Journal of Universal Computer Science, vol. 27, no. 2 (2021), 170-189
submitted: 19/4/2020, accepted: 1/2/2021, appeared: 28/2/2021

K-Step Crossover Method based on Genetic Algorithm for Test Suite Prioritization in Regression Testing

P.K. Gupta
(Department of Computer Science and Engineering Jaypee University of Information Technology Waknaghat, Solan, HP, India 173 234
https://orcid.org/0000-0003-0416-7097, pkgupta@ieee.org)

Abstract: Software is an integration of numerous programming modules (e.g., functions, procedures, legacy system, reusable components, etc.) tested and combined to build the entire module. However, some undesired faults may occur due to a change in modules while performing validation and verification. Retesting of entire software is a costly affair in terms of money and time. Therefore, to avoid retesting of entire software, regression testing is performed. In regression testing, an earlier created test suite is used to retest the software system's modified module. Regression Testing works in three manners; minimizing test cases, selecting test cases, and prioritizing test cases. In this paper, a two-phase algorithm has been proposed that considers test case selection and test case prioritization technique for performing regression testing on several modules ranging from a smaller line of codes to huge line codes of procedural language. A textual based differencing algorithm has been implemented for test case selection. Program statements modified between two modules are used for textual differencing and utilized to identify test cases that affect modified program statements. In the next step, test case prioritization is implemented by applying the Genetic Algorithm for code/condition coverage. Genetic operators: Crossover and Mutation have been applied over the initial population (i.e. test cases), taking code/condition coverage as fitness criterion to provide a prioritized test suite. Prioritization algorithm can be applied over both original and reduced test suite depending upon the test suite's size or the need for accuracy. In the obtained results, the efficiency of the prioritization algorithms has been analyzed by the Average Percentage of Code Coverage (APCC) and Average Percentage of Code Coverage with cost (APCCc). A comparison of the proposed approach is also done with the previously proposed methods and it is observed that APCC & APCCc values achieve higher percentage values faster in the case of the prioritized test suite in contrast to the non-prioritized test suite.

Keywords: Test case minimization, Test case prioritization, Genetic Algorithm, Crossover, Mutation, Test Suite, Regression testing.
Categories: D.2, D.2.4, D.2.5
DOI: 10.3897/jucs.65241

1 Introduction

The rising growth of the software industry worldwide puts a lot of pressure on developers and testers to complete their assigned tasks on or before the deadline. Most of the software consists of several bugs that result in system failures and produce incorrect, inconsistent, and incomplete results. However, software testing techniques evolved are used to reduce various drawbacks and try to make software bug free. Still,

in particular scenarios of software development where the regular modification or reuse of data takes place, these bugs get introduced in software knowingly or unknowingly. If done with all test cases, system retesting cost in computation, manpower, and revenue to the developer. As described in [Yoo and Harman 2012], mostly regression testing is used to keep account of each modification in software and retesting of test cases that focus only on modifications in all subsequent testing phases. In [Askarunisa, Shanmugapriya and Ramaraj 2010] [Kaur and Goyal 2011] [Suman and Seema 2012] [Yoo and Harman 2012] regression test prioritization is performed over test cases and analyzed for the average percentage of code coverage. This paper presents the effect of regression testing over a system. It provides an effective solution with 2-Phase algorithms where Phase-1 is associated with the test case selection, and Phase-2 is connected with the prioritization of test cases. The proposed k-step crossover method reduces the time and cost of the testing process.

1.1 Significance of Research in Science

Typically, testing is the key and basic concept in any model to verify how much the model is worthy, at what extent it is accurate. Before implementing any model, it is mandatory to test that model thoroughly. For instance, biomedical researchers test the drug on rats or other species to check the efficacy of the medicine. Once all the tests are passed, it is implemented for human beings. Similarly, proposed genetic algorithm presents a novel method that brings the attention to software industry and provides the effective results on large scale complex software systems that includes web-based systems, mobile-based system, compiled systems, automation software systems, and management information system etc. The main motivation behind this work is derived from Charles Darwin's [Darwin and Wallace 1858] theory of natural selection, which talks about selecting a particular breed concerning its ability to survive. Darwin proposed that nature "selects" the traits that helped an animal survive, and the gene is further transferred into the upcoming generations. Inspired by this theory, the Genetic Algorithm, a search heuristic, was introduced. The fittest individuals or the most optimized test cases are selected as the genes to be passed on in the next generation through reproduction. The major focus of this work can be summarized as follows:

- Use of textual differencing implemented Longest Common Sub-sequence Algorithm for test case selections.
- K-step crossover method used in Genetic Algorithms for prioritization of test cases.
- Analysis of test case prioritization based on APCC and APCCc.

This paper is organized into various sections. Section 2 presents the previous studies related to using the genetic algorithm in regression testing to minimise and prioritise the test cases. Section 3 discusses the proposed methodology and consists of a detailed description of the proposed K-step crossover mechanism which is based on a genetic algorithm. The proposed algorithm performs its operations into two phases. Section 4 focuses on the obtained results and at the end of phase II when prioritization is achieved using APCC and APCCc to analyze the obtained prioritization results. This section also

includes the comparative analysis of the proposed method with the other method. Finally, section 5 concludes the work.

2 Semantic-based Retrieval using Metadata

This section summarizes the recent trends and works for minimizing and prioritizing the test cases using various techniques.

