CHAPTER 3

DESCRIPTION OF RESEARCH WORK

3.1 OVERVIEW

This chapter describes the conceptual attitude for finding a way out of the problem described earlier. It analyses the faulty credentials of a software product in an elaborate manner. Necessities of the test cases are defined and dependence between test cases is examined with efficacy and is removed if found excessive. Prioritizing categories of test cases are inspected in detail. Among the methodologies studied, the appropriate one is opted for dissertation. It helps in unstructured, faster implementation, testers, and developers cooperate with each other, and testing is initiated at the end of every logic release.

3.2 PROBLEM DESCRIPTION

The main purpose of testing can be quality assurance, reliability estimation, validation or verification. In the existing approaches, the software testing processing includes a process of sequential execution. The existing mechanisms focus only on the generation of the test cases for this agile software testing. The sequence of test case processing in the testing environment acts as a role in providing the optimal result. But the existing mechanism may not have any appropriate process for producing the execution sequence of test cases. The fault percentage in the testing process also needs more attention to develop the software products. In case of any fault occurrence in testing, the tolerance mechanism is also needed.

The researcher has formulated the scope of the research work with a new proposal on improved Software testing environment for overcome this problem.
Initially, need to focus on the agile software testing mechanism with the Directed Acyclic Graph (DAG) based model. Then, some of the dependency structures for the each module in the software testing of agile are generated. Following this, some structured priorities for this test module are applied. Finally, we need to focus on fault tolerance. Hence, the some dependency and similarity approaches are implemented for predicting the similarity between the models and grouping them using clustering based on the distance value finally. This module is being processed in a parallel manner. It will be very useful for complex and large projects from the researcher’s approach for creating a mechanism to prioritize the test cases based on dependency of test cases with the optimized fault tolerant in order to provide the quality software.

3.3 AGILE TESTING MECHANISM WITH DAG BASED MODEL

3.3.1 Hybrid Engineering Approach

An Agile risk based methodology based on the Hybrid engineering approach [Rahimian, V.R.R., 2008] was used for iterative-incremental development on the basis of a predefined set of requirements and the knowledge gained from the existing models/designs. This approach has been formatted as a top-down iterative, incremental process consisting of following tasks:

3.3.1.1 Prioritization of the requirements

It is performed at the beginning of the process and repeated at the conclusion of each iteration. At the beginning of the process, abstraction is at a higher level and the scopes confines the entire lifecycle. Therefore, the requirement with lifecycle impacts is given priority.
- Instantiation- an already available process meta-model was useful when designing higher level aspects of methodology.
- Artifact oriented- formulates a flawless chain of artifacts and building the process around it.
- Composition- using the process patterns which were available in the library.
- Integration- integrating ideas, features and techniques from existing methodology.

In the first iteration, the practice was detailed through generic rules and development. Analysis was split into preliminary analysis and detailed analysis to reduce errors. The second iteration focuses on the borrowing activities from the New Product Development (NPD) and design process of the second iteration involves reusability of the New Product Development (NPD) process. In the third iteration, the process development engine is enriched by integrating ideas from New Product Development (NPD). The requirement based hybrid approach ensures that the requirement were properly addressed and validated. This approach was easily fitted for the production of mobile software system based on agile methodology.

3.3.2 Agile Testing Approach

In the world of software development, the term agile typically refers to any approach to project management that strives to unite teams around the principles of collaboration, flexibility, simplicity, transparency, and responsiveness to feedback throughout the entire process of developing a new program or product. In the Agile approach, developers and testers are seen as two sides of the same production coin, two parallel lines that should always meet and compare notes daily. From an Agile
perspective, efficient production is severely hampered if the developers are striving to refine their code to a state of perfection before passing it on to a separate testing team, who then strive to break it in as many ways as they can before sending their report on damage back to the development team. This two-step process requires time, money, and frequently leads to internal division between a given company’s developers and testers.

Instead, Agile suggests that these two essential functions be merged—not necessarily in terms of people, but in terms of time and process—thus bridging the illusory divide between code creators and code breakers, and even reducing the need for robust testing teams, while still respecting the necessity of both roles. One could even say that in Agile, developers are encouraged to think more like testers, continually checking their own code for potential errors, and testers are encouraged to think more like developers, tempering their natural destructive tendencies to engage more fully in the creative process itself.

3.3.3 Non Sequential Execution for Agile Process

This methodology investigates the feasibility of identifying, linking, and modeling Agile Loose Cases (ALCs) with Agile Choose Cases (ACCs) and Agile Use Cases (AUCs). The NORMAP methodology was validated through NORMATIC, which is a Java based tool. Agile Use Cases (AUC) were newly suggested hybrid use cases and agile stories. The required quality and the project management metric were used for calculating a risk driven requirement implementation scheme by Non-functional Requirement Model for Agile Process (NORMAP) methodology. Agile groups can substantially benefit from the Non-functional Requirement Model for
Agile Process (NORMAP) methodology by engaging a systematic and risk-driven lightweight engineering process to visually model and plan New Product Development (NPDs) in agile environments.

3.3.4 Use Case and Test Case Estimation Method

This model deals with reducing the gap between estimated effort and real time implement effort. A use case defines the system’s behaviour under different condition as it responds to one of the stake holder’s request, called primary actor. The primary actor initiates an interaction with the system to achieve a goal. This model gathers all the different use cases/scenarios and converts them into numbers and drive. The first step of estimation is to detect size and the next step is to do effort estimation. This model offers effective effort estimation in testing. A Model Driven Architecture (MDA) focuses on the role of models during product development. This is highly effective and prone to fewer errors. This holds in different strategies, code for Model Driven Architecture (MDA) is retrieved from a pattern and use cases are generated from an analysis or behavior models. The model based software development suggests the use of the model as the primary artifact for requirements, documentation, code generation and development. An intensive set of trial cases is imperative for a guarantee of the calibre of an evolving system.

3.3.5 Just-In-Case (JIC) and Just-In-Time (JIT) Approach

This approach ensures better interaction, collaboration and an optimal exploitation in the software development effort. JIC and JIT construct resources for making and retrieving results from social media. In the JIC approach, apparent development tasks, requirement gathering, analysis tasks were covered
simultaneously. This is possible with the collaboration between the analyst and the developer at the initial level. The concept here is to originate and keep ready, a significant number of development projects required for a future project. This approach helps saving of time in completing the task inside the deadline. The JIT approach had been applied during the software evolution operation. If the expected solution is usable, it is readily incorporated into the software project for completing the undertakings. If it is not available, a query is posted to a social media in order to find the best result. This access can produce a common platform and a pool of ideas to address the issue effectively and provide results which are better and economically practicable.

