the selected test cases.
- **Iteration 1 (for T1):** Edges between the covered requirements (R1) and other test cases that give traceability to those requirements are deleted. Since there is no edge except R1 and T2, proceed to the next step.
- **Iteration 1 (for T1):** Using uncovered requirements (R4), the coverage numbers of test cases are determined and sorted in descending order (Table 10).
- **Iteration 1 (for T1):** The first element, T3, is selected.
- **Iteration 1 (for T1):** All the edges between T3 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T3 is added to the selected test cases.
- **Iteration 1 (for T1):** Edges between the covered requirements (R4) and other test cases that give traceability to those requirements are deleted. Since there is no edge except R4 and T3, proceed to the next step.

Table 9 Algorithm H3 Iteration 1 - Coverage numbers of remaining test cases

Test case	Coverage count
T2	1
T3	1

Table 10 Algorithm H3 Iteration 1 - Coverage number of last remaining test case

Test case	Coverage count
T3	1

- **Iteration 1 (for T1):** The iteration ends as all requirements are covered.
- **Iteration 1 (for T1):** Selected Test Cases: T1, T2, T3. Selected Test Cases Count: 3
- **Iteration 1 (for T1):** Since $3 < \infty$ (GlobalCount), in the final state, there are 3 test cases (T1, T2 and T3) in F and GlobalCount is set to 3.
- **Iteration 2 (for T2):** T2 is selected.
- **Iteration 2 (for T2):** All the edges between T2 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T2 is added to the selected test cases.
- **Iteration 2 (for T2):** Edges between the covered requirements (R1 and R2) and other test cases that give traceability to those requirements are deleted.
- **Iteration 2 (for T2):** Using uncovered requirements (R3 and R4), the coverage numbers of test cases are determined and sorted in descending order (Table 11).
- **Iteration 2 (for T2):** The first element, T3, is selected.
- **Iteration 2 (for T2):** All the edges between T3 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T3 is added to the selected test cases.
- **Iteration 2 (for T2):** Edges between the covered requirements (R3 and R4) and other test cases that give traceability to those requirements are deleted.
- **Iteration 2 (for T2):** The iteration ends as all requirements are covered.
- **Iteration 2 (for T2):** Selected Test Cases: T2, T3. Selected Test Cases Count: 2
- **Iteration 2 (for T2):** Since $2 < 3$ (GlobalCount), in the final state, there are 2 test cases (T2 and T3) in F and GlobalCount is set to 3.
- **Iteration 3 (for T3):** T3 is selected.
- **Iteration 3 (for T3):** All the edges between T3 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T3 is added to the selected test cases.
- **Iteration 3 (for T3):** Edges between the covered requirements (R3 and R4) and other test cases that give traceability to those requirements are deleted.
- **Iteration 3 (for T3):** Using uncovered requirements (R1 and R2), the coverage numbers of test cases are determined and sorted in descending order (Table 12).
- **Iteration 3 (for T3):** The first element, T2, is selected.

Table 11 Algorithm H3 Iteration 2 - Sorted coverage numbers of remaining test cases

Test case	Coverage count
T3	2
T1	1

Table 12 Algorithm H3 Iteration 3 - Sorted coverage numbers of remaining test cases

Test case	Coverage count
T2	2
T1	1

- **Iteration 3 (for T3):** All the edges between T2 and all the requirements it covers are deleted. These requirements are added to the covered requirements. T2 is added to the selected test cases.
- **Iteration 3 (for T3):** Edges between the covered requirements (R1 and R2) and other test cases that give traceability to those requirements are deleted.
- **Iteration 3 (for T3):** The iteration ends as all requirements are covered.
- **Iteration 3 (for T3):** Selected Test Cases: T2, T3. Selected Test Cases Count: 2
- **Iteration 3 (for T3):** Since 2 = 2 (GlobalCount), there is no update on F.
- **Iteration 3 (for T3):** Finally, there are 2 test cases (T2 and T3) in F and GlobalCount is set to 2.
- **Iteration 3 (for T3):** The algorithm terminates as all elements in maxCoveringElements have been processed.
- **Iteration 3 (for T3):** Selected Test Cases: T2, T3.

After all algorithms are applied on the given example, the final state is summarized in Table 13.