In [Rothermel and Harrold 2015], have discussed the test case selection algorithm that constructs the control flow graphs (CFGs) for both the input program and the modified version of the program used for the selection of test cases with the intent of fault detection. In the obtained results, they have claimed that the proposed technique reduces the cost of regression testing. In [Habtemariam and Mohapatra 2019], have classified test cases' prioritisation as one of the NP-hard class of problems. They have proposed a solution based on a genetic algorithm. The proposed algorithm shows a better average percentage of fault detection (APFD) in their obtained results. In [Bajaj and Sangwan 2019], have performed a systematic review of various test case prioritization techniques using genetic algorithms. From their survey, it is concluded that genetic algorithms have many advantages while working on the issue of test case prioritization. In [Mishra, Panda, Mishra and Acharya 2019], a genetic algorithm-based prioritization solution has been proposed. The proposed technique considers various prioritization factors like statement coverage, total mutant coverage, and total fault exposed. They have further measured the efficiency of the proposed algorithm by Average Percentage of Statement Coverage (APSC) metric. In [Rothermel, Untcn, Chu and Harrold 2001], they have described several techniques for prioritizing the test cases. Proposed techniques consider the coverage of code and fault detection ability for the generation of test cases. In their obtained results, it is found that the proposed technique improves the rate of fault detection and cost-benefits trade-offs. However, they have also stated that there is still a lot of room for further improvement in the proposed technique. As described in [Yadav and Dutta 2019b], have focused on minimizing the cost of regression testing and therefore they have proposed a prioritization algorithm based on K-mean clustering techniques. In their obtained results they have claimed the highest fault detections. In [Noemmer and Haas 2020], have focused on the test suite minimization during regression testing of large software projects. They have compared the four different algorithms, and it is found that there is a reduction of 69% of test cases with the proposed algorithms. However, on the dark side of the proposed algorithm, it takes more execution time and loss in fault detection. In [Agrawal et al. 2020], a safe regression test case selection method is based on the hybrid whale optimization algorithm. They have also compared the proposed approach with various other nature-based computing techniques like Bat Search, ACO- based approach, etc. In [Harikarthik, Palanisamy and Ramanathan 2019], have discussed a regression test case prioritization approach that generates and forms the cluster of test cases using the fuzzy c-means clustering technique. This technique makes the relevant and irrelevant cluster of test cases to maximize the probability of fault detection. In [Yadav and Dutta 2019a], have discussed an object-oriented test case selection and prioritization approach. They have used the dependency graphs to find the program's changes, and

then select test cases performed using object-oriented models. In their results, they have achieved the high value of the average percentage of fault detection. In [Vokolos and Frankl 1998], have focused on implementing textual differencing and implemented selective regression testing technique that compares the old and new version of source code files. The obtained results found that textual differencing is very fast and reduces a significant number of test cases.

3 Proposed Methodology

As discussed in the literature survey, the test case prioritization and selections are two major challenges in regression testing. In this section, a K-step crossover methodology has been proposed, which is based on a genetic algorithm. The proposed methodology is divided into two phases and represented in the Algorithm 1. According to proposed methodology, phase I performs regression test selection in which two source codes under test are converted into their canonical forms. Then a difference between two source codes is obtained.

Further, to determine the coverage of modified statements and their execution trace a decision to decision (DD) path graph has been constructed. In phase II, test case prioritization has been performed in which a pool of genetic chromosomes has been created that consist of the randomly selected test cases. A pair of randomly selected chromosomes that fulfils the fitness criterion is further processed with genetic operators like crossover and mutation to obtain a prioritized chromosome. The fitness criterion selected for the algorithm is code/condition coverage which is managed through a DD path graph. This chromosome is used to obtain the prioritized test suite.

Algorithm 1: Two phase methodology for regression test selection and test case prioritization.

Input: Original Source Code, Modified Source Code, and Test Cases.
Output: Prioritized Test Cases (Reduced).
BEGIN PHASE I;
1. K-form ← source codes into canonical form;
2. Differencer ← modified statements in source code
3. Generate Test Execution Trace based on DD-path graph;
4. Select only test cases that assess modified statements;

END;
BEGIN PHASE II;
5. Generate initial population of *n* chromosomes;
6. Initialize chromosomes with *m* Test cases;
7. Set fitness function criterion for coverage of code;
8. Select the best 2 chromosomes based on fitness function;

Do Crossover for selected Chromosome(s)
for k ← 1 to 3
9. Interchange alternate k test cases between 2 chromosomes;

```
            if (all conditions are covered) then
10.             Break;
            endif
        end
    IF Crossover Fails
    11. GO TO 9
        Do Mutation
    12. Remove Duplicate Test Cases;
    13. Minimize Test Cases in the chromosome (all conditions are covered);
    14. Obtained chromosome is Prioritized Test Cases;
END;
```

3.1 Phase I: Reduction of Test Cases

a) Both original and modified source codes are presented in Appendix A in listing 1(a) and listing 1(b), respectively, has been considered an input to K Form function.

K Form Function - removes all comments and non-executable statements from there except executable statements in the codes.

b) The canonical form of original source code and modified source codes are presented in Appendix A in Listing 2(a) and Listing 2(b) respectively, are textually compared based on the longest common sub-sequence and results out the modified statement in the source code. The modified statements in the source code are found using the Longest Common Sub-sequence based textual differencing.

Canonical form – refers to the most straightforward representation of the considered source code. As shown in Appendix A, Listing 2(a) and Listing 2(b) represents the canonical forms of the codes as presented in listing 1(a) and listing 1(b), respectively, with a smaller number of lines in the code.

Differencer – it is a function that performs the comparison and finds the three types of modifications known as change, addition and deletion in the codes.

c) Basic Test Block Identifier is responsible for the generation of Test Execution Trace. This test execution trace activity is performed with various independent paths generated from the DD path graph as constructed for the code presented in Listing 2(b) and shown in Table 3. In other words, the DD path graph is a modified subset of a control flow graph, constructed by combining all non-decisive intermediate nodes into a single node(s). Test cases from the test case suite always follow one of these independent paths. Therefore, the knowledge of test cases following a particular path is stored as a test execution trace. Test Case Selection is the final step of Phase I in which modified statements obtained from the differencer function are used to determine modified edges in the DD-path graph. Finally, these test cases from the test execution trace are selected that consist of modified edges.

