An Optimized technique for Test Case Generation and Prioritization Using “Tabu” Search and “Data Clustering”

Source available on DBLP and SCOUPS

Praveen Ranjan Srivastava¹, Aditya Vijay², Bhupesh Barukha²,
Prashant Singh Sengar², Rajat Sharma²

Computer Science & Information System Group, BITS PILANI – 333031
(INDIA)
{praveenrsrivastava, adityavijay88, bbarukha, prashant06457, rajat091}@gmail.com

Abstract: In practice, an available testing budget limits the number of test cases that can be executed over particular software. This paper presents a “Tabu” search algorithm for the automatic generation of software test cases and their prioritization through clustering technique of data mining. The developed test case generator has a cost function for intensifying the search and another for diversifying the search, used when the intensification is not successful. It also combines the use of memory with a backtracking process to avoid getting stuck in local minima. Test case prioritization technique schedules test cases in an order that increases their effectiveness in meeting some performance goal.

Keywords: Tabu Search; Dynamic test case generation; Test case prioritization; Clustering.

1 Introduction

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. A part of Software Engineering is to do Software Testing which consists of a set of activities conducted with the aim of finding errors in software. As the number of test cases needed for fully testing a software program is a huge number, therefore in practice, exhaustive testing is infeasible except for trivial cases. It has been estimated that software testing entails 50 percent of software development [1], thus, a subset of all
possible test cases has to be determined that satisfies a particular criteria. This cost can be significantly reduced with the automation of test generation [2]. Among the different approaches used for the automation of this process, we may distinguish between specification-oriented approaches (or black-box testing), which generate the test cases from the program specification, and implementation-oriented approaches (or white-box testing), which generate the test cases from the code of the program under test. Test cases have to be generated according to the test adequacy criterion [3], which ‘is considered to be a stopping rule that determines whether sufficient testing has been done and provides measurements of test quality’. Some of these criteria are the structural criteria that specify testing requirements in terms of the coverage of a particular set of elements of the program under test or its specification.

Previous work on automatic test generation for structural criteria can be divided into static methods [4] that generate the tests without executing the program under test, and dynamic methods [5] that carry out a direct search of the tests through the execution of the program, which has to be previously instrumented.

The present techniques are used to generate the test cases randomly and then select the best ones out of the randomly generated test cases hence the quality of test cases depends upon the quality of test cases randomly generated [4]. Once done with generation of test cases we have to do test case prioritization through other techniques which is additional burden.

Test case prioritization techniques schedule test cases in an order that increases their effectiveness in meeting some performance goals. One of the performance goals is the rate of fault detection which is a measure of how quickly faults are detected within the testing process; an improved rate of fault detection can provide faster feedback on the system under test, and let software engineers begin locating and correcting faults earlier than might otherwise be possible. Other goal is Statement Coverage (SC) that is percentage of the total statements present in the code which are covered by the test cases. We have used statement coverage as our performance metric [6]. We have used clustering technique [14] to prioritize the test cases generated by tabu search, the aim of clustering the test cases is to group similar test cases together and then chose optimal number of test cases from each group such that we get the total branch coverage in minimum number of test cases. In this paper we have aimed at merging two different techniques of test case generation and prioritization. We have used the dynamic technique of “Tabu” search method [7] for test case generation called as TCGen(test case generator) which generates the most suitable test cases only and has a provision for doing all the ground work necessary for test case prioritization. Like other dynamic test generators, TCGen also generates the test cases from an instrumented version of the program under test. But to do that it creates the control flow graph (Called CFG) from the source code under test and creates the instrumented source code from the CFG and the source code under test [8]. This generates the test cases and results are obtained from instrumented source code and CFG. Because our method of prioritization uses the path string that is automatically generated during “Tabu” search procedure they are closely integrated together and very little effort is to done in test case prioritization.
2 Related Work

There has been a lot of work done in the field of Test cases generation. The most recent dynamic methods for automatic test generation use the search techniques called ant colony optimization [9], genetic algorithms and simulated annealing where the testing problem is treated as a search or optimization problem [17]. One of these techniques, genetic algorithms [4], is the most widely used technique. Some other techniques that are used in test case generation are scenario based test case generation which uses UML diagrams to generate test cases [10]. Test case generation through “Tabu” search has also been done [8] where they modify the original “Tabu” search method to generate test cases. The existing test case prioritization (TCP) techniques include graph theory based prioritization [1], prioritization through clustering [2]. Besides these some heuristic approaches to prioritization of test cases also exists [3]. All these techniques aim at maximizing a performance goal which may be fault rate detection, branch coverage etc. In our approach the performance goal is dissimilar branch coverage, which is achieved by integrating “Tabu” search and TCP to work towards a common goal. Section 1 of the paper contains the general idea about the techniques used for test case generation and prioritization. Section 2 describes the approach we have followed and section 3 describes in detail how we have used general “Tabu” search algorithm for test case generation. The 4th section describes the test case prioritization and application of clustering in test case prioritization. Section 5 contains 2 case studies to explain in detail the method of test case prioritization. The subsequent section contains detailed analysis and conclusion of our work.