4 Experimental results

Proposed algorithms, H1, H2 and H3, were compared with Genetic Algorithm (GA), Ant Colony Algorithm (ACO) and Particle Swarm Optimization (PSO). Algorithms were run for 1, 15, 30 and 60 minutes on the same data set and the results are shown in the Tables 14,15,16,17,18,19,20,21,22,23,24. The values indicated in the tables are in seconds. While all data sets are randomly generated, the last 5 data sets are generated to include solutions. In this way, the minimum number of test cases that could cover the requirements could be determined and the algorithmic values used in GA, ACO and PSO could be given in advance (Tables 20,21,22,23,24).

Let's analyze Table 17.

Table 13 Comparison of algorithms

Algorithms	Number of test cases required to cover all requirements	Test cases
H1	2	T2, T3
H2	3	T1,T2,T3
H3	2	T2,T3

- When algorithms are run with a 1 minute time limit;
  - It has been observed that H1 and H2 find the solution in less than 1 second and H3 in 60 seconds.
  - GA, ACO and PSO could not find the solution within the specified time.
  - H3 covered all requirements with 26 test cases, H2 with 27 test cases, H1 with 46 test cases.
  - GA was able to cover 977 requirements with 25 test cases in 60 seconds. ACO was able to cover 944 requirements with 25 test cases in 60 seconds. PSO was able to cover 957 requirements with 25 test cases in 60 seconds.
- When algorithms are run with a 15 minute time limit;
  - No improvement was detected in the H1 and H2 algorithms, as they reached the result in a short time.
  - Although the H3 algorithm did not show any improvement in terms of results, average time increased and reached 900 seconds.
  - GA could not find the solution within 900 seconds. However, it increased the number of requirements covered from 977 to 992.
  - ACO could not find the solution within 900 seconds. In addition, it reduced the number of requirements covered from 944 to 943.
  - PSO could not find the solution within 900 seconds. However, it increased the number of requirements covered from 957 to 962.
- When algorithms are run with a 30 minute time limit;
  - No improvement was detected in the H1 and H2 algorithms, as they reached the result in a short time.
  - Although the H3 algorithm did not show any improvement in terms of results, average time increased and reached 1800 seconds.
  - GA could not find the solution within 1800 seconds. However, it increased the number of requirements covered from 992 to 993.
  - ACO could not find the solution within 1800 seconds. However, it increased the number of requirements covered from 943 to 944.
  - PSO could not find the solution within 1800 seconds. However, it increased the number of requirements covered from 962 to 964.
- When algorithms are run with a 60 minute time limit;
  - No improvement was detected in the H1 and H2 algorithms, as they reached the result in a short time.
  - Although the H3 algorithm did not show any improvement in terms of results, average time increased and reached 3600 seconds.

Table 14 Comparison Table - 1

#Total test cases: 100 # Requirements in test case: 5 # Total requirements: 100 # Requirements to be covered:15		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	10	10	10	13	8	8
	Covered	15	15	15	15	15	15
	Uncovered	0	0	0	0	0	0
	Elapsed time	1	35	5	0	0	5
15	Final test cases	10	10	10	13	8	8
	Covered	15	15	15	15	15	15
	Uncovered	0	0	0	0	0	0
	Elapsed time	1	35	5	0	0	5
30	Final test cases	10	10	10	13	8	8
	Covered	15	15	15	15	15	15
	Uncovered	0	0	0	0	0	0
	Elapsed time	1	35	7	0	0	5
60	Final test cases	10	10	10	13	8	8
	Covered	15	15	15	15	15	15
	Uncovered	0	0	0	0	0	0
	Elapsed time	1	35	8	0	0	5

Table 15 Comparison Table - 2

#Total test cases: 1000 # Requirements in test case: 5 #Total requirements: 1000 # Requirements to be covered:25		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	20	20	20	25	20	20
	Covered	25	5	15	25	25	25
	Uncovered	0	20	10	0	0	0
	Elapsed time	24	60	60	0	0	60
15	Final test cases	20	20	20	25	20	20
	Covered	25	6	17	25	25	25
	Uncovered	0	19	8	0	0	0
	Elapsed time	347	900	900	0	0	900
30	Final test cases	20	20	20	25	20	20
	Covered	25	14	17	25	25	25
	Uncovered	0	11	8	0	0	0
	Elapsed time	265	1800	1800	0	0	1800
60	Final test cases	20	20	20	25	20	20
	Covered	25	15	19	25	25	25
	Uncovered	0	10	6	0	0	0
	Elapsed time	943	3600	3600	0	0	3600

