Acceptance testing based test case prioritization

U Geetha¹, Sharmila Sankar² and M Sandhya²

Abstract: Software testing is an important and expensive phase of development. Whenever changes are made in the code, it becomes a time-consuming process to execute all the test cases in regression testing. Therefore, the testing process needs some test case reduction techniques and prioritization techniques to improve the regression testing process. Test case prioritization aims at ordering test cases to increase the fault detection capability. There are many existing techniques for test case reduction as well as for prioritization that use the coverage information which degrades the number of ties uncounted during the prioritization. This paper will take its focus on the multi-level random walk algorithm, which has been used for test case reduction. In this process, test case selection for further reduction is done randomly on every iteration that degrades the performance of a testing process in terms of coverage and will also generate a situation for random test case tie. To overcome this situation of random test case selection and handling test case tie, a solution is being proposed in this paper, which includes a combination of optimized multi-level random walk and genetic algorithm. In regression testing, another important aspect is the test case prioritization that finds fault as early as possible if test cases are prioritized properly. So, this paper introduces new prioritization techniques, which are based on fault prediction in acceptance testing. The performance of the proposed approach in terms of fault detection is evaluated with the help of many programs.

Subjects: Software Engineering & Systems Development; Computer Engineering; Computer Science (General); Ultrasonics; Information & Communication Technology (ICT)

Keywords: Test data generation; test reduction techniques; test case optimization; test case prioritization
1. Introduction

Software testing is expensive for executing code with the intent of finding bugs in the software product. It also deals with the validation and verification process that verifies whether the product meets the technical and business requirements (Chaturvedi and Kulothungan, 2014). Regression testing includes two important parts: functional and non-functional testing. Regression testing is a part of software testing, which means retesting the code after changing the parts of the application. It is a process of testing in which the test cases are re-executed to check whether the previous functionality is working properly (Anand et al., 2013). In regression testing, many issues are faced in terms of resources, time, and cost. So, to reduce the cost of regression testing, software testers introduced the concept of prioritization. Prioritization is done to find the useful and representative set of test cases, by some measure, which is made to run on earlier phases of the regression testing process. (Alian et al., 2016; Fraser et al., 2014; Gilgotic et al., 2015)

The test case prioritization problem is defined as finding the different sequence of test cases for which the values of fault detection are achieved. (Caprara et al., 2000; Wong et al., 1998)

Definition:

Given: $T$, a test suite, DPT, the set of different permutations of $T$. “$F$” is a function from DPT to the numbers.

Problem:

Find $T'$ in DPT such that

$\exists \ T'' \in \text{DPT}$,

$F(T') \geq F(T'')$

Many existing techniques for reduction and prioritization use code coverage information gathered through instrumentation and execution of the code order test that degrade the performance of fault definition rate. (Eghbali & Tahvildari, 2016; Di Nardo et al., 2013)

Some other iterative and greedy approaches iterate “$n$” times, where “$n$” is the number of test cases in the test suite. In each iteration, it selects one test and keeps it into the ordering as the next item.

Sepehr Eghbali and Ladan Tahvildari (Sebastian Elbaum et al., 2000) developed test case prioritization using Lexicographical ordering for improving fault detection capability. They said that most of the approaches use common coverage information from previously executed test cases and use iteration procedure to obtain ordering test cases due to which process will take more time to order test cases. To avoid this problem, they proposed a new heuristic for breaking ties in coverage-based technique. In this paper, initially, they argue that acting randomly in the coverage of ties can degrade the performance of AT algorithm (Duggal and Suri, 2008). They used this proposed approach for breaking the ties effectively. He proposed a basic algorithm using the lexicographical ordering of commutative coverage vector and GetLO algorithm by modifying the basic algorithm to reduce its time complexity.

Sultan H. Aljahdali et al. (Roongruangsawan and Daengdej, 2005–2010) discussed genetic algorithm (GA), feature, and limitation of GA in software testing. In this paper, initially, they have discussed elements of the GA, initial population, calculating fitness value, selection, crossover, mutation and stop criteria, etc. In the second part of the paper, they have discussed and analyzed different approaches which are based on GA and the kind of coverage and fitness function used in the method. At the end of this paper, they have mentioned some limitations that occur when used in the following situation. (i) Using control flow coverage, (ii) simple genetic operator, (iii) not considering some data type and multiple procedures, (iv) manually selecting a path, (v) randomly selecting the initial population, and (vi) solid fitness function. Finally, the authors have said that two parameters that give higher fitness to inputs are considered closer to satisfy the test requirement. The parameters are controlled dependency and branch distance. (Fraser et al., 2014; Yoo & Harman, 2007)
In this research, the GA module is used to find optimizing test cases during the test case reduction process for handling test case ties. The procedure is described as follows:

Experimental set for GA

While (Termination not true)

Do Begin

Population initialization, Selection, Crossover, Mutation,

Replacement for next generation

End

An extensive review has been carried out to find faults in the existing literature, which says that the existing reduction technique used in this paper faces some problem with test case tie due to random selection of test cases on every iteration thus making the overall test suite complex. Therefore, to improve the test suite and reduce the complexity by maintaining the coverage ratio some optimization techniques have been introduced. It considers two optimization techniques which include a combination of optimized multi-level random walk and optimized algorithm (GA). The multi-walk algorithm is a test suite reduction algorithm that finds local and global optima by random walk search to simplify the original problem into a reduced test suite through the backbone and by removing shielded test cases. On the other hand, to improve the ordering of test cases and to reduce the prioritized test cases an optimization algorithm (GA) is used, which is very powerful and is a widely used stochastic search process. A GA is an evolutionary algorithm based on natural selection. It is to find approximate solutions for optimization and search problems. The genetic algorithm aims to achieve better results through selection, crossover, and mutation. (Chen and Lau, 1998; Solanki and Singh, 2014)

A multi-level random walk is a software test case reduction technique that is taken as a focus area in this research. It tries to find an optimal and refined solution for the original problem instance. (McMaster & Memon, 2007)

**1.1. Model for test data generation for multiple path**

The CFG of a program is a diagram \(G = (N, E, S, e)\)

where \(N = \text{Set of nodes}, E = \text{set of edges}, S = \text{Starting node}, e = \text{exit node of the graph.}\)

Each node “\(m\)” is a statement in the program.