3.3.6 Test Case Generation Algorithm Using DAG

In this algorithm, DAG is a directed graph with a vertex v but with no directed circuits. There is no nonempty directed path (Tuglular, T., et al, 2009). Event sequence graphic is an event based model, where input and events are combined and assigned to the vertices of an event transition diagram. A Complete Event Sequence (CES) simply transmits through an Event Sequence Graph (ESG), where it starts at the entry node and ends at the exit node of the Event Sequence Graph (ESG). The testing approach consists of five stages.

- Generating test cases from the firewall rules
- Constructing network test packets from generated test cases
- Sending constructed packets to Function Under Test (FUT)
- Capturing the packets that pass through the Function Under Test (FUT)
- Comparing sent and captured packets to find the results.
This approach concentrates on generation of test cases. Firewall testing agenda is automated using test case generation algorithm and Directed Acyclic Graph (DAG).

3.3.7 DAG Framework for Parallel Computing

This framework schedules tasks in a fully dispersed and active environment, using all the cores of each node. It comprises a runtime engine and a set of tools to build, analyze and pre-compile a representation of Directed Acyclic Graph (DAG). The internal representation of Directed Acyclic Graph (DAG) used by Directed Acyclic Graph (DAG)ue is called JDF. The JDF representation of the Directed Acyclic Graph (DAG) was pre-compiled as C-code and connected in the final binary program. The Directed Acyclic Graph (DAG)ue library incorporates the runtime environment, comprises of distributed multi-level dynamic scheduler. It is an asynchronous communication engine and a data dependency engine. The Directed Acyclic Graph (DAG)ue engine is responsible for moving data from a single Central Processing Unit (CPU) to another when necessary; tasks are enabled only when all the data are marked as IN. Dependencies of JDF are marked with a modifier, which is a type of qualifier. It provides instructions to the communication engine, on the transmission of the data from a remote location to another. In this framework the performance of Directed Acyclic Graph (DAG)ue engine performance has been investigated.

3.3.8 Parallel Task Model

This model addresses a deterministic similar task where each task is assigned as a Directed Acyclic Graph (DAG) with dissimilar nodes having altered desires. Task decomposition is designed, which splits the Directed Acyclic Graph (DAG) into
sequential tasks. Following the parallel tasks, decomposition is scheduled using pre-emptive global Earliest Deadline First (EDF). Ultimately, the disintegration of a resource augmentation for non-pre-emptive global Earliest Deadline First (EDF) scheduling is seen. In the common task model, responsibilities are represented by Directed Acyclic Graph (DAG) where threads can have random execution needs. Presently, pre-emptive and node point non-pre-emptive scheduling are maintained for these Directed Acyclic Graph (DAG). The decomposition is designed in multi real-time scheduling for the unit code Directed Acyclic Graph (DAG). It is applied to a general Directed Acyclic Graph (DAG) where every node is further split into a smaller node. All subtasks of a segment are synchronized at death, where there is denial path of assuring non-pre-emption. In development, mixed parallel application with an advanced reservation involves various algorithms which comprise two phases. In first phase an algorithm decides the number of processors to be assigned to each project. In the second stage, tasks are scheduled in teams of time and place. The theorems are compared over a wide range of applications and scenarios.

### 3.3.9 Sporadic DAG Task Model

This model is a well-known model for projecting the real time system on the basis of a finite number of independent tasks each of which may propagate an unbounded sequence of jobs. Here, real time workloads can be planned as a cluster of independent sporadic Directed Acyclic Graph (DAG)s and are compiled on a platform, which holds \( n \) identical Central Processing Unit (CPU). The program is fully preemptive and permits global interprocessor migration, where each operation may get accomplished on at most one Central Processing Unit (CPU) at each moment.
of time. The sporadic task model is a model of repetitive processes in hard real time systems (Bonifaci, V., A. Marchetti-Spaccamela, 2012). A sporadic task $Si= (Wi, Ri, Mi)$ where $Wi$ is the worst case execution time, $Ri$ is a relative deadline and $Mi$ is minimum inter arrival separation. This model clearly explains about the feasibility study of sporadic Directed Acyclic Graph (DAG).

3.3.10 Continuous Interaction Model

In this model, members of a team integrate and shape to minimize the duration and the product is delivered at any minute. It is a development pattern that requires the developer to incorporate codes into a shared repository (Stolberg, S., 2009). This model helps prompt as containing the bug. The Continuous Interaction (CI) server supervises the repository and checks out the modifications when they come. The server builds the system and runs unit and integration testing, and then it delivers the deployable artifacts for testing. The server allocates a build label to the explanation of the code it just built. If the test fails, server alerts the team and the team fixes the issue. Continuous Interaction (CI) allows for greater collaboration between development and delivery.

3.3.11 Recursive Method for DAG

In this method an algorithm is proposed for structure learning of Directed Acyclic Graph (DAG). The algorithm consists of a series of operations of a binary tree, where the top node of the tree contains a full set of variables. The leaves of the trees are considered as subsets of variables which cannot be disintegrated [Xie, X.G.Z., 2008]. The algorithm involves two procedures: (a) Top-down steps for decomposing the full set of variables into subsections as small as possible, and (b) Bottom-up step is to integrate the local frame into global frame.
3.4 RESEARCH METHODOLOGIES

The major objective of the research work is to create a mechanism for prioritizing the test case using the Directed Acyclic graph based model in agile software testing environment. When compared to the randomly ordered test cases and the prioritized test case, and the suggested technique increases the fault detection rates. The suggested technique addresses the ties effectively and also the fault detection rate. The Prioritization Deciding Factor (PDF) method used for test case prioritization. Experimental analysis provides that the suggested technique increases the fault detection rate, and also maximizes, the cost effectiveness of the test case prioritization.

3.4.1 An Efficient DAGBM-KSJS Algorithm for Agile Software Testing

3.4.1.1 Introduction

An optimal Agile software testing is performed using the Directed Acyclic Graph-based Model (DAGbM). First, pre-processes the test cases dataset, then the dependency between the use cases are determined through deployment of the Depending Assessment for Use Case (DAUC). The K-Shingling based Jaccard Similarity (KSJS) algorithm estimates the similarity between every test case and prioritizes the clustered test cases using Span Clustering based Prioritization (SCP) algorithm. After prioritizing the clustered test cases, the use cases are prioritized based on dependency. Finally, the minimum distance value is exploited for prioritizing the individual test cases. The performance of the suggested method is validated using the parameters such as code coverage, failure rate, prioritization time, and percentage of defects detected. The validation results prove, the suggested method provides optimal results for all the parameters when compared to the existing methods.
The test cases that were dependent on using the graph coverage values were provided higher priority. The advantages of the suggested approach were minimal speed for the testing process and increased fault detection rate. From the analysis of the existing techniques, it is clear that they do not provide an optimal execution sequence for the test cases. Further, the fault percentage in the testing process is higher. Thus, an efficient Directed Acyclic Graph (DAG) based model is suggested for addressing the issues and performing software testing.