Table 16 Comparison Table - 3

#Total test cases: 10000 # Requirements in test case: 25 # Total requirements: 10000 # Requirements to be covered:100		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	50	50	50	86	52	52
	Covered	63	19	28	100	100	100
	Uncovered	37	81	72	0	0	0
	Elapsed time	60	60	60	19	24	60
15	Final test cases	50	50	50	86	52	52
	Covered	93	19	32	100	100	100
	Uncovered	7	81	68	0	0	0
	Elapsed time	900	900	900	19	24	900
30	Final test cases	50	50	50	86	52	52
	Covered	94	19	32	100	100	100
	Uncovered	6	81	68	0	0	0
	Elapsed time	1800	1800	1800	19	24	1800
60	Final test cases	50	50	50	86	52	52
	Covered	96	19	37	100	100	100
	Uncovered	4	81	63	0	0	0
	Elapsed time	3600	3600	3600	19	24	3600

Table 17 Comparison Table - 4

#Total test cases: 1000 # Requirements in test case: 100 # Total requirements: 1000 # Requirements to be covered:1000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	25	25	25	46	27	26
	Covered	977	944	957	1000	1000	1000
	Uncovered	23	56	43	0	0	0
	Elapsed time	60	60	60	0	0	60
15	Final test cases	25	25	25	46	27	26
	Covered	992	943	962	1000	1000	1000
	Uncovered	8	57	38	0	0	0
	Elapsed time	900	900	900	0	0	900
30	Final test cases	25	25	25	46	27	26
	Covered	993	944	964	1000	1000	1000
	Uncovered	7	56	36	0	0	0
	Elapsed time	1800	1800	1800	0	0	1800
60	Final test cases	25	25	25	46	27	26
	Covered	992	945	967	1000	1000	1000
	Uncovered	8	55	33	0	0	0
	Elapsed time	3600	3600	3600	0	0	3600

Table 18 Comparison Table - 5

#Total test cases: 10000 # Requirements in test case: 100 # Total requirements: 10000 # Requirements to be covered:10000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	25	25	25	461	265	265
	Covered	2292	2257	2281	10000	10000	10000
	Uncovered	7708	7743	7719	0	0	0
	Elapsed time	60	60	60	54	49	60
15	Final test cases	25	25	25	461	265	265
	Covered	2392	2257	2287	10000	10000	10000
	Uncovered	7608	7743	7713	0	0	0
	Elapsed time	900	900	900	54	49	900
30	Final test cases	25	25	25	461	265	265
	Covered	2406	2257	2288	10000	10000	10000
	Uncovered	7594	7743	7712	0	0	0
	Elapsed time	1800	1800	1800	54	49	1800
60	Final test cases	25	25	25	461	265	265
	Covered	2409	2257	2293	10000	10000	10000
	Uncovered	7591	7743	7707	0	0	0
	Elapsed time	3600	3600	3600	54	49	3600

Table 19 Comparison Table - 6

#Total test cases: 1000 # Requirements in test case: 10 #Total requirements: 1000 # Requirements to be covered:1000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	50	50	50	235	156	155
	Covered	457	413	424	1000	1000	1000
	Uncovered	543	587	576	0	0	0
	Elapsed time	60	60	60	1	1	60
15	Final test cases	50	50	50	235	156	155
	Covered	491	412	431	1000	1000	1000
	Uncovered	509	588	569	0	0	0
	Elapsed time	900	900	900	1	1	900
30	Final test cases	50	20	50	235	156	155
	Covered	495	411	441	1000	1000	1000
	Uncovered	505	589	559	0	0	0
	Elapsed time	1800	1800	1800	1	1	1800
60	Final test cases	50	20	50	235	156	155
	Covered	496	414	447	1000	1000	1000
	Uncovered	504	586	553	0	0	0
	Elapsed time	3600	3600	3600	1	1	3600

Table 20 Comparison Table - 7

#Total test cases: 50 # Requirements in test case: 50 #Total requirements: 1000 # Requirements to be covered:1000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	20	20	20	20	26	22
	Covered	1000	744	883	1000	1000	1000
	Uncovered	0	256	117	0	0	0
	Elapsed time	19	60	60	0	0	60
15	Final test cases	20	20	20	20	26	22
	Covered	1000	784	954	1000	1000	1000
	Uncovered	0	216	46	0	0	0
	Elapsed time	26	900	900	0	0	900
30	Final test cases	20	20	20	20	26	22
	Covered	1000	797	941	1000	1000	1000
	Uncovered	0	203	59	0	0	0
	Elapsed time	18	1800	1800	0	0	1800
60	Final test cases	20	20	20	20	26	21
	Covered	1000	796	963	1000	1000	1000
	Uncovered	0	204	37	0	0	0
	Elapsed time	18	3600	3600	0	0	3600