Each edge \((m_i, m_j)\) indicates a transfer control from node \(m_i\) to \(m_j\).

A path in the sequence \(P4 = m_1, m_2 \ldots m_n\) Such that there exists edge from node \(m_i\) to \(m_{j+1}\)

where \(i = 1, 2 \ldots n-1\).

Here sequence path varies for larger application to reduce the complexity. We can use string \((0, 1)\).

\[
0, PA \text{ not includes branch (i)}
\]

\[
C_{(i)} =
\]
1, PA includes branch (i)

Let input vector \( V = (x_1, \ldots, x_n) \) and domain of \( x_1 \) be \( D_i \);

Input domain \( D_{\text{program}} = D_1, xD_2, \ldots, D_S \)

Then here program accept “\( V \)” as input, then path denoted by \( PA(V) \).

1.2. Objective function

Applying GA for test reduction problem. The approach to forming an objective function consists of two parts, one is approach level (AL) and another one is branch distance (BD).

The approach level deals with how execution comes to the conditional node, which controls the testing object.

If \( PA \neq PA(V) \), approach level of input \( V \) to a target path \( PA \) is number string between \( PA \) and \( PA(V) \), Otherwise, \( AL_{PA}^{(0)} = 0 \).

For example:

Conditional Statement

(If \( C \leq 10 \) Then branch distance of \( V \) defined as

\[
0 \text{ if } C(V) \leq 10 \\
10 + C(V) \text{ otherwise}
\]

1.3.1. Model for genetic algorithm

Genetic algorithm one of the most popular optimization algorithms that are based on natural genetics and selection process. Before applying GA to any problems whole units are divided into a small unit that is called genes.

The main steps of genetic process are as follows.

1) Generating number of population that is equation no of test cases in test suite. (P)
2) Set up the termination criteria \( T \)
3) Calculate Cyclomatic complexity to find the number of independent paths.
4) Calculating cross over probability \( C_p \)
5) Calculating mutation probability \( M_p \)
6) Generate initial population

\( G_j = \{ g_{j1}, g_{j2}, \ldots, g_{jn} \} \)

Suppose target path \( i \) is \( \{ P_1, P_2, \ldots, P_n \} \)

Then generate subpopulation randomly

\( N^{(1)}(P_j) = \{ V_{i1}^{(1)}, V_{i2}^{(1)}, \ldots, V_{im}^{(1)} \} \)
i = 1,2, … , n and calculate the value of $AL_{p1}(V_i(t))$. Here $t$ is iteration.

7. Calculate fitness value of each test case that means each individual $V_i$

$$f(d_j) = \sum_{i=1}^{k} w_i/C(V_j)$$

$C(V_j)$ is the cost of $V_j$.

For subpopulation

$$f(V_j(t)) = AL_{p1}(V_j(t) + BD_{p1}(X_j(t))$$

8. Generation cross-over mutation operation based on weight age of statement (or) cost of the module.

Algorithm 1.1 GA algorithm

The remaining part of the paper is organized as follows. Section 2 describes the problem description and the multi-walk algorithm for the test case reduction technique. Section 3 introduces the enhanced multi-walk algorithm with help of a GA for handling test case ties during the reduction process and also presents a proposed model for test case prioritization. Section 4 presents performance analysis with the existing approach. Section 5 describes the empirical studies. Section 6 describes the related work. Finally, Section 7 concludes the paper and presents some future research.

2. Problem description

A multi-level random walk is a software test case reduction technique that is taken as an area for research. It tries to find an optimal and refined solution for the original problem instance. However, this algorithm still has few shortcomings associated with it. At every level, a search is being performed, selection of random test cases is made and this selection of random test cases increases the complexity of the entire test suite. Moreover, re-execution of test cases will affect regression testing making it time-consuming and expensive process. Further, there will be a situation when the random test cases will meet a test case tie at some point of time, which leads to statement coverage ratio being impacted. To overcome all such scenarios and to make the overall test suite more effective a solution that is preferred is the incorporation of optimization technique (GA) with the existing reduction and prioritization technique. In regression testing, another important one is test case prioritization that finds fault as early as possible if test cases are prioritized properly. But most of the existing prioritization techniques do not consider the effectiveness of acceptance techniques. Due to this reason, this paper introduced new prioritization techniques which are based on fault prediction in acceptance testing. (Fraser & Arcuri, 2013; Girgis, 2005)

2.1. Multi-walk algorithm for test suite reduction

Test case reduction is important for regression testing because no test cases affect the cost of the regression testing process. In this situation, the system needs effective test cases from the original test suite to check whether the existing products are getting affected by the modified ones.

A multi-level random walk algorithm is used in this paper for test case reduction. One of the most common algorithms for test case selection is the random walk algorithm that uses local optima and backbone test cases to simplify the original problem into small problems by removing the shielded test cases. At each level, a random walk is made and an intersection or the common part is locked, discarding those test cases which are not locked or not shielded. But this algorithm reduces the problem through random selection during the selection process that removes some effective test cases. To overcome the problem of this, the proposed approach uses genetic and
multi-walk algorithm for optimizing test cases instead of random selection. (Akimoto et al., 2015; Watkins, 1995)

At this point the solution obtained by multi-level random walk is not much optimized due to the fact that statement coverage ratio is not maintained properly hence some optimized algorithms can be thought of as invocation with it.