The overall flow of the suggested Directed Acyclic Graph based Model-K-Shingling Jaccard Similarity (DAGbM-KSJS) is represented in figure 3.1. From the figure, it is clear that the key processes involved in the suggested model are as follows:

- Test case pre-processing;
- Use case dependency estimation;
- Similarity estimation over the test cases;
- Optimized test case execution.

Figure 3.1 Overall Designs of DAGbM-KSJS-SPAN Clustering Algorithm
3.4.1.2 DAGBM-KSJS Algorithm for Agile Software Testing

3.4.1.2.1 Test Case Pre-Processing

This Phase model exploits a mobile dataset as input. It provides the use case ID and Test case ID for all entries in the dataset. The initial step involved in the suggested DAGbM–KSJS is test case pre-processing. The suggested model exploits a mobile dataset as input. A sample of the dataset is represented as follows,

**Sample of the input dataset**

*Check for background music and sound effects*# ON/OFF Sound and background music

*Check for background music and sound effects*# Receive the call and check

*Check for background music and sound effects*# verify if sound effects are in synchronization

*User Interface* # Check in Landscape/Portrait mode

*User Interface* # Check for animation, movement of characters, graphics

*User Interface* # there should not be any clipping

During the pre-processing step, the suggested model provides the use case ID and Test case ID for all entries in the dataset. An example of the pre-processing step is given in Table 3.1.

<table>
<thead>
<tr>
<th>S. No.</th>
<th>U_ID</th>
<th>Use case</th>
<th>Test case</th>
<th>Test_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td><em>Check for background music and sound effects</em></td>
<td>ON/OFF sound and background music</td>
<td>TID_1</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td><em>Check for background music and sound effects</em></td>
<td>Receive the call and check</td>
<td>TID_2</td>
</tr>
<tr>
<td>3.</td>
<td>2</td>
<td><em>User Interface</em></td>
<td>Check in Landscape and Portrait mode</td>
<td>TID_3</td>
</tr>
<tr>
<td>4.</td>
<td>2</td>
<td><em>User Interface</em></td>
<td>Check for animation, movement of characters and graphics</td>
<td>TID_4</td>
</tr>
</tbody>
</table>
3.4.1.2.2 Use Case Dependency Estimation

Following pre-processing the dataset, the dependency between the use cases is determined using DAUC algorithm. The suggested algorithm consumes a set of use case and a set of the test case as input. For every use case in the dataset, the dependency between them is estimated using equation (1). Based on the dependency values of the use cases, the test cases are grouped.

3.4.1.2.3 Algorithm-DAUC

\textit{Algorithm 1: DAUC algorithm}

\begin{itemize}
  \item Set of Use cases (UC)
  \item Set of Test cases (TC)
  \item Let i=1, 2, 3… N be no. of UC \_i
  \item Let j=1, 2, 3…N be the no. of TC \_jm
  \item Where m=1, 2, 3… N of UC
\end{itemize}

\begin{algorithm}
\textbf{Begin}
  \textbf{For each} (c ← UC \_i),
  \begin{align*}
  \text{Compute } Z &= \frac{1}{N} \sum_{m=1}^{N} TC \_m \\
  \text{Compute } \text{Dep}(UC) \_i &= \frac{1}{Z}
  \end{align*}
\textbf{Continue until} i=N
\textbf{End for}
\textbf{End}
\end{algorithm}

<table>
<thead>
<tr>
<th>S. No.</th>
<th>TID</th>
<th>UID</th>
<th>Use case</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>TID_1, TID_2, TID_3</td>
<td>1</td>
<td>Check for background music and sound effects</td>
<td>0.2</td>
</tr>
<tr>
<td>2.</td>
<td>TID_6, TID_7, TID_19</td>
<td>2</td>
<td>User interface</td>
<td>0.08</td>
</tr>
<tr>
<td>3.</td>
<td>TID_29, TID_30</td>
<td>8</td>
<td>Save settings</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Based on the dependency values of the use cases, the test cases are grouped in the above table 3.2.
3.4.1.2.4 Similarity Estimation over the Test Cases

After grouping the test cases using the DAUC algorithm, the similarity and distance for every test case is determined using KSJS algorithm. The overall steps involved in the suggested KSJS algorithm are illustrated below. The proposed algorithm consumes the set of use cases and set of the test cases as input. At first, the algorithm initializes the Shingling size and flag variable, then for every test case, the similarity with the other test case is determined using Jaccard similarity represented as follows,

\[ JS(T_i, T_j) = \frac{\left| T_i \cap T_j \right|}{\left| T_i \cup T_j \right|} \] ........................ (1)

The estimated similarity values are split and initialized to the hash set \( H(i, j) \), and then the resultant values obtained from the application of K-shingling algorithm is allocated to the hash sets such as \( H(i) \) and \( H(j) \). When the similarity between the test cases is greater than zero, the flag is incremented. The distance between two test cases is determined using equation (4). When the number of iteration reaches the use case size, a Directed Acyclic Graph (DAG) is constructed.

3.4.1.2.5 Algorithm –K-Shingles Jaccard Similarity (KSJS)

**Input:**
Let \( i=1,2,3…N \) be set of UC \( n \)
Let \( j=1,2,3…N \) be set of TC \( m \)

**Output:**
Similarity & distance Measure of Test Cases

**Begin**
Set \( K=2,3… \) Size of (Strings in TC \( m \)), flag=0
For each \( x \leftarrow UC_i \)
For all TC \( m \) where \( m = 1,2,...N \)
TList \( \leftarrow \) add(TC \( i \))
For each $T_i$ from $T_{List_n}$ $J=i+1$;
Compute Similarity ($T_i$, $T_j$)
$H_{i,j} \leftarrow \text{Split}(T_i, T_j)$
$H(i) \leftarrow \text{Compute } K(\text{shing}(T_i))$
$H(j) \leftarrow \text{Compute } K(\text{shing}(T_j))$
Compare ($H(T_i, H(T_j))$)
If (Similarity>0)
Increment flag
Continue until $N=H_{size}$
Compute $Sim_{val} = \text{flag} / \sum_{x=1}^{N}(Hs(i) + Hs(j))$
Compute $Dist_{(r,r_i)} = 1 - Sim_{val}(T_i, T_j)$
$\text{DAG (UC)} \leftarrow Dist_{(r,r_i)}$
End for
End

3.4.1.2.6 Optimized Test Case Execution

An optimized test case execution is possible through test case clustering and prioritization. Hence, after the construction of DAG, the test cases are clustered and prioritized using SCP algorithm. The distance estimated and the use case dependency values are provided as input to the algorithm. For every use case in the dataset, the average distance between the test cases is estimated and initialized as the threshold value are represented in figure 3.2.