Table 21 Comparison Table - 8

#Total test cases: 100 # Requirements in test case: 10 #Total requirements: 100 # Requirements to be covered:100		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	10	10	10	22	17	15
	Covered	92	75	86	100	100	100
	Uncovered	8	25	14	0	0	0
	Elapsed time	45	60	60	0	0	60
15	Final test cases	10	10	10	22	17	15
	Covered	91	76	88	100	100	100
	Uncovered	9	24	12	0	0	0
	Elapsed time	542	900	900	0	0	900
30	Final test cases	10	10	10	22	17	12
	Covered	97	77	91	100	100	100
	Uncovered	3	23	9	0	0	0
	Elapsed time	464	1800	1500	0	0	1800
60	Final test cases	10	10	10	22	17	11
	Covered	92	78	93	100	100	100
	Uncovered	8	22	7	0	0	0
	Elapsed time	2004	3600	2963	0	0	3600

Table 22 Comparison Table - 9

#Total test cases: 1000 # Requirements in test case: 40 #Total requirements: 1000 # Requirements to be covered:1000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	25	25	25	92	58	56
	Covered	736	661	683	1000	1000	1000
	Uncovered	264	339	317	0	0	0
	Elapsed time	60	60	60	0	0	60
15	Final test cases	25	25	25	92	58	56
	Covered	784	662	699	1000	1000	1000
	Uncovered	216	338	301	0	0	0
	Elapsed time	834	900	900	0	0	900
30	Final test cases	25	25	25	92	58	56
	Covered	759	663	695	1000	1000	1000
	Uncovered	241	337	305	0	0	0
	Elapsed time	1800	1800	1800	0	0	1800
60	Final test cases	25	25	25	92	58	56
	Covered	779	664	707	1000	1000	1000
	Uncovered	221	336	293	0	0	0
	Elapsed time	3285	3600	3600	0	0	3600

Table 23 Comparison Table - 10

#Total test cases: 10.000 # Requirements in test case: 50 # Total requirements: 10.000 # Requirements to be covered:10.000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	200	200	200	523	422	421
	Covered	6435	6403	6422	9283	9854	9856
	Uncovered	3565	3597	3578	717	146	144
	Elapsed time	60	60	60	60	57	60
15	Final test cases	200	200	200	768	462	461
	Covered	6625	6403	6429	10000	10000	10000
	Uncovered	3375	3597	3571	0	0	0
	Elapsed time	900	900	900	75	59	900
30	Final test cases	200	200	200	768	462	460
	Covered	6655	6403	6434	10000	10000	10000
	Uncovered	3345	3597	3566	0	0	0
	Elapsed time	1800	1800	1800	75	51	1800
60	Final test cases	200	200	200	768	462	460
	Covered	6690	6404	6441	10000	10000	10000
	Uncovered	3310	3596	3559	0	0	0
	Elapsed time	3600	3600	3600	75	63	3600

Table 24 Comparison Table - 11

#Total test cases: 100.000 # Requirements in test case: 1000 # Total requirements: 100.000 # Requirements to be covered:100.000		Algorithm					
		GA	ACO	PSO	H1	H2	H3
1	Final test cases	100	100	100	0	0	0
	Covered	63421	63616	63543	0	0	0
	Uncovered	36579	36384	36457	100000	100000	100000
	Elapsed time	60	60	60	60	60	60
15	Final test cases	100	100	100	0	0	0
	Covered	63809	63616	63678	0	0	0
	Uncovered	36191	36384	36322	100000	100000	100000
	Elapsed time	900	900	900	900	900	900
30	Final test cases	100	100	100	31	32	32
	Covered	63990	63616	63723	26464	28888	28888
	Uncovered	36010	36384	36277	73536	71112	71112
	Elapsed time	1800	1800	1800	1800	1800	1800
60	Final test cases	100	100	100	523	414	414
	Covered	64165	63616	63734	97141	100000	100000
	Uncovered	35835	36384	36266	2859	0	0
	Elapsed time	3600	3600	3600	3600	3436	3436

- GA could not find the solution within 3600 seconds. In addition, it reduced the number of requirements covered from 993 to 992.
- ACO could not find the solution within 3600 seconds. However, it increased the number of requirements covered from 944 to 945.
- PSO could not find the solution within 3600 seconds. However, it increased the number of requirements covered from 964 to 967 (Table 17).