2.1.1. Initial coverage matrix

Table 1 shows the initial coverage matrix of all the test cases, the set of statements in the program, and its weight. Here intersect value represents coverage information about the test case. If statement executed by test case, then marked ‘1’ otherwise ‘0’.

Following steps are followed in the multi-walk algorithm for the reduction process:

Step 1: Selection of two random test cases, i.e. (T3, T7)

Step 2: Selection of these test cases is covering statements (S2, S6)

Step 3: These test cases and the statements that are taken into consideration will become 0 and will contribute to a reduced level 1 matrix.

Step 4: Now, these covered statements and test cases will get a locally optimal solution by finding shielded test cases.

Step 5: Here, the shielded test case taken is (T6) which is shown in the reduction level 1 matrix.

Step 6: Further, next random test cases selected are (T4, T5) covering statements S1, S3, S4, S5, S7.

Step 7: Reductive level 1 will give a reduced and refined matrix but the solution so obtained is not the expected optimal solution according to the statement weightage covered and the remaining test cases will not even contribute to any statements and test case coverage.

2.1.2. Reductive level 1 matrix

Backbone test cases for initial coverage matrix: (T3, T7)

Shielded test case: (T6)

| Test case | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|-----------|----|----|----|----|----|----|----|
| T1        | 1  | 0  | 0  | 1  | 1  | 0  | 0  |
| T2        | 0  | 0  | 1  | 0  | 1  | 1  | 1  |
| T3        | 0  | 1  | 0  | 0  | 1  | 1  | 0  |
| T4        | 1  | 1  | 0  | 1  | 0  | 0  | 1  |
| T5        | 1  | 1  | 1  | 0  | 1  | 0  | 0  |
| T6        | 0  | 0  | 0  | 1  | 1  | 1  | 1  |
| T7        | 0  | 1  | 0  | 0  | 0  | 0  | 0  |
Backbone test cases after reductive level 1 matrix: (T4, T5)

Therefore, after covering most of the test cases and statements the entire matrix is on the verge of becoming 0 throughout, and the optimal solution found by test cases (T3, T4, T5, T7).

The above entire reduction process is shown in Table 2.

2.1.3. Test case reduction percentage
After performing the multi-level random walk, the test case reduction percentage comes out to be 57%, which has more probability of getting enhanced and improvised.

2.1.4. Effectiveness of test cases in terms of statement weightage
If we find the statement coverage and its effectiveness in terms of statement weightage it comes out to be: 4.7

\[
T3 = 1 \times 0.6 = 0.6 \\
T4 = 1 \times 0.5 + 1 \times 0.3 + 1 \times 0.2 + 1 \times 0.8 = 1.8 \\
T5 = 0.5 + 0.3 + 0.8 + 0.4 = 2 \\
T7 = 0.3, \\
\text{Total} = 0.6 + 1.8 + 2 + 0.3 = 4.7
\]

3. Proposed approach for test case reduction and ordering

3.1. Optimized multi-walk algorithm for test reduction
The comparison proves that the GA is more agile and profound in terms of coverage weightage w.r.t multi-level random walk. But to achieve the optimal solution a proposed approach is thought of which describes a combination of multi-level random walk and optimized algorithm (GA). The proposed approach of a combined algorithm is more effective and improvised than the existing methodology.

Genetic algorithms or hereditary algorithms are effective, broadly relevant stochastic pursuit and advancement strategies in light of the thoughts of normal choice and characteristic assessment. It chips away at a populace to the advancement issue. These issues, which either cannot be defined in correct or in an exact numerical shape, may contain uproarious, or sporadic information or they basically cannot be settled by the customary computational techniques.

3.1.1. Algorithm for test case reduction
# Combination of optimized multi-walk algorithm (OMA) and GA:
Input:

\[ T = \langle T_1 \ldots T_n \rangle \text{ set of test cases} \]

\[ S = \langle S_1 \ldots S_n \rangle \text{ set of statements} \]

R1- Reduction level 1

Output:

S1- Reduced test-suite

for \( i = 1 \) to \( R_1 \)

\( F = \text{call genetic()} \)

\( b_i = b_i \cup F \)

Collect Refined and reduced test cases covered by \( F \)

and reduce instance

Find shielded test cases (SHi)

Discard SHi in \( T \)

End

Return \( (T = F) \)

Int[] genetic()

//initial population & return value

For every Test case \( T_i \) in \( T_n \) and for every statement \( S_i \) in \( S_n \)

If (\( T_i \) covering \( S_i \))

\( \text{Mark} = 1; \)

\( A[i] = A[i] + w; \)

Else

\( \text{Mark} = 0; \)

1. Select two test cases which has highest weight and find the target

While (target not reached)

2. If (! Target)

Crossover (\( T_1 \ (2), \ T_2 \ (6) \))
3. If (! Target)

Mutation (Crossover (1))

4. if (! Target)

Select highest fitness one and one from existing suite

Repeat step 1

Else

return (Selected test cases)

End For

End Algo

3.2. Algorithm 3.1 Optimal Multi Walk Algorithm

3.2.1. Initial coverage matrix

Table 3. shows the initial coverage information of all the test cases. This information will be considered as an initial population of genetic algorithm and Table 4 presents the initial setup of the genetic process.

