Test Cases Prioritization using Open Dependency Structure Algorithm

Indumathi C P¹, Selvamani K²
¹Research Scholar, Department of CSE, Anna University, BIT Campus, Tiruchirappalli-620024, India
²Assistant Professor, Department of CSE, Anna University, CEG Campus, Chennai-600025, India.

Abstract

Test case prioritization is the process of ordering the test cases to be conducted eventually. Prioritizing test cases aids to meet two important constraints namely time and budget in software testing in order to enhance the fault detection rate as early as possible. This results us finding the problem earlier due to improved fault detection rate at early stage and deliver the system at short period of time. Interactions and relationships between a systems or sub system of an user is called as dependency. Few techniques that have been used in the past are namely Random order, greedy technique as well as fine grained and coarse grained. But all these techniques have considered the test cases in a test suite as an independent way. Few techniques have been derived dependency structures among test cases manually. Hence, this paper focuses to provide a set of algorithms to derive dependency structures among functions in a system and find exact number of dependents of each function in a system to prioritize the test cases in automated manner. High priority is assigned to test cases that are more dependent in using graph coverage values. By executing high priority (more complex) test cases earlier, the rate of fault detection will be improved at earlier stage and help to fix the fault earlier. For this experimentation purposes, Siemens test suite has been taken to establish the truth of verifying the given proposed technique.

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of scientific committee of International Conference on Computer, Communication and Convergence (ICCC 2015)
Keywords: Functional Dependancy; Testing; Debugging; Test Case Prioritization;
1. Introduction

In the past, there are many test cases that uses prioritization techniques [9], [4], [14], [13] for demonstrating and increasing the effectiveness of enhancing fault detection rate. Most of these techniques have been used as statement level and functional level prioritization techniques to prioritize the test cases in a test suite. They treated each test case as independent test cases and prioritize them but they have not considered the functional dependency among various test cases. Hence, this paper aims in dependency structures among test cases, which helps to improve the rate of fault detection in comparison with earlier systems.

Functional dependencies are the interactions among system functionality determining their execution sequence for example, a function X will be executed only if function Y has completed its execution. Hence test cases inherit these type dependencies. Therefore, some test cases have to be executed first before the execution of other test cases. A common way to handle test case dependency is to group the test cases into coarse-grained tests. For large test a suite in which prioritization is important, the testers want to avoid running redundant test cases.

2. Related Work

Many works pertaining to test case prioritization has been proposed and implemented by many researchers in the past. Some of the few important works has been cited in this section. Existing approaches that uses [6], [1] derived dependency structures among various test cases in a test suite is performed manually which consumes more time to test the job by a tester as well as tests the system time. Haidry’s technique is used [1] for finding total number of dependents for each test cases using DSP(Dependency Structure Prioritization) volume is not working properly.

Hence, in this research paper, two algorithms have been proposed. The first algorithm will extract the dependency structures among test cases automatically that helps to reduce the time taken to test the system and the second is level ordering for computing the exact number of dependents for each test cases.

2.1 Dependency Structures

This section explains the information about dependency structures and various definitions [1] about dependency structures.

A functional dependency is the one which has some interactions cannot occur until some other interactions occur first. Interaction I_b is dependent on interaction I_a if I_a has to be executed first.

A dependency structure is a directed acyclic graph (DAG), G={V,E}. In this paper, V- set of test cases, E - set of arcs connecting the nodes means functional dependency among the tests.

Open and Closed dependency structures: In this scenario consider a functional dependency between test cases T_a and T_b. In open dependency structures, T_a must be executed at any time but before T_b. For example, in fig. 2.1, if node T1 is executed, then node T3 and T5 are available. If node T2 is executed, then nodes T3, T4 and T5 are about to be executed. In closed dependency structures, T_b can be executed iff T_a has been executed before T_b. For e.g. In Fig 2.1, if nodes T1 and T2 are executed sequentially, then to execute node T3, node T1 must be executed again. Fig 2.1 shows that T3 is dependent upon T1 and T7 is directly dependent upon T3 whereas T7 is indirectly dependent upon T1.

Independent test cases are the test cases that execute the test case which is not dependent on any other test cases.

Dependent test case that execute the test case which is dependent on the execution of some other test cases.

Riser and Glinz [11] divide dependencies into three categories as Abstract dependency, temporal dependency, and Causal dependency (similar to our open dependency)
3. Prioritizing Test Cases Based On Dependency Structures

High coupling and more interactions among the parts of the subsystem that leads to more complexity. In this assumption is that, the dependency structure among test cases pertaining to the interactions among the parts of the system. Hence by testing more interactions subsystems in a system earlier may enhance the rate of fault detection.