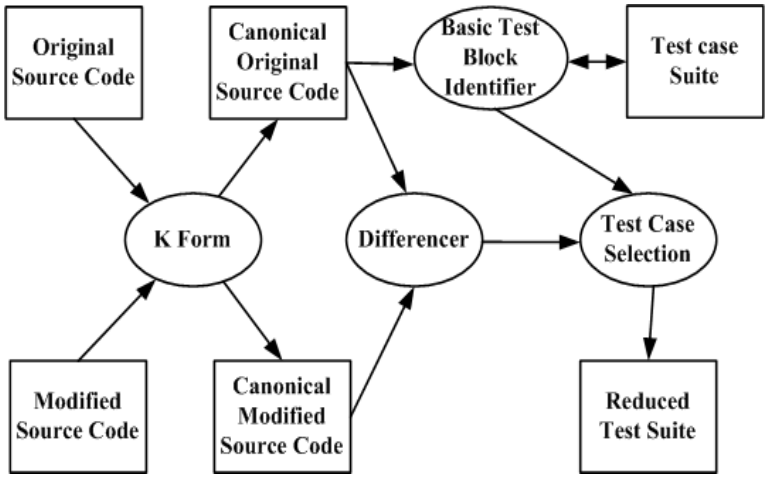


Figure 1: Reduction of test cases

d) The methodology for the reduction of test cases is shown in Fig. 1. Here, both original and modified source codes and their canonical forms and differences are shown in Listing 1 and Listing 2, respectively. K Form function removes all the comments and header files from the source code, and differencer provides the modified statements along with their line numbers, as shown in Table 1.

Original Canonical form [see Listing 2(a) in Appendix A]	**Modified Canonical form [see Listing 2(b) in Appendix A]**
10. cout << "Young whippersnapper," <<name<<"! \n";	10. cout << "You young Citizen " <<name<<"! \n"; 11. cout <<"You can Vote !!";
18. cout << "Existing" <<age<<"," <<name<<"? \n";	19. cout << "Really" <<age<<"," <<name<<"? \n";

Table 1: Differencer provides modified program segments along with their line numbers

e) Table 2 presents the test case structure for the code segment, as shown in Listing 2(a) and Listing 2(b), respectively. The designed test suite consists of 20 such test cases that have been used. Basic Test Block Identifier maps these 20 test cases on DD path graph of the source code as shown in Fig. 2, and the results of a test execution trace along the various independent paths are shown in Table 4. Here, various independent paths also consist of the cost of condition coverage. Test case selection based on textual differencing considers the differences in code and selects only those test cases from test execution traces consisting of modified edges. A reduced test execution trace is shown in Table 5.

ID	Input	Output
I1	Name=XYZ, Age=15	Young whippersnapper, XYZ!
I2	Name=ABC, Age=35	ABC, you're still in your prime!
I3	Name=XYZ, Age=55	You're over the hill, XYZ!
I4	Name=ABC, Age=75	I bow to your wisdom, ABC!
I5	Name=XYZ, Age=95	Existing 95, XYZ?
I6	Name=6782, Age=-5	Young whippersnapper, 6782!

Table 2: Example of Test Cases

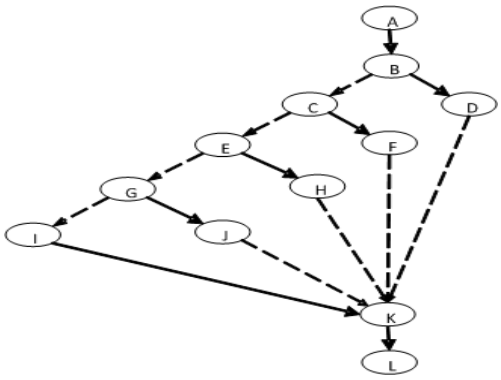


Figure 2: DD path graph of modified canonical program code

Line Number[see listing 2(b)]	Type of Node	Corresponding DD Path Node
1-8	Sequential	A
9	Decision	B
10-11	Sequential	D
12	Decision	C
13	Sequential	F
14	Decision	E
15	Sequential	H
16	Decision	G
17	Sequential	J
18-19	Decision and Sequential	I
20	Sequential	K
21	Sequential	L

Table 3: Line numbers from the Canonical Modified program source code corresponding to respective DD Path Node.

Test	Independent Path Covered	Cost
T1	A-B-D-K-L	1
T2	A-B-C-F-K-L	1
T3	A-B-C-E-H-K-L	1
T4	A-B-C-E-G-J-K-L	1
T5	A-B-C-E-G-I-K-L	1
T6	A-B-C-E-G-I-K-L	1
T7	A-B-C-E-H-K-L	1
T8	A-B-C-F-K-L	1
T9	A-B-D-K-L	1
T10	A-B-C-E-G-J-K-L	1
T11	A-B-C-E-H-K-L	1
T12	A-B-C-E-G-J-K-L	1
T13	A-B-C-E-G-I-K-L	1
T14	A-B-C-E-G-I-K-L	1
T15	A-B-C-E-H-K-L	1
T16	A-B-C-F-K-L	1
T17	A-B-D-K-L	1
T18	A-B-C-E-G-J-K-L	1
T19	A-B-D-K-L	1
T20	A-B-C-E-G-J-K-L	1

Table 4: Test Execution Trace

Test	Independent Path Covered
T5	A-B-C-E-G-I-K-L
T6	A-B-C-E-G-I-K-L
T9	A-B-D-K-L
T13	A-B-C-E-G-I-K-L
T14	A-B-C-E-G-I-K-L
T17	A-B-D-K-L
T19	A-B-D-K-L

Table 5: Reduced Tests Execution Trace

3.2 Phase I: Reduction of Test Cases

In this phase, one of the structural testing techniques known as path testing has been implemented for prioritization based on total code coverage. Path testing identifies the set of independent test paths with the help of a control flow graph. To achieve total code coverage, all independent paths need to be exercised at least once during path testing. Further, the use of genetic algorithm provides the better optimal solution for

the desired population. The selected fitness criterion is the total code coverage for generation and initialization of test cases. Genetic operations known as crossover and mutation are used to obtain the prioritized test case. Here, crossover recombines the two individuals, and mutation randomly swaps them and removes the redundant test cases. A detailed model of test case prioritization is shown in Fig 3. In the following section, various steps, as implemented by the proposed model, have been discussed.