![Fig. 3.2 Example for DAG representation](image-url)
If the distance of a test case is less than or equal to the threshold $T_{Th_i}$, the corresponding test case is added to cluster 1, else the test case is added to cluster 2. They are prioritized after clustering the test cases. Similarly, as represented in equation (7), the use cases are prioritized based on their dependency values. For every use case, the overall distance is computed such that the test cases that have minimum distance are prioritized thus resulting in an optimized test case execution.

3.4.1.2.7 Algorithm: Span Clustering based Prioritization (SCP)

<table>
<thead>
<tr>
<th>Inputs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computed Distance &amp; Dependency values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered &amp; prioritized Test Cases</td>
</tr>
</tbody>
</table>

Begin
For each $Uc_n$ where $n = 1, 2, ... N$
$Th_i = \text{Compute Avg}_d (\text{dis} (Uc_i))$
If (\text{dis} (Tc_i < Th_i) \lor \text{dis} (Tc_i = Th_i))
    Add $(Tc_i) \rightarrow c1 (T_i)$
Else
    Add $(Tc_i) \rightarrow c2 (T_i)$
Continue until $i = Uc_{size}$
Order (C1) $\leftarrow$ $\text{dis} (\text{clus}_{Uc1})$
Order (C2) $\leftarrow$ $\text{dis} (\text{clus}_{Uc2})$
Apply $TPrio (c1, c2)$
End for
For each $(T_{ci}) \leftarrow \text{List(TC)}$
Compute $D_{Prio} = Pr (Dep_{Tc_i})$
Result $\leftarrow \text{Produce} (Pr(TC_N))$
End for
End

Table 3.3 represents clustering of test cases using SPAN clustering algorithm. The use case1 contain 10 test cases and distance value also considered for clustering,
here set the threshold value is 1, and apply clustering for the use cases, according to the threshold value the test cases are split into two group i.e. cluster 1 and cluster 2, in cluster 1 contains less than 1 The complexity of Clustering algorithm is NP Complete.

**Table 3.3 Clustering the test cases using SPAN clustering algorithm**

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Uid</th>
<th>Cluster</th>
<th>DAG</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>1</td>
<td>TID-2</td>
<td>TID-4</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1</td>
<td>TID-1</td>
<td>TID-4</td>
</tr>
<tr>
<td>3.</td>
<td>1</td>
<td>1</td>
<td>TID-1</td>
<td>TID-3</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>1</td>
<td>TID-3</td>
<td>TID-5</td>
</tr>
<tr>
<td>5.</td>
<td>1</td>
<td>2</td>
<td>TID-1</td>
<td>TID-2</td>
</tr>
<tr>
<td>6.</td>
<td>1</td>
<td>2</td>
<td>TID-1</td>
<td>TID-5</td>
</tr>
<tr>
<td>7.</td>
<td>1</td>
<td>2</td>
<td>TID-2</td>
<td>TID-3</td>
</tr>
<tr>
<td>8.</td>
<td>1</td>
<td>2</td>
<td>TID-2</td>
<td>TID-5</td>
</tr>
<tr>
<td>9.</td>
<td>1</td>
<td>2</td>
<td>TID-3</td>
<td>TID-4</td>
</tr>
<tr>
<td>10.</td>
<td>1</td>
<td>2</td>
<td>TID-4</td>
<td>TID-5</td>
</tr>
</tbody>
</table>

**3.4.1.3 Result and Discussion**

The performance results of the DAGbM-KSJS model for the following metrics,

- Code coverage
- Number of test cases Vs. Number of clusters
- Failure rate
- Prioritization time
- Percentage of defects detected
3.4.1.3.1 Code Coverage

The code average is based on the number of statements that are traversed by a particular cluster. It is estimated for every test case executed in a test suite. The computation of the code coverage is based on the following equation:

\[
\text{Code coverage} = \frac{\text{Number of statements that are covered}}{\text{Total number of statements}} \times 100 \quad \ldots \quad (2)
\]

A comparison of code coverage for the prioritized and non-prioritized cases is represented in Table 3.4. From the table, it is analysed that the code coverage for the prioritized cases is more than the code coverage for the non-prioritized cases.

Table 3.4 Comparison of code Coverage for non-prioritized test cases and PDF based prioritized test cases

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Category</th>
<th>Code coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Non Prioritized</td>
<td>0.61</td>
</tr>
<tr>
<td>2.</td>
<td>Prioritized</td>
<td>0.69</td>
</tr>
</tbody>
</table>

3.4.1.3.2 Number of Test Cases vs. Number of Clusters

The comparison of the number of test cases executed with respect to the number of clusters is validated for the prioritized and non-prioritized approaches. From the comparison results represented in Fig. 3.3, it is clear that the prioritized approach executes more number of test cases than the non-prioritized approaches, the x axis represents number of clusters are executed from the both non prioritized and prioritized test cases and y axis represents the number of test cases are executed.
3.4.1.3.3 Failure Rate

The most precise source of data is the test of samples of the actual devices test cases for generating failure data. This is often affluent, so that the previous data sources are often used in its place. Firstly conduct the trials or analyse the data that how many times of test cases going to fail in a given number of tests. Then divide the number of failures by the total number of trials. Finally, multiply the previous step by 100 per cent to calculate the failure rate.

3.4.1.3.4 Fault Detection

The APFD estimates the rate of fault detection per percentage of test suite execution. It is estimated by considering a biased average of the percentage of faults detected by the test suite execution. Generally, the value of the APFD varies from 0 to 100. The higher the value of APFD, the better are the fault detection rates as represented in the below figure 3.4. In this figure x axis represents the test case
dataset, this graph shows there are four datasets, each dataset contains three types of data such as string based technique with pre-processing, string based technique without pre-processing and DAGbM-KSJS and the y-axis represents percentage of fault detection rate of test case dataset.

![Figure 3.4 Fault Detection](image)

Average percentage fault detection values are compared with some existing techniques such as Ant V6, Derby v1, Derby v2, and Derby v3, here the graph shows DAG-KSJS is higher rate compared with other techniques.