When all the tables were examined, the following results were determined:

- Since algorithmic values in GA, ACO and PSO could not be given in advance, no result was found in some cases.
- GA, ACO and PSO produced worse results in some cases, although the time increased.
- H3 is the only algorithm that guarantees to improve the result with an increase in time.
- H3 produced mostly the same results as H2, despite the increase in time.
- H3 was able to solve the problem at the same time as H2. However, as H3 continues to iterations, its average duration has increased.
- Most of the time, there has been no improvement in the H3 algorithm although time has increased.
- Since iterations with large parameters take a long time, the average times have also increased. However, ACO was most affected by this situation.
- H1, H2 and H3 started to fail to produce results in a short time with big parameters.
- Since H1, H2 and H3 are deterministic algorithms, they produced the same result in each run.
- In cases where no solution is included in the data set, the H2 algorithm has come to the fore. When the solution was added, other algorithms were also able to produce good results.

5 Conclusion

In this research, information about the software life cycle was given and the test phase, which is an important step of this cycle, was examined. Examples of different testing methodologies used while testing the software are provided. The usage purposes of regression tests, one of these methodologies, are mentioned. Regression test methods, conditions controlled while performing these tests and difficulties that may occur during testing are stated. The purpose of use of test case prioritization, which is one of the methods that makes regression testing more effective, has been explained. The algorithms used within the scope of test case prioritization were examined with examples and 3 new algorithms were introduced using the dominating set method. These developed algorithms were compared with metaheuristic algorithms and performance analysis was performed using the same data set.

The newly developed algorithms were compared with metaheuristic algorithms and performance analyzes were made using the same data sets. As a result of the comparisons, the H2 algorithm showed the best performance in the first 6 data sets that did not contain a solution. Since these data sets do not contain solutions, the necessary parameters for GA, ACO and PSO could not be given. This has negatively affected the performance of the algorithms. In the last 5 datasets containing the solutions, the H2 algorithm has come to the fore, while the performance of metaheuristic algorithms has also increased. All of the developed algorithms were able to solve the problem in every data set. However, as the data set grew, they could not produce solutions in a short time. It has been observed that the short-time performance of metaheuristic algorithms is better in large data sets. As the dataset grew, the performance of ACO decreased more than other algorithms. While the H1 algorithm performed best in certain datasets, it lagged behind H2 in most datasets. Although the H3 algorithm produced mostly the same result as H2, it was determined that the average time was longer. In addition, it has been determined that the only algorithm with a guarantee of improving the result is H3.

Funding The authors have not disclosed any funding.

Data availability Enquiries about data availability should be directed to the authors.

Code Availability All the codes implemented during this study are available from the corresponding author on reasonable request.

Declarations

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

Ansari A, Khanb A, Khanc A, Mukadamd K (2016) Optimized regression test using test case prioritization. Procedia Comput Sci 79:152–160

Azizi M, Do H (2018) Graphite: A Greedy Graph-Based Technique for Regression Test Case Prioritization, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 15–18 October, 245–251. Memphis, TN, USA

Bagherzadeh M, Kahani N, Briand L (2021) Reinforcement learning for test case prioritization. IEEE Trans Software Eng. https://doi.org/10.1109/TSE.2021.3070549

Bian Y, Li Z, Zhao R, Gong D (2017) Epistasis based aco for regression test case prioritization. IEEE Trans Emerg Top Comput Intell 1(3):213–223

Chen J, Zhu L, Chen TY, Towey D, Kuo FC, Huang R, Guo Y (2018) Test case prioritization for object-oriented software: an adaptive random sequence approach based on clustering. J Syst Softw 135:107–125

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 163:110539

Elbaum S, Malishevsky AG, Rothermel G (2002) Test case prioritization: a family of empirical studies. IEEE Trans Softw Eng 28(2):159–182

Elbaum S, Rothermel G, Kanduri S, Malishevsky AG (2004) Selecting a cost-effective test case prioritization technique. Softw Qual J 12(3):185–210