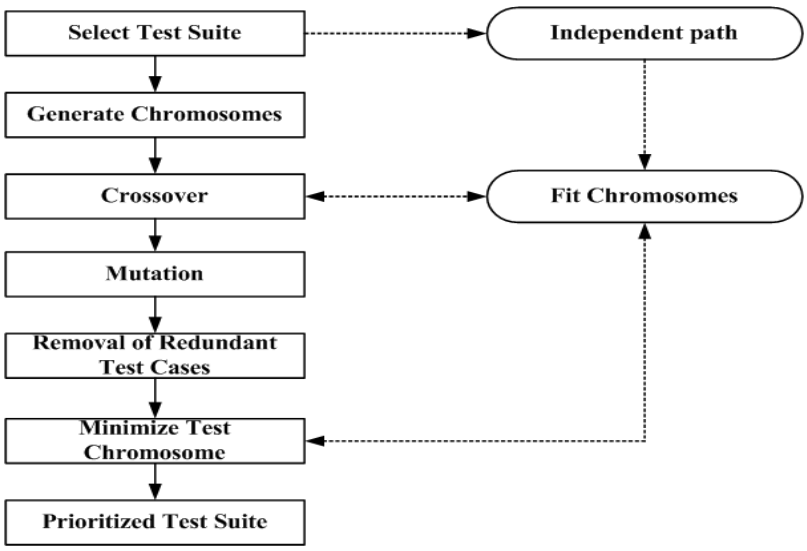


Figure 3: Model for Test Case Prioritization

a) *Select test suite:* The fitness Criterion selected for the algorithm is the total code coverage, and all the unique independent paths have been identified for the example as mentioned above. Every fit chromosome must cover all independent paths to ensure test cases and further implement genetic operation on them. Since the proposed algorithm applies prioritization over reduced test suite but for more clarity of concept and explanation, the example's original test suite has been used. It is because, the reduced test suite is too small to be used for demonstrating the detailed process. However, it does not affect the quality of the obtained results.

b) *Generation of chromosomes:* is the first ad important step of prioritization. It is because all the genetic operations are applied to the generated chromosomes only. Chromosomes are the combination of numerous test cases and two conditions must be followed while generating chromosomes: i) the number of chromosomes is at-least equal to the number of test cases, and ii) the number of test cases in a chromosome is at least equal to its cyclomatic complexity of code for which prioritization is being done. Chromosomes generated are a randomized selection of test cases.

Chromosomes	**Initialization of Chromosomes**									
C1	T20	T14	T18	T15	T2	T9	T3	T6	T10	T11
C2	T8	T13	T10	T17	T9	T16	T7	T15	T14	T3
C3	T19	T11	T10	T13	T8	T1	T4	T15	T9	T6
C4	T12	T8	T13	T18	T2	T3	T15	T6	T11	T5
C5	T11	T1	T17	T3	T8	T18	T7	T12	T4	T13
C6	T2	T19	T17	T18	T7	T15	T5	T14	T8	T10
C7	T13	T1	T17	T6	T11	T15	T12	T10	T16	T19
C8	T20	T7	T4	T11	T14	T15	T2	T17	T8	T16
C9	T19	T18	T10	T15	T14	T11	T3	T12	T16	T9
C10	T1	T4	T12	T19	T18	T6	T16	T17	T10	T5
C11	T9	T7	T19	T15	T16	T5	T17	T13	T2	T8
C12	T20	T3	T4	T12	T15	T7	T14	T2	T8	T17
C13	T13	T7	T18	T8	T1	T19	T12	T2	T11	T15
C14	T12	T5	T6	T16	T17	T15	T9	T8	T10	T4
C15	T5	T2	T14	T15	T19	T6	T12	T10	T4	T13
C16	T11	T8	T14	T17	T15	T12	T9	T5	T3	T7
C17	T7	T19	T14	T3	T16	T6	T20	T13	T17	T9
C18	T7	T15	T6	T1	T2	T11	T9	T8	T10	T17
C19	T4	T18	T3	T14	T16	T11	T17	T13	T12	T20
C20	T5	T8	T17	T14	T1	T20	T3	T6	T15	T7

Table 6: Chromosome Population

c) *Genetic operators:* are applied on two fit chromosomes randomly selected from the population, as provided in Table 6. Chromosomes fulfilling fitness criterion, i.e. total code coverage, are supposed to cover all the DD path graphs' independent paths. Here, one condition that must be followed is that both chromosomes should have the same length. The following genetic operators are applied to these chromosomes:

Crossover: is a process of yielding new pair of chromosomes. Initially, operator Crossover is applied over two fit chromosomes i.e. C2 and C1 and provides 1-step Crossed Chromosomes, as shown in Fig. 4, (a), 2-step Crossed Chromosomes shown in Fig. 4(b), and 3-step Crossed Chromosomes shown in Fig. 4(c). Stepping only occurs in crossover when both crossed

chromosomes fail total code coverage (fitness criterion). Fig. 4 represents an implementation of a k-step crossover method on chromosomes that interchanges alternate 'k' test cases between the selected chromosome pair.