| Test case | m1 | m2 | m3 | m4 | m5 | m6 | m7 |
|-----------|----|----|----|----|----|----|----|
| T1        | 1  | 0  | 0  | 1  | 1  | 0  | 0  |
| T2        | 0  | 0  | 1  | 0  | 1  | 1  | 1  |
| T3        | 0  | 1  | 0  | 0  | 1  | 1  | 0  |
| T4        | 1  | 1  | 0  | 1  | 0  | 0  | 1  |
| T5        | 1  | 1  | 1  | 0  | 1  | 0  | 0  |
| T6        | 0  | 0  | 0  | 1  | 1  | 1  | 1  |
| T7        | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

Table 4. Genetic parameter setup

| Genetic steps | Parameters |
|---------------|------------|
| Initial population | Statement coverage information is used with respective chromosomes as '0' (or) '1'. |
| Fitness value | Calculated by fitness value $F(t) = \sum_{i} w_i \cdot \text{coverage}(S_i)$ |
| Selection | Tournament based selection |
| Crossover | 1st and 2nd of First fitness value and 4th and 5th of 2nd fitness value of the test cases |
| Mutation | 5th bit of crossover result (discrepant bit) |
| Termination | New four test cases to be tested that should have 75% coverage value. |
3.2.2. Configuration setting for genetic algorithm

3.2.3. Reduction process

3.2.3.1. Initial population. The bits stored in the form of ‘0’ or ‘1’ in the coverage matrix are the chromosomes here for the reduction techniques.

3.2.3.2. Fitness values. The fitness values can be calculated using with the associated test cases by the formula $F(t1) = \sum wi*coverage\ statement(Si)$. The calculated fitness value of all the test cases that are shown in Table 5.

For further manipulation we will select those test cases that have the highest weighted fitness value.

3.2.3.3. Selection. Here genetic loop uses selection on the basis of tournament-based selection. Select any two test cases and perform XOR operation as shown in Table 6.

3.2.3.4. Crossover. Crossover can be performed on the same two types of bits and again the XOR operation is performed to get the crossover value satisfying 75% coverage on statements shown in Table 7.

4. Mutation

Most of the bits of the output will get changed into 1, and after performing mutation, it will be on the verge of achieving 100% coverage on statements, which will become 1 throughout after crossover. Now, suppose if 100% coverage on the statement is not achieved, then it is needed to continue for all possible combinations. All combinations will involve the fitness value with the highest weightage and a moment will come where all these combinations will contribute to achieving the 100% target. In this example, we can take (T2, T6) as a backbone test case for the reduction problem.

As we have performed a multi-level random walk in the initial coverage matrix, it is difficult to get the optimal solution by reduced test cases, so, the invocation of a GA is performed to compare the effectiveness of a GA with a multi-level random walk.

| Table 5. Fitness value for test cases |
|--------------------------------------|
| T1 | T2 | T3 | T4 | T5 | T6 | T7 |
|----|----|----|----|----|----|----|
| 1.1| 2.6| 0.6| 1.8| 2  | 2  | 0.3|

| Table 6. Output after selection |
|----------------------------------|
| T2 | 0   | 0   | 1   | 0   | 1   | 1   | 1   |
| T6 | 0   | 0   | 0   | 1   | 1   | 1   | 1   |
| XOR| 0   | 0   | 1   | 1   | 1   | 1   | 1   |

| Table 7. Output after crossover |
|----------------------------------|
| T2 | 0   | 0   | 1   | 0   | 1   | 1   | 1   |
| T6 | 0   | 0   | 0   | 1   | 1   | 0   | 1   |
| XOR| 0   | 1   | 1   | 1   | 1   | 1   | 1   |
We will take (T2, T6) as backbone test cases for the reduced level 1 matrix, which will cover the respective coverage statements as depicted. The output of each level of reduction is shown in Tables 8-10 respectively. The output of each level of reduction is shown in Tables 8-10, respectively.

Here two shielded test cases are possible for test cases T2 and T6, i.e. T1 and T4. But we will go for T4 as it is covering more no of 1 and moreover T1 will ultimately become optimal after further coverage with T4 as a shielded test case. Only one backbone is needed to be selected so no need to go for optimization. After third level reduction, all the intersecting values of the table become zero because all the statements are covered by selected test cases that are shown in Table 11.

As 1 is left uncovered in the T5 test case, therefore, it is taken as the optimal solution: (T5)

| Test case | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|-----------|----|----|----|----|----|----|----|
| T1        | 1  | 0  | 0  | 1  | 1  | 0  | 0  |
| T2        | 0  | 0  | 1  | 0  | 1  | 1  | 1  |
| T3        | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
| T4        | 1  | 1  | 0  | 1  | 0  | 0  | 1  |
| T5        | 1  | 1  | 1  | 0  | 1  | 0  | 0  |
| T6        | 0  | 0  | 0  | 1  | 1  | 1  | 1  |
| T7        | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

| Test case | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|-----------|----|----|----|----|----|----|----|
| T1        | 1  | 0  | 0  | 1  | 1  | 0  | 0  |
| T2        | 0  | 0  | 1  | 0  | 1  | 1  | 1  |
| T3        | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
| T4        | 1  | 1  | 0  | 1  | 0  | 0  | 1  |
| T5        | 1  | 1  | 1  | 0  | 1  | 0  | 0  |
| T6        | 0  | 0  | 0  | 1  | 1  | 1  | 1  |
| T7        | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

| Test case | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|-----------|----|----|----|----|----|----|----|
| T1        | 1  | 0  | 0  | 1  | 0  | 0  | 0  |
| T2        | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| T3        | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| T4        | 1  | 1  | 0  | 1  | 0  | 0  | 0  |
| T5        | 1  | 1  | 1  | 0  | 0  | 0  | 0  |
| T6        | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| T7        | 0  | 1  | 0  | 0  | 0  | 0  | 0  |
A reduction Test case in terms of weightage:

Backbone test case 1 = (T2, T6)

Backbone test case 2 = (T4)

Optimal solution = (T5)

Total weightage of statement covered by all the test cases = $2.6 + 2 + 1.8 + 2 = 8.4$

Therefore, after covering most of the test cases and statement the entire matrix is at the verge of becoming 0 throughout and the optimal solution found by test cases (T2, T6, T4, T5).