Farooq F, Nadeem A (2017) A fault based approach to test case prioritization. In: International conference on Frontiers of information technology (FIT), 18–20 Dec, Islamabad, Pakistan, pp 52–57

Gokilavani N, Bharathi B (2021) Test case prioritization to examine software for fault detection using PCA extraction and K-means clustering With ranking. Soft Comput 25:5163–5172

Hla KHS, Choi Y, Park JS (2008) Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In: IEEE 8th international conference on computer and information technology workshops, 8–11 July, Sydney, Australia, pp 527–532

Huang R, Zhang Q, Towey D, Sun W, Chen J (2020) Regression test case prioritization by code combinations coverage. J Syst Softw 169:110712

Jeffrey D, Gupta N (2006) Test case prioritization using relevant slices. In: 30th annual international computer software and applications conference (COMPSAC'06), 17–21 Sept, Chicago, USA, pp 411–420

Jun W, Yan Z, Chen J (2011) Test case prioritization technique based on genetic algorithm. In: International conference on internet computing and information services, 17–18 Sept, Hong Kong, China, pp 173–175

Kaur A, Bhatt D (2011) Hybrid particle swarm optimization for regression testing. Int J Comput Sci Eng 3(5):1815–1824

Kavitha R, Kavitha VR, Suresh Kumar N (2010) Requirement based test case prioritization. In: International conference On communication control and computing technologies, 7–9 Oct, Ramanathapuram, India, pp 826–829

Kayes MI (2011) Test case prioritization For regression testing based on fault dependency. In: 3rd international conference on electronics computer technology, 8–10 Apr, Kanyakumari, India, pp 48–52

Kim J, Porter A (2002) A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th international conference on software engineering (ICSE 2002), 25–25 May, Orlando, USA, 119–129

Konsaard P, Ramingwong L (2015) Total coverage based regression test case prioritization using genetic algorithm. In: 12th international conference on electrical engineering/electronics, computer, telecommunications and information technology (ECTI-CON), 24–27 June, Hua Hin, Thailand, pp 1–6

Krishnamoorthi R, Sahaaya SA, Mary SA (2009) Regression test suite prioritization using genetic algorithms. Int J Hybrid Inf Technol 2(3):35–52

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

Liedlof M (2008) Finding a dominating set on bipartite graphs. Inf Process Lett 107(5):154–157

Lin C, Chen C, Tsai C, Kapfhammer GM (2013) History-based test case prioritization with software version awareness. In: 18th international conference on engineering of complex computer systems, 17–19 July. Singapore, Singapore, pp 171–172

Luo L (2001) Software Testing Techniques. Institute For Software Research International Carnegie Mellon University Pittsburgh, USA

Malik KI, Khalid H, Hassan S, Fatima K (2013) Software testing methodologies for finding errors. Res J Soc Sci Manag 3(7):7–14

Mohapatra SK, Prasad S (2013) Evolutionary search algorithms for test case prioritization. In: International conference on machine intelligence and research advancement, 21–23 Dec, Katra, India, pp 115–119

Murali R, Koyutürk M, Ananth G, Suresh J (2008) PHALANX: a graph-theoretic framework for test case prioritization. In: Proceedings of the ACM symposium on applied computing, 16–20 Mar, Fortaleza, Ceara, Brazil, pp 667–673

Navdeep Jain C, Gambhir A (2015) Cost effective regression testing. Cognition 4:8–10

Panwar D, Tomar P, Singh V (2018) Hybridization of Cuckoo-ACO algorithm for test case prioritization. J Stat Manag Syst 21(4):539–546

Rothermel G, Untch RH, Chu C, Harrold MJ (1999) Test case prioritization: an empirical study. In: Proceedings IEEE international conference on software maintenance, 30 Aug–3 Sept, Oxford, England, pp 179–188

Singh Y, Kaur A, Suri B (2010) Test case prioritization using ant colony optimization. Softw Eng Notes 35(4):1–7

Srivastava PR (2008) Test case prioritization. J Theor Appl Inf Technol 4:178–181

Wang Y, Zhao X, Ding X (2015) An effective test case prioritization method based on fault severity. In: 6th IEEE international conference on software engineering and service science (ICSESS), 23–25 Sept, Beijing, China, pp 737–741

Wong W, Horgan J, London S, Agrawal H (1997) A study of effective regression testing in practice. In: Proceedings of the eighth international symposium on software reliability engineering, pp 230–238