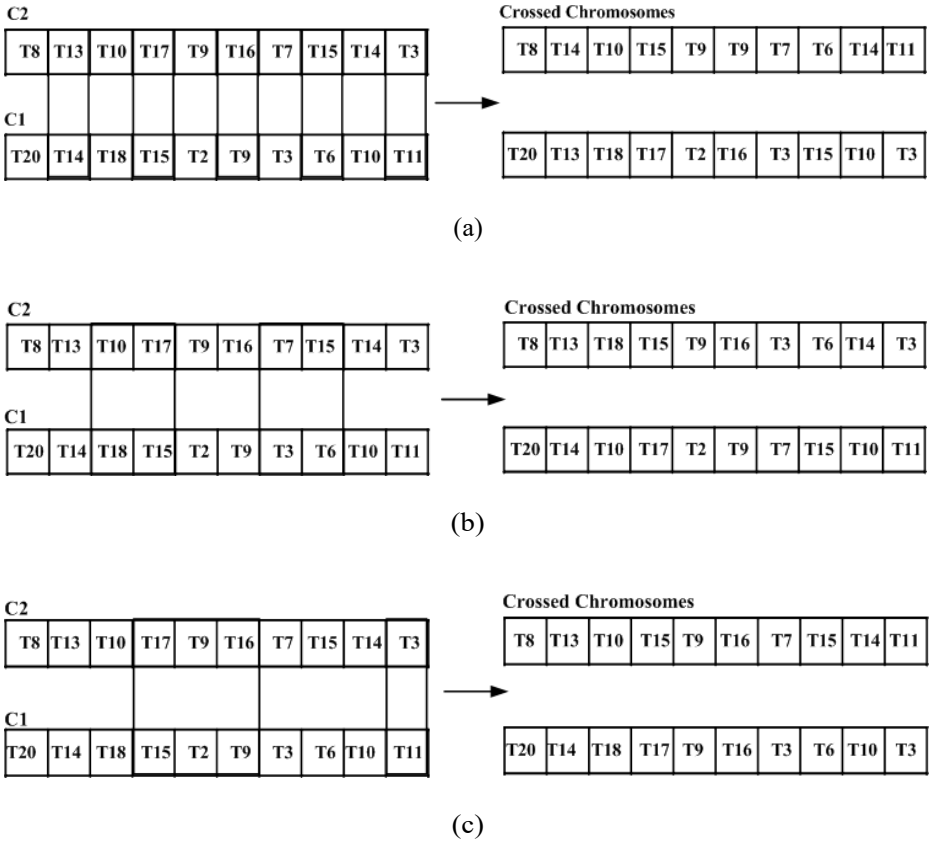
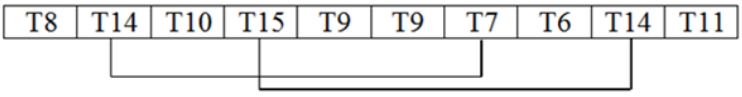


Figure 4: K-step Crossovers (a) 1-step crossover, (b) 2-step crossover & (c) 3-step crossover

Mutation: alters an individual in the population by randomly replacing the part of chromosome. Crossed chromosomes obtained may have both fit chromosomes or only one chromosome is fit. If both the chromosomes are fit, one chromosome is selected for mutation from either of the chromosomes. In case, if only one chromosome is fit then the fit chromosome is selected for mutation. The mutation is performed by randomly interchanging of a test case in the chromosome. Mutated Chromosome is after that sent to remove redundant test cases from it. As in the current example, crossed chromosomes obtained are both fit chromosomes. For mutation first, one is selected, and two test cases are randomly interchanged as shown in Fig. 5(a). After those two redundant test cases i.e. T14 and T9 are removed from the chromosome as shown in Fig. 5(b).



(a)

(b)

Figure 5: Mutated Chromosome (a) Interchanging of test cases (b) removal of redundant test cases.

d) *Removal of redundant test cases:* mutated chromosomes are free from redundant test cases. However, there is a possibility that the chromosome has still more than enough test cases required to cover all conditions. Thus, the minimization of the chromosome is an essential step to get a minimum number of test cases covering all conditions. A prioritized chromosome is the minimized mutated chromosome.

e) *Minimization of chromosome:* for minimization purposes greedy approach has been used. In the first iteration, arbitrarily one test case from the chromosome is selected as a seed chromosome and is merged with all other test cases in the chromosome to form chromosomes. After the end of the iteration, the seed chromosome is the chromosome with the highest conditions covered with the minimum number of test cases, are used.

f) *Prioritized chromosome:* resulted after the minimization of the mutated chromosome obtained in the previous step will speed up the testing of modified statements. A prioritized test suite as shown in Fig. 6 will contain test cases of prioritized chromosome first and the rest will be the remaining test cases of the used test suite.

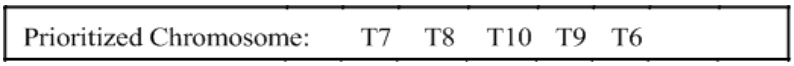


Figure 6: Prioritized Test Suite

4 Result Analysis

In this section, a detailed analysis of obtained results has been done for the proposed test suite prioritization algorithm by using various performance evaluation metrics. As per the discussed scenario, there is a total of 20 test cases in the test suite, and only program statements at line numbers 10 & 18 have been modified. At the end of phase