4.1. Test case prioritization based on fault prediction in acceptance testing

Considering the coverage matrix for the modules:

4.1.1. Initial matrix

In the initial matrix, we will take some modules with the designated sizes that are shown in Table 12. Each module will have different sizes. Some manipulation will be done by adding some new statements and some statements in the module will be taken from the reused code. The probability of faults in newly added statements will be more compared to reused code as already, in the reused code testing is performed at least once. The test case coverage ratio in each module can be seen as shown in Table 13:

4.1.2. Test case coverage in each module

Reduced test cases from the coverage matrix are covering the number of statements in all modules. The values covering the statements are assumed and then the reduction is performed.
Regression testing is a very expensive phase in testing and time consuming. It is not possible to conduct regression testing in every phase of the testing to deliver a quality product to users. Additionally, it never provides or sequences to the test cases and therefore acceptance testing is applied to get a quality product by sequenced test cases.

Now, suppose there is some previous release done for the product. Therefore, fault in acceptance testing from the previous release includes Table 14.

4.1.3. Fault in acceptance testing from previous release
There are many possible strategies to get the value of test effort of every module after prediction. In this paper, we will use some of the prediction values for assessment and allocation of test efforts.

4.1.4. Algorithm for test effort calculation

[A1] Test effort is directly proportional to module size

We will take a set of module (M1 . . . Mn), allocated test effort $E_i$ for ith module $M_i$, which is defined as: $E_i = E_{total} * K_i / K_{total},$

where $E_{total}$ = test effort taken by all modules, $K_i$ = ith module size

$K_{total}$ = total size of all modules.

This strategy works well for assessing larger modules for test efforts easily. The test effort for its module size of each module is calculated and presented in Table 15.

Ordered test case: T5, T2, T4, T6

[A2] Test effort is directly proportional to new/modified size + 0.1*reused size

Allocated test effort $E_i$ is given as

Table 13. Coverage matrix for each module

| M/T | M1 | M2 | M3 | M4 | M5 | M6 | M7 |
|-----|----|----|----|----|----|----|----|
| T2  | 0  | 0  | 2  | 0  | 4  | 4  | 1  |
| T6  | 0  | 0  | 0  | 6  | 8  | 4  | 4  |
| T4  | 3  | 2  | 0  | 2  | 5  | 5  | 2  |
| T5  | 4  | 2  | 3  | 0  | 8  | 0  | 0  |

Regression testing is a very expensive phase in testing and time consuming. It is not possible to conduct regression testing in every phase of the testing to deliver a quality product to users. Additionally, it never provides or sequences to the test cases and therefore acceptance testing is applied to get a quality product by sequenced test cases.

Table 14. History of acceptance testing

| Module | No. of faults | Test cases used | Rejected test case | Prediction |
|--------|---------------|-----------------|--------------------|------------|
| M1     | 2             | T4, T5          |                    | No         |
| M2     | 0             | T4, T5, T3      | T3                 | No         |
| M3     | 0             | T2, T5          | T2                 | No         |
| M4     | 2             | T1, T4, T5      | T1                 | Yes        |
| M5     | 5             | T1, T2, T3, T5, T6 | T1, T3       | Yes        |
| M6     | 1             | T2, T3, T6      | T3                 | Yes        |
| M7     | 2             | T2, T4, T6      |                    | No         |
Ei = E_{total}(K_{i [new]} + 0.1*K_{i [reused]})/(K_{total [new]} + 0.1*K_{total [reused]}),

where K_{i [new]} = modified new module

K_{total [new]} = sum of all modified code

K_{total [reused]} = sum of all reused code

This analysis is an improvement of A2 algorithm, which differentiate new code and reused code. It takes 10% of reused lines of code and is considered conventional for some of the companies. The test effort for its code size of each module is calculated and presented in Table 16.

Ordered test case: T5, T2, T4, T6.

[B1] Test Effort is directly proportional to no. of predicted faults

Allocated test efforts Ei is given as: Ei = E_{total}PFi/PF_{total}

where Ei = allocated test effort E_{total} = total of all test efforts

PFi = Predicted faults of ith module PF = sum of predicted faults in a module

This method applies to all those parts of modules where the prediction of fault is quite high and is a straightforward method that is preferred and that results presented in Table 17. Ordered test case: T5, T6, T2, T4.

[B2] Test effort directly vs. predicted fault density

Test effort Ei is defined as

| Table 15. Test effort vs. module size |
|--------------------------------------|
| Test case/ Module | M1 | M2 | M3 | M4 | M5 | M6 | M7 | Effort |
|-------------------|----|----|----|----|----|----|----|--------|
| T2                | 0  | 0  | 1.40 | 0  | 5.26 | 2.45 | 2.10 | 11.25  |
| T4                | 1.75 | 2.80 | 0  | 4.21 | 0  | 0  | 2.10 | 10.37  |
| T5                | 1.75 | 2.80 | 1.40 | 4.21 | 5.26 | 0  | 0  | 15.42  |
| T6                | 0  | 0  | 0  | 0  | 5.26 | 2.45 | 2.10 | 9.81   |