### 3.4.1.3.5 Prioritization Time

Prioritization time is defined as the average time consumed for performing the test case prioritization process. A comparison of prioritization time for the existing Greedy, ART, Genetic Algorithm (GA) algorithms and the proposed DAGbM is represented in Fig. 3.5. From the figure, shows that the suggested DAGbM algorithm using minimal prioritization time than the existing algorithms. Here the x-axis represents percentage of pools i.e. datasets utilized in particular period of time and y-axis represents the execution time of existing and proposed prioritized test cases.
3.4.1.3.6 Percentage of Defects Detected

The effectiveness of the proposed DAGbM algorithm is validated against the existing random order execution and weight factor based prioritization algorithm. The validation result is represented in Fig. 3.6. The figure helps analysis which shows the suggested DAGbM algorithm detecting higher number of defects in the execution of the test cases, when compared to the existing algorithms. The x axis represents percentage of test cases executed, it includes existing random based, weight factor based, and proposed DAGbM-KSJS test cases and y axis represents percentage of defects detected from existing and proposed DAGbM-KSJS prioritized test cases.
3.4.2 A Novel Prioritized Deciding Factor (PDF) Approach for DAG Based Test Case Prioritization Using Agile Testing Methodology

Agile methodology is mainly adopted for increasing software development, reducing documentation, and offering satisfaction to customers through continuous delivery of the product. Most of the software companies accept only agile based projects because the duration is smaller than seen in the waterfall model. Waterfall model takes a long time and changes affect the entire work- from the beginning till the end. However in agile method, changes does not affect at any level. In the agile methodology, user interaction is most important at all the levels to produce the right product. Customers’ requirements are added at any level instantly, and these requirements are created as use cases i.e. user board story; these use cases are split into many test cases.
3.4.2.1 Introduction

Agile testing involves all members of the cross functional agile team to ensure the achievement of business values desired by customer. It includes various methods, such as, Test Driven Development (TDD), Feature Driven Development (FDD), Extreme programming (XP), SCRUM, Crystal Clear (CC), Dynamic Systems Development Method (DSDM), and so on. The tester designs the test cases on the basis of the type of testing. The collection or set of many test cases is known as Test Suite. Therefore, during the testing phase, the tester can decide which test case is going to test first, then execute the software with those test cases, and then check and verify the results produced by the executions. Here a new method called Prioritized Deciding Factor (PDF) is introduced.

The main aim of this method is to prioritize the test cases based the Prioritized Deciding Factor (PDF) value, the Prioritized Deciding Factor (PDF) value calculated from the similarity and dependency values. Value created test cases are prioritized through this Prioritized Deciding Factor (PDF), i.e. if the Prioritized Deciding Factor (PDF) value is greater than 0, that value is taken up for consideration and corresponding test cases are selected for the prioritization process. If the Prioritized Deciding Factor (PDF) value is less than zero, it is not selected and the corresponding test cases are considered as fault nodes in the Directed Acyclic Graph (DAG) list.

3.4.2.2 Overall process of PDF Approach

Figure 3.7 represents the overall process of the Prioritized Deciding Factor(PDF) approach i.e. first of all we can take input from mobile functions as use cases; remove the irrelevant information through pre-processing; split the use cases
into number of test cases; find the similarity between the test cases; from the similarity value, distance value can be calculated through subtraction from 1; calculate dependency value from use cases; then, calculate Prioritized Deciding Factor (PDF) value from similarity value and dependency value through multiplication. If the Prioritized Deciding Factor (PDF) value is greater than or equal to zero, then those test cases are selected for the process of prioritization, unless the test cases are removed from the list. Finally, the selected test cases are prioritized by ranking of Prioritized Deciding Factor (PDF) value in ascending order.

Figure 3.7 Prioritization Deciding Factor (PDF) Approach
In this proposed work, there are a number of processes that would be performed step by step for the prioritization process. Prioritization is the main aim of this proposed work and the processes are as follows:

- Pre-processing
- Similarity value calculation
- Distance value calculation
- Dependency value calculation
- Prioritized Deciding Factor (PDF) Value calculation
- Prioritization of test cases
- Directed Acyclic Graph (DAG) representation

### 3.4.3 Prioritized Deciding Factor (PDF) Algorithm Approach

#### 3.4.3.1 Pre-processing

The initial process of this proposed work is pre-processing. This method is used for removing irrelevant information from the raw data. Here the inputs can be taken from the game applications. It is an unstructured data, i.e raw data. The raw data may contain more irrelevant information. From this raw data, we need to define the structured data, such as use cases. Use cases means user board story, which contains number of test cases. The table 3.6 shows the list of use cases.

#### Table 3.5 Lists of Use Cases

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Use cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Check for background music and sound effects #ON/OFF sound &amp; background music</td>
</tr>
<tr>
<td>2.</td>
<td>Check for background music and sound effects #Receive the call and check</td>
</tr>
<tr>
<td>3.</td>
<td>Check for background music and sound effects #Verify if sound effects are in sync with action</td>
</tr>
</tbody>
</table>
Table 3.6 represents use cases relating to mobile functions such as check for background music and sound effects, user interface and multimedia and so on. These are all considered as use cases. After pre-processing the use cases are split into number of test cases. Then all test cases and use cases are identified by unique numbers, such as 1, 2, and so on.

Table 3.6 Lists of Use Cases and Test Cases with Unique Identification

<table>
<thead>
<tr>
<th>S.No.</th>
<th>UID</th>
<th>Use Case</th>
<th>TID</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>Check for background music and sound</td>
<td>1</td>
<td>On/off sound and background music</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>Check for background music and sound</td>
<td>2</td>
<td>Receive the call and check</td>
</tr>
<tr>
<td>3.</td>
<td>1</td>
<td>Check for background music and sound</td>
<td>3</td>
<td>Verify if sound effects are in sync with action</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>Check for background music and sound</td>
<td>4</td>
<td>On/off device sound(native sound) and check</td>
</tr>
<tr>
<td>5.</td>
<td>1</td>
<td>Check for background music and sound</td>
<td>5</td>
<td>Check for vibration effect if present</td>
</tr>
</tbody>
</table>

Table 3.7 represents, use case 1, i.e. check for background music and sound effects is the first use case and it contains 5 test cases, i.e. on/off sound background, and receive the call and check, and so on. All the listed test cases are related to use case ‘check for background music and sound effects’ i.e. use case 1.
3.4.3.2 Similarity Value Calculation

Similarity is a complex concept which has been widely discussed in the linguistic, philosophical, and information theory communities. Similarity means a process for finding the same type of words, sentences, and so on. Two test cases are considered to illustrate calculation of similarity values. Each set is delimited by a separator like comma. Now each test case contains a number of words. The first test case is considered as $i$ and the second test case considered as $i+1$ or $j$, i.e., $i=1$ and $j=i+1$. Each use case contains $n$ number of test cases; use case 1 contain 5 test cases, and the first test case 1 is compared with test case 2 i.e. $1 \rightarrow 2, 1 \rightarrow 3$ up to $4 \rightarrow 5$. However, this is not necessary for the reverse process. Here, we use a method to find the similarity between the test cases named jaccard similarity and k-shingles. The formula for this method is

$$JS(T_1, T_2) = \frac{(T_1 \cap T_2)}{(T_1 \cup T_2)} \quad \ldots (3)$$

**Experiment 1:**

Test case 1=on/off, sound,&,background,music

Test case 2=receive, the, call, and, check

The Equation 1 Represented as

$$Sim(1,2)=\{\} / \{on/off,sound,&,background,music,verify,if,effects,are,in,sync\}$$

$$=0/11=0$$

**Experiment 2:**

Test case 1=on/off, sound,&,background,music

Test case 3= Verify if sound effects are in sync

$$Sim(1,3) = \{\text{sound}\} / \{on/off,sound,&,background,music,verify,if,effects,are,in,sync\}$$

$$=1/11=0.7143$$
3.4.3.3 Distance value Calculation

The distance value is used for finding the shortest distance between the nodes. The threshold of distance is 1. The distance is calculated from the similarity value i.e. similarity value subtracted by 1. That result value is considered as a distance value. The formula for the distance value calculation is

\[ \text{Distance value} = 1 - \text{Similarity value} \quad \text{.....(4)} \]