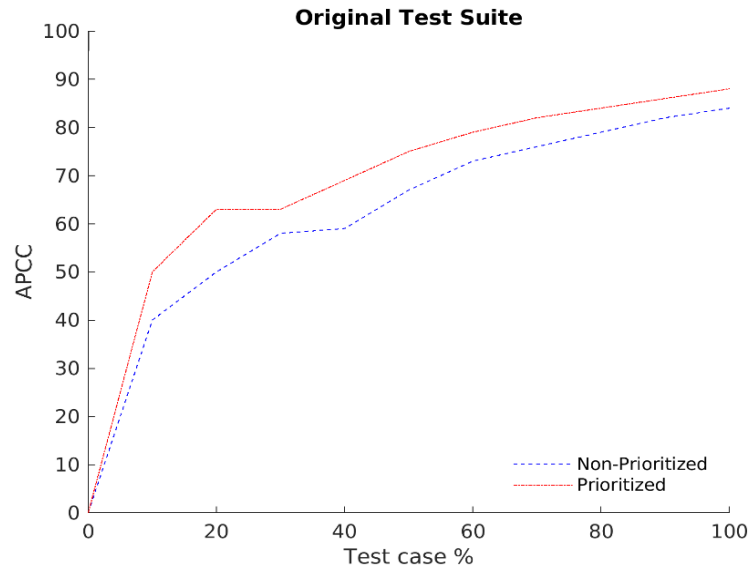
I, test cases covering only line numbers 10 & 18 have been selected which provide intermediate results in the form of reduced test cases counting to 7, in the test suite. Here, a significant drop of 65% test cases has been observed. Similarly, in the case of edge covering the total number of edges covered by the test suite are 118 and the total number of edges covered by the reduced test cases is 40, thus cutting by 66%. In phase II, several intermediate values and results have been obtained, ordered as Test Cases along Unique Paths, Unique Edges, Unique Nodes, Chromosomes, Fit Chromosomes, Crossed Chromosomes, Mutated Chromosome, and Prioritized Chromosome that provides the intermediate results for used Genetic Algorithm. Prioritized Chromosomes obtained contain a minimized number of test cases from Mutated Chromosome, supported by Total Condition Coverage. When prioritisation is achieved at the end of phase II, Average Percentage Condition Coverage (APCC) and Average Percentage Condition Coverage with cost (APCCc) have been used to analyze the obtained prioritization results. APCC measures the rate at which a prioritized test suite covers the conditions. In contrast, APCCc measures the number of test cases covering particular conditions and vice versa, which is used as cost. The APCC for test suite T' and APCCc of The weighted average percentage condition coverage during the execution of test suite T' is given by the following equations:

$$APCC = 1 - \frac{(TF_1 + TF_2 + ... + TF_m)}{nm} + \frac{1}{2n} \qquad (1)$$

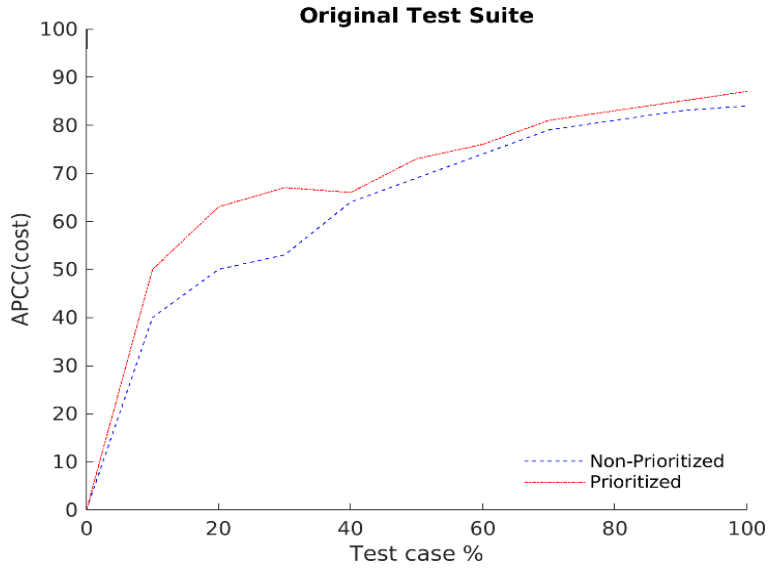
$$APCC_c = \frac{\sum_{i=1}^{n}(c_i \times (\sum_{i=TF_i}^{n} t_i - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} c_i} \qquad (2)$$

Where, $T \in$ test suite, $n \in$ number of test cases, $m \in$ set of covered conditions, ¸$c \in$ reduced test suite, $TF_i \in$ first test case in ordering $T'$of T which covers condition i in the test suite, $t_i \in$ the number of conditions covered by the i^{th} test case, $c_m \in$ cost of i^{th} test cases covering m conditions.

In the results, the value of APCC and APCCc obtained for non-prioritized test suite are 83.50% and 84.25% respectively [see Fig. 7(a) and Fig. 7(b)]. Similarly, the value of APCC and APCCc obtained for the prioritized test suite is 87.50% and 87.25% respectively. In a similar approach, results obtained for APCC and APCCc for non-prioritized reduced test suite are 85.71% and 86.73% respectively [see Fig. 7(a) and Fig. 7(b)]. Also, results obtained for APCC and APCCc for prioritized reduced test suite 91.62% and 92.38% respectively [see Fig. 8(a) and Fig. 8(b)].

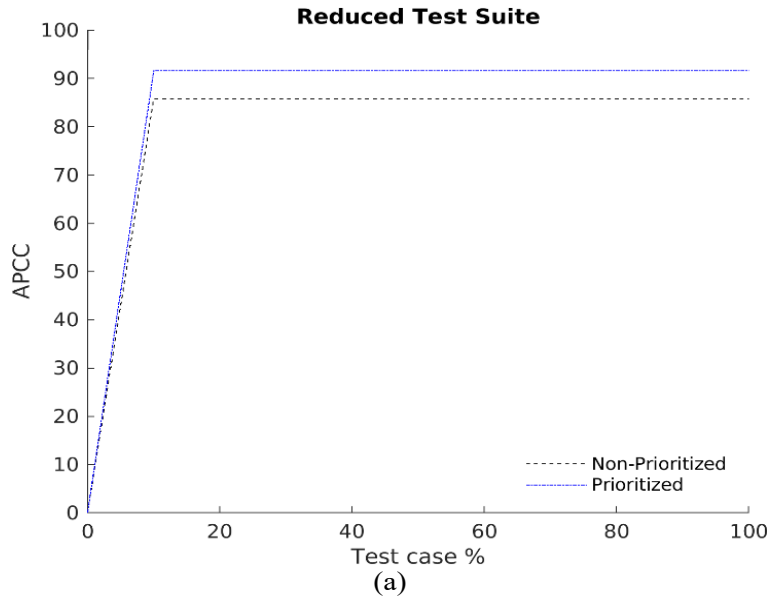


(a)