| Table 16. Test effort vs. code size after modification |
|-----------------------------------------------|
| Test case/ Module | M1 | M2 | M3 | M4 | M5 | M6 | M7 | Effort |
|-------------------|----|----|----|----|----|----|----|--------|
| T2                | 0  | 0  | 3.33 | 0  | 3.13 | 1.64 | 4.77 | 12.87  |
| T4                | 3.13 | 1.64 | 0  | 1.99 | 0  | 0  | 4.77 | 12.56  |
| T5                | 3.08 | 1.59 | 3.18 | 2.23 | 2.99 | 0  | 0  | 13.06  |
| T6                | 0  | 0  | 0  | 0  | 3.08 | 1.59 | 4.92 | 9.59   |
\[
E_i = \frac{E_{\text{total}} (PFi/Ki)}{\text{summation of } (PFi/Ki)}.
\]

where \(E_i\) = allocated test effort \(E_{\text{total}}\) = total of all test efforts \(PFi\) = predicted faults in modules \(Ki\) = size of the \(i\)th module

Fault density is given more emphasis for the assessment of large modules consisting of faults. The test effort for its predicted fault of each module is calculated and presented in Table 18.

Ordered test case: T5, T4, T2, T6

4.1.5. Test case mapping for final ordering of the test cases
According to test case order of the above algorithm, the new order has been generated based on its occurrence position in the four different order of the test cases

Ordered test case [A1]: T5, T2, T4, T6

Ordered test case [A2]: T5, T2, T4, T6

Ordered test case [B1]: T5, T6, T2, T4

Ordered test case [B2]: T5, T4, T2, T6

Final ordering is: T5, T2, T4, and T6

5. Performance analysis

5.1. Performance analysis basic and optimized multi-walk algorithm for test case reduction
Figure 1 shows that what are the test cases are selected using optimal multi-walk algorithm and basic multi-walk algorithm. It also shows that OMA is more efficient than BMA in terms of statement coverage.

| Test case/Module | M1    | M2    | M3    | M4    | M5    | M6    | M7    | Effort |
|------------------|-------|-------|-------|-------|-------|-------|-------|--------|
| T2               | 0     | 0     | 0     | 0     | 8.33  | 1.66  | 3.33  | 13.32  |
| T4               | 3.33  | 0     | 0     | 1.99  | 0     | 1.6   | 3.33  | 9.25   |
| T5               | 3.33  | 0     | 0     | 3.33  | 8.33  | 1.66  | 3.33  | 19.95  |
| T6               | 0     | 0     | 0     | 3.33  | 8.33  | 1.66  | 3.33  | 16.65  |

| Test case/Module | M1    | M2    | M3    | M4    | M5    | M6    | M7    | Effort |
|------------------|-------|-------|-------|-------|-------|-------|-------|--------|
| T2               | 38.09 | 0     | 0     | 15.85 | 31.71 | 13.60 | 31.71 | 130.95 |
| T4               | 38.09 | 0     | 0     | 15.85 | 31.71 | 13.57 | 31.74 | 130.96 |
| T5               | 38.09 | 0     | 0     | 15.85 | 31.71 | 13.57 | 31.74 | 130.96 |
| T6               | 38.09 | 0     | 0     | 15.85 | 31.71 | 13.57 | 31.71 | 130.93 |
5.2. **Performance analysis of random and proposed prioritization technique for test cases prioritization**

Figures 2 and 3 shows how test cases are prioritized based on A1 and A2 and improves the performance of the testing process compared to random prioritization techniques.

Figures 4 and 5 shows how test cases are prioritized based on B1 and B2 and improves the performance of the testing process compared to random prioritization techniques.

*Figure 6* shows how test cases are prioritized based on A1, A2, B1, B2, overall test effort (A1, A2, B1, and B2) and improves the performance of the testing process compared to random prioritization techniques.

5.3. **Discussion on the results**

**RQ1:** How statement coverage improved?

*Table 19* shows that reduced test cases and its coverage information from different reduction techniques such as GA), OMA, Greedy, coverage-based technique (CBT), and basic multi-walk algorithm (BMA). From *Figure 7*, OMA is more efficient than other techniques in terms of coverage statement. basic GA is somewhat close to OMA but some of the effective test cases are rejected during the reduction process due to consideration of only coverage information.
RQ2: How fault detection rate improved?

Table 20 shows that prioritized test cases and fault detection rate from different prioritization techniques such as random and acceptance-based test case prioritization. From Figure 8,
Acceptance-based test case prioritization is more efficient than other techniques in terms of fault detection rate because acceptance-based test case prioritization is covering most of the statement as early as possible.

6. Related works
Shubhra Bajerji developed a technique called the orthogonal array approach for the reduction of test cases. The author has discussed a different existing method to achieve maximum test
coverage (Elbaum et al., 2002). But there was some problem. Due to this problem, the proposed approach uses an orthogonal array approach for test case optimization (Harman et al., 2012; Gupta et al., 2012). In the orthogonal array approach, they have used two main terminologies that are FACTOR (f) Parameter that the tester intentionally changes during the testing, LEVELS (p) different independent value factor. Finally, the author presented two different case studies for different Browser–OS–Database combinations using the orthogonal array approach. The limitations of this approach are that the risk factor is more because it selects few test cases. Another limitation of this is making an assumption of each FACTOR (f) is independent (Hsu and Orso, 2009; Krishnamoorthi and Sahaya Arul Mary, 2009). But the advantage of this approach is clearly brought out with the help of graph analysis (Alian et al., 2016; Yoo & Harman, 2012).