**Experiments:**

The Equation 2 follows as

\[ D_1 = 1 - 0 = 1 \]
\[ D_2 = 1 - 0.077 = 0.923 \]
\[ D_3 = 1 - 0.1 = 0.9 \]

Table 3.8 represents calculation of the similarity between test case 1 and test case 2, and similarity between test case 1 and test case 3 and so on. Then the distance value is calculated from the similarity value i.e. similarity value subtracted by 1. Here, the distance value is only used to represent Directed Acyclic Graph (DAG) for the valid test cases, and is not used for prioritization process.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Start Node</th>
<th>End Node</th>
<th>Similarity value</th>
<th>Distance value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>3</td>
<td>0.07143</td>
<td>0.928</td>
</tr>
<tr>
<td>3.</td>
<td>1</td>
<td>4</td>
<td>0.09091</td>
<td>0.9</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6.</td>
<td>2</td>
<td>4</td>
<td>0.18182</td>
<td>0.81818</td>
</tr>
<tr>
<td>7.</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8.</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9.</td>
<td>3</td>
<td>5</td>
<td>0.06667</td>
<td>0.9333</td>
</tr>
<tr>
<td>10.</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
3.4.3.4 Dependency value Calculation

The dependency value is calculated using the average value of the number of test cases. The purpose of dependency value is to prioritize the test cases based on the use cases. Each use cases contain number of test cases. These test cases are one to one interrelated. The formula for the dependency calculation is

\[
\sum_{i=1}^{n} UC_n = 1 / \sum_{i=1}^{n} TC_n
\]

….. (5)

Where UCn means Number of Use cases TCn means number of Test cases.

The Equation 3 represented as follows

Experiment 1:

UID-1=TID1+TID-2+TID-3+TID-4+TID-5=5

UID-1=\( \frac{5}{5} \)=0.25

Experiment 2:

UID-2=TID-6+TID-7+TID-8+TID-9+TID-10+TID-11+TID-12+TID-13+TID-14+
TID-15+TID-16+TID-17

UID-2=\( \frac{2}{2} \)=0.0833

Table 3.8 Dependency Values

<table>
<thead>
<tr>
<th>S.No.</th>
<th>UID</th>
<th>UID Count</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>5</td>
<td>0.20000</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>12</td>
<td>0.08333</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>2</td>
<td>0.50000</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td>6</td>
<td>0.16667</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>2</td>
<td>0.50000</td>
</tr>
<tr>
<td>6.</td>
<td>6</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>7.</td>
<td>7</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>8.</td>
<td>8</td>
<td>3</td>
<td>0.33333</td>
</tr>
<tr>
<td>9.</td>
<td>9</td>
<td>3</td>
<td>0.33333</td>
</tr>
<tr>
<td>10.</td>
<td>10</td>
<td>4</td>
<td>0.25000</td>
</tr>
</tbody>
</table>
Table 3.9 represents the dependency value of test cases. 1 divided by the number of test cases is considered as the dependency value of each use cases. The first uses case contains 5 test cases, and then the dependency value of use case 1 is 0.2 i.e. and so on.

3.4.3.5 Calculation of Prioritized Deciding Factor (PDF)

The Prioritized Deciding Factor (PDF) is a new approach for optimizing the prioritization process. Testing is an important portion in software development life cycle. The testing process takes half of the allotted time. The main aim of the Prioritized Deciding Factor (PDF) is to reduce the testing time for the testers. The algorithm for the Prioritized Deciding Factor (PDF) approach is

3.4.3.6 Algorithm-PDF

| Inputs:                                      |
| Computed Similarity, Distance & Dependency values |
| Output:                                      |
| Prioritized Test Cases                      |

Begin
For each x ∈ UC_i
    For all TC_m where m = 1, 2, ... N
        TList ← add (TC_i)
        For each T_i from TList_n
            Compute PDF (T_i, T_j)
            If (PDF(T_i, T_j) > 0)
                Add (T_i, T_j) → Res (UC_i)
            End
    End For
End For
For each (T_i, T_j) ← List(UC_i)
    Compute D_{Priso} = Pr (PDF(T_i, T_j))
    Result ← Produce (Pr (T_i, T_j))
End for
End
The formula of the Prioritized Deciding Factor (PDF) approach is

$$ Prioritized \ Deciding \ Factor (PDF) = \text{similarity value} \times \text{dependency value} \ldots (6) $$

The algorithm represents the use cases, which are separated from the test cases and added to the TList. Then Prioritized Deciding Factor (PDF) value is computed from similarity and dependency values. If Prioritized Deciding Factor (PDF) value is greater than zero, the test cases are selected to the prioritization list. Otherwise they are considered as false nodes in the Directed Acyclic Graph (DAG) representation and removed from the list. And then the duplicated nodes are removed from the list and provide rank for the selected Prioritized Deciding Factor (PDF) values. Ranking provided is based on the Prioritized Deciding Factor (PDF) value in ascending. After that, the test cases are prioritized based on the Prioritized Deciding Factor (PDF) value.

The Equation 4 represented as

**Experiment 1:**

Prioritized Deciding Factor (PDF) value = 0 × 0.20000

= 0.00000

**Experiment 2:**

Prioritized Deciding Factor (PDF) value

= 0.07143 × 0.08333

= 0.1429

Table 3.10 represents the calculation of Prioritized Deciding Factor (PDF) value, the Prioritized Deciding Factor (PDF) value is found from multiplication of the similarity value and the dependency value, A set of values are calculated from this
new process. Here the Similarity value of test case 1→2 is 0 because no words are match, and the dependency value of use case 1 is 0.2, then the multiplication of 0 *0.20 is 0 and so on. According to this process, the test cases are short listed, i.e., which Prioritized Deciding Factor (PDF) value is greater than or equal to zero, that value and test cases only selected and prioritized. Here we consider the first node i.e., test case 1→2 of Prioritized Deciding Factor(PDF) value is 0, so it is not selected for the prioritization process, likewise, whose value is less than zero that node should be removed from the list of test cases, which is considered as FALSE node.