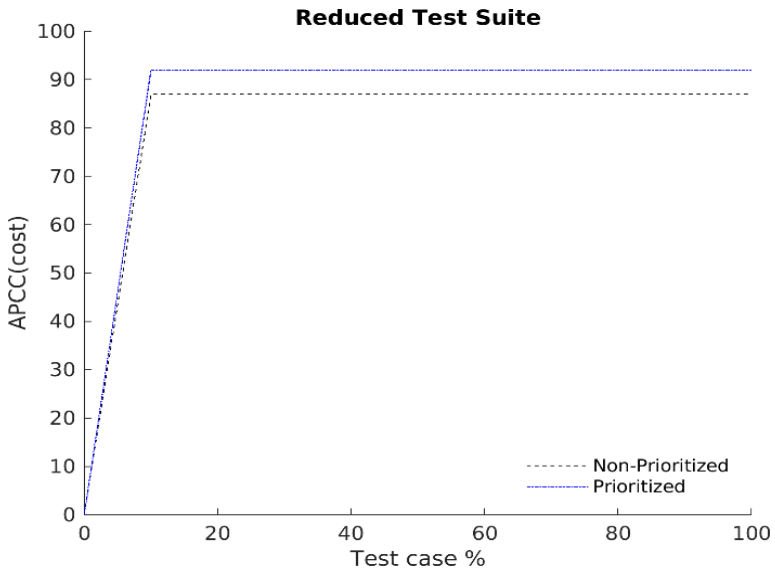


(b)

Figure 7: Original Test Suite a) APCC b) APCCc



(a)



(b)

Figure 8: Reduced Test Suite a) APCC b) APCCc

The obtained results have been compared with other approaches like Particle Swarm Optimization and Genetic algorithm with GA order as shown in Table 7. From the

obtained results, APCC & APCCc values achieve higher percentage values faster in the case of the prioritized test suite in contrast to the non-prioritized test suite. This ensures that the reduced test suite obtained from the proposed algorithm will provide complete and effective testability analysis of the provided source code.

Approach	**Test Suite**	**APCC %**
Particle Swarm Optimization [Kaur and Bhatt 2011]	Proposed Order	87.80
Genetic Algorithm [Kaur and Goyal 2011]	GA Order	88.30
Proposed Approach	Non-Prioritized Test Suite	83.50
	Prioritized Test Suite	87.50
	Non-Prioritized Reduced Test Suite	85.71
	Prioritized Reduced Test Suite	91.62

Table 7: Comparison with other test case prioritization approaches.

5 Conclusion

The proposed work discusses the implementation of two regression testing techniques designed to achieve better testability for any software system. Here, “any software systems” addresses mobile-based systems, web-based systems, and compiled systems as the objective of regression testing is to reduce the efforts and overall cost of the testing process. One can test the new version of the system(s) with reduced effective test cases. As a part of test case reduction, the textual differencing based test case selection method has been used. A large test case suite for entire software is reduced to a subset of test cases concerning only the software module's modifications. Genetic Algorithm, an evolutionary strategy is further used for test case prioritization which uses code/condition coverage as a fitness criterion. The obtained results for APCC and APCCc provide a higher percentage of test case prioritization with fewer test cases required to maximize code/condition coverage, which ensures that the reduced test suite

obtained by the proposed algorithm will provide complete and effective testability analysis of the provided source code. In future work, the proposed work can be extended to consider interprocedural control flow. Test case selection by using path analysis will provide an upper hand compared to textual differencing. Similarly, prioritization based on fault detection will also provide better testability than code/condition coverage.

Appendix

A) Program Listings

This appendix presents the detailed listing of programs used for developing a model for test case selection, and test case prioritization. Original and Modified source codes have been presented in these listings. The modified statements in the source code are found using Longest Common Sub-sequence based Textual differencing.

```
1. // An interactive example
2. // for Hands-on C++
3. #include <iostream.h>;
4.
5. main()
6. {
7. Sequential Statement 1;
8. Sequential Statement 2;
9. Sequential Statement 3;
10. Sequential Statement 4;
11. Sequential Statement 5;
12. Sequential Statement 6;
13. if (age > 21) {
14.     cout << "Young whippersnapper," <<name<<"! \n";
}
15. else if (age > 42) {
16.     Sequential Statement 7; }
17. else if ( age > 62) {
18.     Sequential Statement 8; }
19. else if (age > 82) {
20.     Sequential Statement 9; }
21. else {
22.     cout << "Existing" <<age<<"," <<name<<"? \n"; }
23. return 0;
24. }
```

(a)

```
1. // An interactive example
2. // for Hands-on C++
3. #include <iostream.h>;
4.
5. main()
6. {
7. Sequential Statement 1;
8. Sequential Statement 2;
9. Sequential Statement 3;
10. Sequential Statement 4;
11. Sequential Statement 5;
12. Sequential Statement 6;
13. if (age > 21) {
14.     cout << "You young Citizen," <<name<<"! \n";
15.     cout <<"You can Vote !!"; }
16. else if (age > 42 ) {
17.     Sequential Statement 7; }
18. else if (age > 62) {
19.     Sequential Statement 8; }
20. else if (age > 82 ) {
21.     Sequential Statement 9; }
22. else {
23.     cout << "Really" <<age<<"," <<name<<"? \n"; }
24. return 0;
25. }
```

(b)

Listing 1: General program source code a) Original b) Modified

```
1. main()
2. {
3. Sequential Statement 1;
4. Sequential Statement 2;
5. Sequential Statement 3;
6. Sequential Statement 4;
7. Sequential Statement 5;
8. Sequential Statement 6;
9. if (age > 21) {
10.     cout << "Young whippersnapper," <<name<<"!
\n"; }
11. else if (age > 42) {
12.     Sequential Statement 7; }
13. else if (age > 62) {
14.     Sequential Statement 8; }
15. else if (age > 82) {
16.     Sequential Statement 9; }
17. else {
18.     cout << "Existing" <<age<<"," <<name<<"? \n"; }
19. return 0;
20. }
```