Srirong Roongruangsuwan and Jirapun Daengdej discussed different prioritization techniques such as coverage-based and cost-based, etc (Kosindrdecha and Daengdej, 2010; Lin et al., 2013). In this paper, they have discussed some existing prioritization techniques and problems in the existing method. Most of the existing systems have failed to prioritize multiple suites and test cases with the same priority value. The proposed MTSSP approach has taken the above issue and given a new solution for the problem (Kumar et al., 2013). It also resolves the issue for duplicated values and multiple test suites. But this approach is not suitable for the automatic prioritization of multiple test suites with real commercial data (Deason et al., 1991; Mayers, 2004).

Dongdong Gao et al. proposed a new approach to prioritizing the test cases using the Ant Colony algorithm. In this paper, authors used new parameters to order the test cases (Mei et al., 2012). Parameters are chosen based on faults covered by a test case, execution time, and error detection capability of the test cases. By considering the above-mentioned parameters they find the optimized test cases to test the software (Rothermal and Harrold, 1996). The authors compared their proposed approach with the existing techniques. The proposed systems (average percentage of fault detection (APFD) is higher than the existing techniques (Moshini and Bardsiri, 2015; Pargas et
al., 1999). Even there are some test cases that have more effective APFD, which are rejected by the existing reduction technique in which the proposed approach is used. (Fraser et al., 2014; Sihan et al., 2010)

Daniel Di Nardo et al. created coverage-based test case prioritization with one industrial case study. In the first paper, the author mentioned how to find different coverage information such as total coverage, additional coverage, total coverage of modified code, and additional coverage of modified code. In the second part, the author provides answers to a few research questions after the analysis of the case study. The research questions are how granularity of coverage criteria affects fault detection, how different coverage-based test case prioritization technique compared in terms of fault detection capability, whether coverage-based technique improve fault detection (Roongruangsuwan and Doengdej, 2005-2010; Rothermal et al., 1998). In the third part, the author has given an experimental design with the help of the studs tool. By this tool, they generated test cases, identify faults, measured effectiveness, data collection for obtaining the coverage information, etc. Finally, the author has analyzed and reported the effect of coverage prioritization and effectiveness of prioritization to answer the (Rout et al., 2011) research questions. (Andrews et al., 2011; Rothermal et al., 2001)

7. Conclusion and future work
On taking a tour through this paper, we experienced that regression testing is a very expensive form of testing that includes re-execution of all test cases thus making the overall test suite more cumbersome and leading to an increase of test cases. So to enhance the regression testing initially a multi-level random walk is performed and a common intersection point is found but still, that walk being performed is not optimized. Therefore, an optimized random walk is made which tries to decrease the redundancy in the test suite and improves the overall test suite. When it comes to redundancy, test case prioritization also plays a vital. To get a more profound test suite one of the optimization techniques (GA) is incorporated with the existing multi-level random walk reduction technique. Thus, a combination of both reduction and prioritization techniques with optimization technique (GA) is a good recommendation which is tried to be concluded in this paper.

In this paper, we have tried to overcome the random test case tie situation and to decrease the redundancy in the test suite by using a combination of multi-level random walk and optimization algorithm (GA). But prioritization can also work as a very good remedy for reducing the redundancy of test cases in the test suite as it will provide a sequence and order to test cases, which can help in simplifying the entire test suite. In the future, a good prioritization technique can be combined with an optimization technique. Moreover, some faults will also be generated at the time of the acceptance testing phase that can also be worked upon for optimization using some reduced and prioritized optimization techniques.