### Table 3.9 Prioritized Deciding Factor (PDF) Value with Similarity, Dependency Values

<table>
<thead>
<tr>
<th>S.No.</th>
<th>UID</th>
<th>DAG</th>
<th>Similarity value</th>
<th>Dependency Value</th>
<th>PDF Value(Sim*Dep)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>1-2</td>
<td>0</td>
<td>0.20000</td>
<td>0.00000</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1-3</td>
<td>0.07143</td>
<td>0.08333</td>
<td>0.1429</td>
</tr>
<tr>
<td>3.</td>
<td>1</td>
<td>1-4</td>
<td>0.09091</td>
<td>0.50000</td>
<td>0.04545</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>1-5</td>
<td>0</td>
<td>0.16667</td>
<td>0.00000</td>
</tr>
<tr>
<td>5.</td>
<td>1</td>
<td>2-3</td>
<td>0</td>
<td>0.50000</td>
<td>0.00000</td>
</tr>
<tr>
<td>6.</td>
<td>1</td>
<td>2-4</td>
<td>0.18182</td>
<td>1.00000</td>
<td>0.18182</td>
</tr>
<tr>
<td>7.</td>
<td>1</td>
<td>2-5</td>
<td>0</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>8.</td>
<td>1</td>
<td>3-4</td>
<td>0.00000</td>
<td>0.33333</td>
<td>0.00000</td>
</tr>
<tr>
<td>9.</td>
<td>1</td>
<td>3-5</td>
<td>0.06667</td>
<td>0.33333</td>
<td>0.01333</td>
</tr>
<tr>
<td>10.</td>
<td>1</td>
<td>4-5</td>
<td>0.00000</td>
<td>0.25000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

#### 3.4.3.7 Prioritization of test cases

The test cases are prioritized on the basis of the Prioritized Deciding Factor (PDF) value. If the Prioritized Deciding Factor (PDF) value is >= 0 then the test cases are considered for the prioritization process otherwise it is removed from the list. First, there is need to merge the Prioritized Deciding Factor (PDF) value with valid
nodes. Then, prioritize the test cases on the basis of the prioritized deciding factor (PDF) value. In step 6 include the following process.

3.4.3.7.1 Duplicate removal

This process removes the replication of test cases. For example, the Prioritized Deciding Factor (PDF) value of $1 \rightarrow 3$ and $1 \rightarrow 4$ are 0.0077 and 0.1 respectively. Here 1 occurs twice. Any test cases can occur once in the list. According to this process, there are 54 test cases that are selected from the full list without duplication.

3.4.3.7.2 Ranking the test cases based on PDF value

The selected Prioritized Deciding Factor (PDF) value and corresponding test cases are ordered by ranking on the basis of Prioritized Deciding Factor (PDF) value from the highest to the smallest. In this method, the Prioritized Deciding Factor (PDF) value is not unique, i.e. the same value for more than test cases will create some confusion during the priority process.

3.4.3.7.3 Prioritization of test cases based on PDF value

Here, the test cases are ranked on the basis of Prioritized Deciding Factor (PDF) value. The same values are avoided up to the next round selection in the prioritization. Table 3.11 represents the prioritized test case based on the Prioritized Deciding Factor (PDF) value, if Prioritized Deciding Factor (PDF) is $\geq 0$ the test case is selected for prioritization. Otherwise the test case is removed from the list of test cases. A simple method is only applied. Here, the Prioritized Deciding Factor (PDF) value is ordered in ascending order. The test cases are arranged on the basis of the prioritized deciding factor (PDF) ascending.
Table 3.10 Prioritized Test Cases

<table>
<thead>
<tr>
<th>S. No.</th>
<th>PDF Ranking</th>
<th>UID</th>
<th>TID</th>
<th>PDF Value</th>
<th>Use Case</th>
<th>TCase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>2</td>
<td>17</td>
<td>0.01740</td>
<td>User Interface</td>
<td>Check other objects too (ex. if it’s a car race)</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>14</td>
<td>75</td>
<td>0.00175</td>
<td>Multiplayer game</td>
<td>Login with registered but</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>0.00267</td>
<td>User interface</td>
<td>Check in landscape portrait mode</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td>2</td>
<td>29</td>
<td>0.00286</td>
<td>User interface</td>
<td>Test whether one object overlaps with another</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>0.00308</td>
<td>User interface</td>
<td>Test whether one object overlaps with another</td>
</tr>
<tr>
<td>6.</td>
<td>6</td>
<td>2</td>
<td>10</td>
<td>0.00334</td>
<td>User interface</td>
<td>Verify if loading indicator is displayed wherever required</td>
</tr>
<tr>
<td>7.</td>
<td>7</td>
<td>2</td>
<td>12</td>
<td>0.00471</td>
<td>User interface</td>
<td>Test for enable and disable images, icon, buttons etc.,</td>
</tr>
<tr>
<td>8.</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>0.00534</td>
<td>User interface</td>
<td>There should not be any clipping (cutted background)</td>
</tr>
<tr>
<td>9.</td>
<td>9</td>
<td>2</td>
<td>13</td>
<td>0.00572</td>
<td>User interface</td>
<td>Check for screen title</td>
</tr>
<tr>
<td>10.</td>
<td>10</td>
<td>14</td>
<td>79</td>
<td>0.00637</td>
<td>Multiplayer game</td>
<td>Check user statistics graph</td>
</tr>
<tr>
<td>11.</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>0.00715</td>
<td>Check for background</td>
<td>Verify if sound effects are in sync with action</td>
</tr>
<tr>
<td>12.</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>0.00770</td>
<td>Check for background</td>
<td>On/off sound &amp; background music</td>
</tr>
<tr>
<td>13.</td>
<td>13</td>
<td>2</td>
<td>15</td>
<td>0.00800</td>
<td>User interface</td>
<td>Check scrolling</td>
</tr>
<tr>
<td>14.</td>
<td>14</td>
<td>3</td>
<td>18</td>
<td>0.01192</td>
<td>Performance</td>
<td>Check the loading time of a game</td>
</tr>
<tr>
<td>15.</td>
<td>15</td>
<td>6</td>
<td>45</td>
<td>0.01250</td>
<td>Time out</td>
<td>Check for time out</td>
</tr>
<tr>
<td>16.</td>
<td>16</td>
<td>5</td>
<td>41</td>
<td>0.01318</td>
<td>Score</td>
<td>Score calculation</td>
</tr>
<tr>
<td>17.</td>
<td>17</td>
<td>1</td>
<td>21</td>
<td>0.01667</td>
<td>Check for background music and sound effects</td>
<td>On/off sound &amp; background music</td>
</tr>
</tbody>
</table>