(a)

```
1. main()
2. {
3. Sequential Statement 1;
4. Sequential Statement 2;
5. Sequential Statement 3;
6. Sequential Statement 4;
7. Sequential Statement 5;
8. Sequential Statement 6;
9. if (age > 21) {
10.     cout << "You young Citizen " <<name<<"! \n";
11.     cout <<"You can Vote !!"; }
12. else if (age > 42 ) {
13.     Sequential Statement 7; }
14. else if (age > 62 ) {
15.     Sequential Statement 8; }
16. else if ( age > 82 ) {
17.     Sequential Statement 9; }
18. else {
19.     cout << "Really" <<age<<"," <<name<<"? \n"; }
20. return 0;
21. }
```

(b)

Listing 2: Canonical forms of program source code a) Original b) Modified

References

[Agrawal et al., 2020] Agrawal, A. P., Choudhary, A., Kaur, A.: "An effective regression test case selection using hybrid whale optimization algorithm"; International Journal of Distributed Systems and Technologies, 11, 1 (2020), 53–67, https://doi.org/10.4018/IJDST.2020010105

[Askarunisa, Shanmugapriya and Ramaraj, 2010] Askarunisa, M. A., Shanmugapriya, M. L., Ramaraj, D. N.: "Cost and Coverage Metrics for Measuring the Effectiveness of Test Case Prioritization Techniques"; INFOCOMP Journal of Computer Science, 9, 1 (2010), 43-52, http://infocomp.dcc.ufla.br/index.php/infocomp/article/view/289

[Bajaj and Sangwan, 2019] Bajaj, A., Sangwan, O. P.: "A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms"; IEEE Access, 7 (2019), 126355–126375, https://doi.org/10.1109/ACCESS.2019.2938260

[Darwin and Wallace, 1858] Darwin, C., Wallace, A.: "On the Tendency of Species to form Varieties; and on the Perpetuation of Varieties and Species by Natural Means of Selection"; Zoological Journal of the Linnean Society, 3, 9 (1858), 45-62, https://doi.org/10.1111/j.1096-3642.1858.tb02500.x

[Habtemariam and Mohapatra, 2019] Habtemariam, G. M., Mohapatra, S. K.: "A Genetic Algorithm-Based Approach for Test Case Prioritization"; Communications in Computer and Information Science (Vol. 1026). Springer International Publishing (2019), https://doi.org/10.1007/978-3-030-26630-1_3

[Harikarthik, Palanisamy and Ramanathan, 2019] Harikarthik, S. K., Palanisamy, V., Ramanathan, P.: "Optimal test suite selection in regression testing with testcase prioritization using modified Ann and Whale optimization algorithm"; Cluster Computing, 22, 5(2019), 11425–11434. https://doi.org/10.1007/s10586-017-1401-7

[Kaur and Bhatt, 2011] Kaur, A., Bhatt, D.: "Particle Swarm Optimization with CrossOver Operator for Prioritization in Regression Testing"; International Journal of Computer Applications, 27, 10 (2011), 27-34, https://doi.org/10.5120/3336-4589

[Kaur and Goyal, 2011] Kaur, A., Goyal, S.: "A genetic algorithm for regression test case prioritization using code coverage"; International Journal on Computer Science and Engineering (2011). 3, 5 (2011), 1839-1847, http://www.enggjournals.com/ijcse/abstract.html?file=11-03-05-144

[Mishra, Panda, Mishra and Acharya, 2019] Mishra, D. B., Panda, N., Mishra, R., Acharya, A. A.: "Total fault exposing potential based test case prioritization using genetic algorithm"; International Journal of Information Technology, 11, 4 (2019), 633–637, https://doi.org/10.1007/s41870-018-0117-0

[Noemmer and Haas, 2020] Noemmer, R., Haas, R.: "An Evaluation of Test Suite Minimization Techniques"; Lecture Notes in Business Information Processing, 371 LNBIP (2020), 51–66, https://doi.org/10.1007/978-3-030-35510-4_4

[Rothermel and Harrold, 2015] Rothermel, G., Harrold, M. J.: "A safe, efficient regression test selection technique"; ACM Transactions on Software Engineering and Methodology (TOSEM), 6, 2 (2015), 173–210, https://doi.org/10.1145/248233.248262

[Rothermel, Untcn, Chu and Harrold, 2001] Rothermel, G., Untcn, R. H., Chu, C., Harrold, M. J.: "Prioritizing test cases for regression testing"; IEEE Transactions on Software Engineering, 27, 10 (2001), 929–948, https://doi.org/10.1109/32.962562

[Suman and Seema, 2012] Suman, Seema: "A Genetic Algorithm for Regression Test Sequence Optimization"; International Journal of Advanced Research in Computer and Communication Engineering (2012), 1, 7(2012), 478-481, https://ijarcce.com/wp-content/uploads/2012/03/A-Genetic-Algorithm-for-Regression-Test-Sequence.pdf

[Vokolos and Frankl, 1998] Vokolos, F. I., Frankl, P. G.: "Empirical evaluation of the textual differencing regression testing technique"; In Conference on Software Maintenance (1998), https://doi.org/10.1109/icsm.1998.738488

[Yadav and Dutta, 2019a] Yadav, D. K., Dutta, S.: "Regression test case selection and prioritization for object oriented software"; Microsystem Technologies, 26, 5 (2019a), 1463–1477. https://doi.org/10.1007/s00542-019-04679-7

[Yadav and Dutta, 2019b] Yadav, D. K., Dutta, S. K.: "Test case prioritization using clustering approach for object oriented software"; International Journal of Information System Modeling and Design, 10, 3 (2019b), 92–109, https://doi.org/10.4018/IJISMD.2019070106

[Yoo and Harman, 2012] Yoo, S., Harman, M.: "Regression testing minimization, selection and prioritization: A survey"; Software Testing Verification and Reliability (2012), 22, 2 (2012), 67-120, https://doi.org/10.1002/stv.430