References
Akimoto, S., Yaegashi, R., & Takagi, T., “Test Case Selection Technique For Regression Testing Using Differential Control Flow Graphs” IEEE 2015
Alian, M., Suleiman, D., & Shaout, A., “Test case reduction techniques: survey” IJACSA 2016
Anand, S., Burke, E., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Zhu, H., Harold, M. J., McMinn, P., Bertolino, A., Jenny Li, J., & Zhu, H. (2013, August). An orchestrated survey on automated software test case generation. Journal of Systems and Software, 84(8), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061
Andrews, J. H., Menzies, T., & Felix, C. H. L. (2011). Genetic algorithms for randomized unit testing. IEEE Transactions on Software Engineering, 37(1), 1–15. https://doi.org/10.1109/TSE.2010.46
Bach, J., “Useful features of a test automation system (part iii)”, Testing Techniques Newsletter, 1995.
Caprara, A., Toth, P., & Fischetti, M. (2000, December). Algorithms for the set covering problem. Annals of
Operations Research, 98(1–4), 353–371. https://doi.org/10.1023/A:101925072893
Chaturvedi, S., & Kulathungam, A. (2014, March - April). Improving fault detection capability using coverage-based analysis. JOSR-Journal of Computer Engineering (JOSR-JCE), 16(2), 22–30. https://doi.org/10.1109/0661-16262230
Chen, T. Y., & Lau, M. F. (1998). A simulation study on some heuristics for test suite reduction. Information and Software Technology, 40(13), 777–787. https://doi.org/10.1016/S0950-5849(98)00094-9
Deason, W. H., Brown, D. B., Chang, K. H., & C. J. H. (1991). A rule - based software test data generator. IEEE Transactions on Knowledge and Data Engineering, 3(1), 108–117. https://doi.org/10.1109/69.75894
Di Nardo, D., Alshahwan, N., Briand, L., & Labiche, Y. "Coverage-based test case prioritization: An industrial case study", Proceedings of international conference on software testing, verification and validation, 2013, pp.302–311.
Duggal, G., & Suri, B. "Understanding regression testing techniques",2008.
Eghbal, S., & Tahvildari, L. (2016). Test case prioritization using lexicographical ordering. IEEE 12/1178-1195/ 42. https://doi.org/10.1109/tse.2016.2550441
Elbaum, S., Malishevsky, A., & Rothemel, G. (2002). Test case prioritization: A family of empirical studies. IEEE Transaction on Software Engineering, 28(2), 159–182. https://doi.org/10.1109/32.988497
Elbaum, S., Malishevsky, A. G., & Rothemel, G. (2000). “Prioritizing test cases for regression testing", Technical Report, 1–9. University of Nebraska-Lincoln.
Fraser, G., & Arcuri, A. (2013). Whole test suite generation. IEEE Transaction on Software Engineering, 39(2), 276–291. https://doi.org/10.1109/TSE.2012.14
Fraser, G., McMinn, P., & Arcuri, A. (2014). A memetic algorithm for whole test suite generation. The Journal of Systems and Software, 13(48), 1–17. https://doi.org/10.1016/j.jss.2014.05.032
Girgis, M. R. (2005). Automatic test data generation for data flow testing using a genetic algorithm. Journal of Universal Computer Science, 11(6), 898–915. http://dx.doi.org/10.3217/jucs-011-06-0898
Gligor, M., Eloussi, L., & Marinov, D. “Practical regression test selection with dynamic file dependencies,” in Proc. int. symp. software testing and analysis, Baltimore, MD, July 12–17, 2015, pp. 211–222. Basel, Switzerland.
Gupta, S., Raprio, H., Kapur, E., Singh, H., & Kumar, A. (2012, June). A Novel approach for test case prioritization. At IJCSEA, 2(3), 293–302. https://doi.org/10.1109/IJCSEA.2013.6724209
Harman, M., Mansouri, S. A., & Zhang, Y. (2012, November). Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv., 45(11), 1:1–11:61. https://doi.org/10.1145/2379776.2379787
Hsu, H., & Orso, A., “MINTS: A general framework and tool for supporting testsuite minimization,” in Proc. 31st int. conf. software engineering, Vancouver, Canada, May 16–24, 2009, pp. 419–429. Canada.
Kosindroo, N., & Doangdej, J. (2010). A test case generation process and technique. Journal of Software Engineering, 4(4), 265–287. https://doi.org/10.3923/jse.2010.265.287
Krishnamoorthi, R., & Sahaya Arul Mary, S. A. (2009). Regression test suite prioritization using genetic algorithms. International Journal of Hybrid Information Technology, 2(5), 35–52.
Kumar, R., Singh, S., & Gopal, G. (2013). Automatic test case generation using genetic algorithm. International Journal of Scientific & Engineering Research, 4(6), 1135–1141. https://doi.org/10.1109/ IICCE.2013.6724209
Lin, C.-T., et al. (2013). History-based test case prioritization with software version awareness. Engineering Complex Computer Systems (ICECCS).
Mayers, G. J. (2004). The art of software testing. John Wiley & Sons Publication.
McMaster, S., & Memon, A. (2007). Fault detection, probability analysis for coverage-based test suite reduction. IEEE, 12/48-54/6. https://doi.org/10.1109/ ICCSM.2007.6362646
Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., & Rothermel, G., “A static approach to prioritizing junit test cases” IEEE 2012
Moshini, M. M., & Bardasini, A. K. (2015). The application of Meta-Heuristic Algorithms in automatic software test case generation. International Journal of Mathematical Sciences and Computing, 3(12), 1–7. https://doi.org/10.5815/jmsc.2015.03.01
Narjani, D., Gotlieb, A., & Sen, S. “Test Case Prioritization for continuous regression testing: An industrial case study”, Proceedings of internationalconference on software maintenance, 2013, pp.540-543. Netherlands: IEEE.
Pargas, R. P., Harrold, M. J., & Peck, R. R. (1999). Test-data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability, 2 (18),1–19. https://doi.org/10.1002/jsv.1099-1689 (199912194%)43523:tad-strv1099-3e.0.coo2-y
Roorongruangsuan, S., & Doengdej, J. (2005-2010). Test case prioritization techniques. Journal of Theoretical and Applied Information Technology, 45–60.
Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. IEEE Transactions on Software Engineering, 22(8), 529–551. https://doi.org/10.1109/32.536955
Rothermel, G., Harrold, M. J., Ostrin, J., & Hong, C. “An empirical study of the effects of minimization on the fault detection capabilities of test suites”, In Proceedings of IEEE International Test Conference on Software Maintenance (ITC&M’98), 1998, pp. 34–43.
Rothermel, G., Untch, R., Chu, C., & Harrold, M. (2001). Test case prioritization. IEEE Transactions on Software Engineering, 27(10), 929–948. https://doi.org/10. 1109/32.962562
Rout, J., et al. (2011). An efficient test suite reduction using priority cost technique. International Journal of Computer Science and Engineering, 4(7–8), 89. https://doi.org/10.1105/1998.738487
Sihan, L., Bian, N., Chen, Z., You, D., & Yuchen, H., “A simulation study on some search algorithms for regression test case prioritization” 2016,1–10.
Solanki, K., & Singh, Y. (2014). Novel classification test case prioritization techniques. International Journal of Computer Applications, 100(12), 36–42. https://doi.org/10.5120/17580-8356
Watkins, A. L., “The automatic generation of test data using genetic algorithms”, Proceedings of 4th software quality conference, 1995, Volume-2, 300–309. Dundee (UK).
Wong, W. E., Jr, H., London, S., & Mathur, A. P. (1998). Effect of test set minimization on fault detection effectiveness. In Software Practice and Experience (pp. 179–188). wiley.

Yoo, S., & Harman, M. (2007). Regression testing minimisation, selection and prioritization: A survey. In Software testing, verification and reliability (pp. 1–59).

Yoo, S., & Harman, M. (2012, March). Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab, 22(2), 67–120. https://doi.org/10.1002/stv.430