In this section, presented a technique to derive dependency structures among test cases and prioritize them automatically.

3.1. Tool development

Previous existing system extracted dependency structures among test cases manually. This paper proposed an algorithm to extract dependency structures among functions in a system automatically.

**Algorithm 1 for adjacency matrix**

1. $fc$ – No.of function names in a system
2. Separate the corresponding function code (fcode) for each function.
3. for $i = 1$ to $fc$ { Take fcode,
4. Check if fcode, contains any other function names.
5. If so, store that function name(called function) under that calling function name.(farray-called function name list) }
6. Display all the function names in row wise and column wise order.
7. for $i=1$ to $fc$ { Take the function namei and its corresponding farrayi,
8. $fcounti$-number of function names farray,
9. for $j=1$ to $fcounti$
10. { Take function name, from the farrayi, in row order and assign ‘1’ for the corresponding calling function name in column order. }

This algorithm generates dependency structures among functions in a system as adjacency matrix form. If there is a dependency between two functions, then it will be represented by ‘1’ otherwise it will be represented by ‘0’.

3.2. Arrange in Level order

After extracting dependency structures manually, the existing system directly computes volume and height using adjacency matrix. Here introduced level order algorithm to the functions to be get a new arrangement adjacency matrix values. Here uses adjacency matrix as its input and produce the level wise arranged adjacency matrix.

**Algorithm 2 for level ordering arrangement**

1. Find all the independent functions in the system. T_ind[]
2. Find the length (len) of the array T_ind[]
3. for $i = 0$ to len do { for col =0 to len do {
4. if(adj[ T_ind[i][col] ]==1) { Check for redundancy in T_ind
5. Add ‘col’ value in T_ind } } //if
6. Update length of the array T_ind }
7. Rearrange the adjacency matrix values as per the order of function names in T_ind

This algorithm finds the order of functions to be displayed in row wise and column wise order and then rearranges the adjacency matrix values. Table 4.2 in the section of Result and discussion shows the result of the above algorithm for the program tot_info.c from Siemens test suite

3.3 Prioritization using open dependency structures

Based on open dependency structures, we have two measures: DSP (Dependency Structure Prioritization) volume, DSP height.

3.3.1. DSP Volume

This measure assigns higher weight to those test cases that have high number of dependents.

For computing the DSP volume of a test case, we should find all the dependents (direct and independent) of that test case. Algorithm 3 that computes all the dependents.

**Algorithm 3 for DSP Volume**
**Input:** Adj: an m x m adjacency matrix.

**Output:** adjacency: an m x m adjacency matrix.

1. Copy adj into adjacency
2. Take the first function name from the adjacency matrix
3. Check if any of the column contains “1”
4. If so, take that function name at that column
5. Do or operation between the current row and the row contains the name i.e., retrieved from the above steps.
6. Repeat steps 2 to 5 for all the functions.

For computing the weight of a function, counts number of ‘1’ s in each row.

### 3.3.2 DSP Height

This measure assigns higher weight to those test cases that have the longest path.

For computing the DSP Height of a test case, we should find the height of all paths from that test case.

Algorithm 4 computes the length from the test cases.

**Algorithm 4 for DSP Height**

**Input:** adj: an m x m adjacency matrix.

**Output:** adjacency: an m x m adjacency matrix which represents deepest dependent.

1. Copy adj into adjacency.
2. Take the first function name from row wise (i)
3. Take the first function name from column wise (i)
4. Find the adjacency [i, j] by the maximum between adjacency [i, j] and adjacency [i, k] + adjacency [k, j])
5. Repeat steps 2 to 4 for all the functions.

For computing the weight of a function, find the maximum value in each row.

### 3.3.3 Finding Prioritization

The below two algorithms are used to compute the prioritization of test cases in a test suite w.r.t to the calculation of weight based.

**Algorithm 5 for Weighted DFS (root level)**

**Input:** adj: an m x m adjacency matrix.

**Input:** wt: a weight function mapping test cases to their graph coverage values.

**Output:** a prioritized test suite.

1. Get independent tests of adj and store in IND
2. Sort all the independent tests using their wt.
3. Use DFS algorithm to prioritize and call function WFDS_visit for avoiding redundant addition of a test

**Algorithm 6 – WDFS_visit (root to leaf level)**

**Input:** an m x m adjacency matrix (adj), node to commence search process (node), a weight function mapping test cases to their graph coverage values (wt).