3.4.3.8 Directed Acyclic Graph (DAG) Representation

The Directed Acyclic Graph (DAG) is a graph that represents the distance between the nodes; the link follow from one node to another in forward. Here, the
backward process is not necessary in Directed Acyclic Graph (DAG) representation. Here, the test cases are nodes. The process can be started from the first test case; however, false nodes are not considered in the list. Use case 1 and Use case 2 valid nodes are shown in the below table 3.12

Table 3.11 Use Case 1 and Use Case 2 Valid Nodes

<table>
<thead>
<tr>
<th>S. No.</th>
<th>UID</th>
<th>DAG</th>
<th>Distance Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>1-3</td>
<td>0.928</td>
</tr>
<tr>
<td>2.</td>
<td>1</td>
<td>1-4</td>
<td>0.9</td>
</tr>
<tr>
<td>3.</td>
<td>1</td>
<td>2-4</td>
<td>0.81818</td>
</tr>
<tr>
<td>4.</td>
<td>1</td>
<td>3-5</td>
<td>0.93333</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
<td>6-7</td>
<td>0.93750</td>
</tr>
<tr>
<td>6.</td>
<td>2</td>
<td>6-13</td>
<td>0.8750</td>
</tr>
<tr>
<td>7.</td>
<td>2</td>
<td>6-14</td>
<td>0.92857</td>
</tr>
<tr>
<td>8.</td>
<td>2</td>
<td>6-15</td>
<td>0.83333</td>
</tr>
<tr>
<td>9.</td>
<td>2</td>
<td>6-17</td>
<td>0.96154</td>
</tr>
</tbody>
</table>

Following the PDF ranking based prioritization, least value test cases are removed from the list, the test cases are reduced from the total list i.e. 80 test cases in all are taken from the mobile application functions, but finally 54 cases only are selected as per the PDF algorithmic approach. Use case 1 contains 10 test cases but after the PDF process 4 test cases only selected in the below figure 3.8.

Fig 3.8 Directed Acyclic Graph (DAG) For Use Case 1
3.4.3.9 Result and Discussion

This section describes the performance analysis of the prioritized deciding factor (PDF) approach for the following metric. The implementation of the prioritized deciding factor (PDF) is done in the platform of PHP 5.5.12 and MYSQL5.6.17. The sample screens are

![Sample Screen](image)

Figure 3.9 Sample Screen of Proposed Work Prioritized Deciding Factor (PDF)

3.4.3.9.1 Fault Detection Using Average Percentage of Fault Detection

Average percentage of Fault detection (APFD) is a standardized metric that is used for finding the magnitude of faults detected. It depends upon the fault criterion considered. Average percentage of Fault detection (APFD) is computed to measure the rate fault detection of coverage based on prioritization techniques. The Average percentage of Fault detection (APFD) measures the weighted average of the percentage of faults detected over the life of a test suite. Average percentage of Fault detection (APFD) values ranges from 0 to 100, higher number simply faster faults detection rates are represented in the below figure 3.15.
Table 3.12 APFD calculation for Execution Time and Prioritized Deciding Factor (PDF) Value of Test Cases

<table>
<thead>
<tr>
<th>Faults/Test cases</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>No. of faults</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>PDF value</td>
<td>0.0077</td>
<td>0.02223</td>
<td>0.00715</td>
<td>0.00716</td>
<td>0.00715</td>
<td>0.00267</td>
<td>0.002668</td>
<td>0.005336</td>
<td>0.00308</td>
<td>0.003336</td>
</tr>
</tbody>
</table>

Average percentage of Fault detection (APFD)

\[ \text{APFD} = 1 - \frac{TF_1 + TF_2 + TF_3 + \ldots + TF_m}{nm} + \frac{1}{2n} \quad \ldots \ldots (7) \]

Generally test cases are executed in run time, during which many faults may occur, such as syntax error, technical error, input error, etc. Average percentage of Fault detection (APFD) method is used for locating the errors and finding the execution time for test cases. Table 3.15 represents the faults of each test case and Prioritized Deciding Factor (PDF) values for the Test cases, consider 10 test cases for fault detection, the following table shows the no of faults and the time taken for each test cases. Non prioritized test cases are in order among 80 first 10 test cases only considered T1,T2,T3,T4,T5,T6,T7,T8,T9,T10

Average percentage of Fault detection (APFD)

\[ \text{APFD} = \frac{1}{10} \left( 5 + 1 + 5 + 2 + 4 + 4 + 4 + 3 + 4 + 3 \right) + \frac{1}{2} \times 10 = 0.60 \]
Prioritized test case in the order among 80 test cases, first 10 only considered here T6, T9, T10 T8, T3, T1, T2, T7, T5, T4

Average percentage of Fault detection (APFD)

\[
\begin{align*}
&= 1-5+2+4+1+2+3+3+2+1/10*10+1/2*10 \\
&= 1-0.27+0.05 \\
&= 1-0.32 \\
&= 0.68
\end{align*}
\]

![Graph for Fault Detection Rate](image)

**Figure 3.10 Graph for Fault Detection Rate**

Figure 3.10 is a graphical representation of the fault detection rate, in x axis- percentage of fault detection, and in y axis- test case type i.e. prioritized and non-prioritized. According to the graph, prioritized test case fault detection rate is higher than non prioritized test case fault detection rate.

**3.4.3.9.2 Execution Time of Test Cases**

The execution time of test cases is calculated using the following formula

\[
T.E.T= E.T * \text{NO OF TEST CASES} * \text{PERIOD} / 8 \quad \text{................ (8)}
\]
E.T=3 milli seconds /TC

Number of TC=80

Period of ROI=3 months

\[=3\times80\times3/8\]

\[=90 \text{ ms}=0.09 \text{ seconds}\]

Compared with other existing algorithms, this method is better in time execution of test cases. It takes totally 0.09 seconds only to execute 80 test cases.

Figure 3.11 represents the execution time of test cases, Here, existing test cases takes 120 milli seconds for 80 test cases i.e. 57% and proposed 90 milli second is taken for 80 test cases i.e. 43% only, therefore the execution time of this proposed work is lesser than the existing work. In the above figure is pie chart, it represent y axis only, here y axis point out the percentage of execution time of all the test cases of both prioritized and existing non prioritized test cases.
3.5 SUMMARY

Two methods are implemented and compared with the first method DAGbM-KSJS algorithms and the second method PDF algorithm, execution time is reduced and fault detection also improves. Here the fault detection process is analyzed by Average Percentage fault Detection method, According to the final result, the PDF method performed more efficient than the DAGbM-KSJS-SCP method. The overall design of the Directed Acyclic Graph using prioritization techniques using three different algorithms is evaluated. The complete workflow of the proposed method is projected simultaneously.