**Output:** a prioritized test suite.

1. Get all the children of first independent test in IND
2. Sort all the children of that independent test using their wt.
3. Use DFS algorithm prioritize the test cases under that independent test
4. Repeat steps 1 to 3 until IND is empty.

### 4 Results and Discussion

For evaluating the proposed technique was implemented with various suites of programs with various lines of code and with variable test pool size, seven programs from Siemens test suite has been considered. The objective of this technique is to whether prioritization based on dependency structure will improve the fault detection rate when compare with the existing techniques.

**Measure: APFD (Average Percentage of Fault Detection) measures the fastness of fault detected by a test suite.**

Higher value indicates faster fault detection rate when compared with early systems.

APFD as follows: Let T – ordered test suite that contains n test cases

TF\(i\) - first test of T that finds the fault \(i\)

\[
\text{APFD} (T) = 1 - \frac{(\sum_{i=1}^{m} \text{TF}_i) / nm}{n/2n}
\]
n – number of test cases in T, m – number of faults in a system

For implementing the proposed technique, seven programs from Siemens test suite has been considered. They are print_tokens and print_tokens2 are used as lexical analyzer, replace is used for pattern matching and replace a sting, schedule and schedule2 are for scheduling the jobs, tot_info is for giving statistical information for the given data and tec is used in aircraft. The Siemens researchers created test pools and faulty versions for each programs.

Table 4.1 shows the details needed for evaluating the technique from Siemens test suite.

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
<th>Test pool size</th>
<th>No.of versions</th>
<th>No.of functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>726</td>
<td>4130</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>570</td>
<td>4115</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>Replace</td>
<td>564</td>
<td>5542</td>
<td>32</td>
<td>21</td>
</tr>
<tr>
<td>Schedule</td>
<td>412</td>
<td>2650</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>schedule2</td>
<td>374</td>
<td>2710</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Tc1</td>
<td>173</td>
<td>1052</td>
<td>23</td>
<td>9</td>
</tr>
<tr>
<td>tot_info</td>
<td>565</td>
<td>1608</td>
<td>41</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.2 Level ordered adjacency matrix for “tot_info.c”

<table>
<thead>
<tr>
<th>Function name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcf(1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>gser(2)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>InfoTbl(3)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LGamma(4)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>main(5)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>QChiSq(6)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>QGamma(7)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig 4.1 Dependency graph and prioritization order for tot_info.c

The above algorithm that uses dependency matrix as its input and produce the level wise order arranged adjacency matrix. After extracting dependency structures manually, the existing system directly computes volume and height using dependency matrix. The functions are arranged here according to the level order in table 4.2 then it shows a better result with existing algorithm. Here for consideration the program “tot_info.c” the order of function as follows Infotbl, LGamma, Main, Gcf, Gser, QGamma, QchiSq before prioritization and by applying the above
algorithms the order(using adjacency matrix) could be rearranged like this for after prioritization LGamma,Gcf,QGamma, QchiSq, Main, Gser, Infotbl

Thus we obtained the Prioritization order as 2,4,6,7,3,5,1 for the program tot_info.c. In fig 4.1, nodes represent method number and edges represent the relationships between methods. Table 4.3 shows the comparative APFD result of our techniques with other prioritization techniques shows that our work gives better result for tcas.c program and same result for other programs.

<table>
<thead>
<tr>
<th>Program Name</th>
<th>APFD</th>
<th>DSP Volume(without level ordering)</th>
<th>DSP Volume(with level ordering)</th>
<th>DSP Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcas</td>
<td>52</td>
<td>56</td>
<td>52</td>
<td>58</td>
</tr>
<tr>
<td>tot_info</td>
<td>73</td>
<td>74</td>
<td>76</td>
<td>76</td>
</tr>
<tr>
<td>print_tokens</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>57</td>
<td>54</td>
<td>57</td>
<td>57</td>
</tr>
<tr>
<td>Schedule</td>
<td>55</td>
<td>43</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>schedule2</td>
<td>50</td>
<td>43</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>Replace</td>
<td>54</td>
<td>50</td>
<td>48</td>
<td>48</td>
</tr>
</tbody>
</table>

### 5. Conclusion

The proposed work was developed and implemented. The experimental results of the dependency structure algorithm for six programs from Siemens’s test suite indicates that the proposed technique provides a better solution to the test case prioritization problem among existing algorithms. In this work, we have taken all the measures based on open dependency structure algorithm. The advantage of proposed system extracts the dependency structure automatically among the test cases. This helps to prioritize the test cases within a very short period time, as well as to minimize the speed of test process and increase the rate of fault deduction at earlier stage and hereby reduce the time taken to deliver the system.

### REFERENCES