3 Our Approach

We have integrated test case generation with test case prioritization in such a way that most of the ground work required for test case prioritization is completed during test case generation without any additional efforts for prioritizing the test cases. In our approach we first do test case generation through “Tabu” search method where the “Tabu” search algorithm is modified to generate test data. The next section describes an intuitive approach to prioritize the test cases through clustering which aims at maximum branch coverage. Brief detail about the techniques used for test case generation and test case prioritization is given in next subsection.

3.1 “Tabu” Search and Data Mining (Clustering).

“Tabu” search technique is based on the use of historical information about a neighborhood search aimed at helping the search to overcome local optima. The general algorithm of “Tabu” search is based on that of the next k neighbors while maintaining memory that avoids repeating the search in the same area “Tabu” of the solution space [7]. Clustering is a data mining technique in which data is divided into groups of similar objects. Each group, called cluster, consists of objects that are similar to each other and dissimilar to the objects of other groups. Representing data by fewer clusters
necessarily loses certain fine details, but achieves simplification. One of the popular clustering algorithms of Data Mining is the ‘k-means clustering algorithm’. This algorithm assigns each point to a cluster whose center (centroid) is nearest. The center is the average of all the points in the cluster — that is, its coordinates are the arithmetic mean for each dimension separately over all the points in the cluster [14].

3.2 Tabu Search Method for Test Case Generation

“Tabu” search method is used for test case generation, this method proposed in [8] was restricted to just test case generation, and we have modified the proposed algorithm to include test case prioritization (TCP) also. While generating the test cases the algorithm does most of the ground work necessary for test case prioritization. Some terms used in subsequent sections are described below:

Subgoal node: The sub goal node (ns goal) is a node in Control Flow Graph (CFG). This node is to be triggered by TCGen in iteration.

Short term Tabu list (ST): This list is stored on the sub goal node so that we do not keep repeating the same set of test cases. Long term Tabu list (LT)-This list will store a few test for every node other than least cost test case so that we do not get stuck in local minima. This process is called backtracking.

Cost function: The cost of a test case C (n) to reach a node is directly proportional to the number of statements it executes.

MAXIT: Maximum number of test cases it executes.

Generation of neighboring candidates

TCGen generates 4*n neighboring candidates of the Current Solution (CS), n being the number of input variables of the program under test. In short, the technique consists in generating two near-neighbor tests and further two more neighboring test cases from the CS. That is to say, if the CS is (V_1, V_2... Vn) where V_1, V2....V_n are the attributes of a test which is acting as a current solution. TCGen maintains the same values for all V_k that satisfy k≠µ and generates four new values for according to the given four formulae in the neighborhood of the CS (i) $V_i = V_i + S(l)$ (ii) $V_i = V_i - S(l)$ (iii) $V_i = V_i + S(m)$ (iv) $V_i = V_i - S(m)$, where S(o) is a short step length and S(µ) is a long step length (o and µ are TCGen parameters). The values for S (o) and S (µ) are dependent on the type and range of the input variables and although they are fixed respectively to the o and µ values at the beginning of TCGen, they change during execution, taking into account the evaluation of the generated tests. In this way, it will be possible to carry out larger jumps when there are appropriate neighbors in the last iteration, and a very fine adjustment of the search when the neighbors do not improve the CS cost.

Method to generate Test cases

TCGen has the goal of covering all the branches of the program under test, i.e. to cover all the nodes of its control flow graph CFG (detailed in the previous section). TCGen generates a subgoal node and for that node it generates tests (partial solutions) and executes them as input for the program under test. A test case x is formed by a vector (tuple) with given values V_1, V_2... V_n and the input variable values as X_1, X_2,..., X_p. The set of values for a variable x_i is determined by its type
(integer, float or character). TCGen generates test cases based on the test case that is the Current Solution (CS). Initially, the Current Solution is a random test but later TCGen selects it according to which subgoal node has to be covered using the Current Solution, TCGen generates a set of neighboring test candidates. When a test case is generated, TCGen checks whether it is a “Tabu” test case. A test case is “Tabu” if it is stored in the TCGen memory. In short, TCGen has a memory formed by two “Tabu” lists: the short-term “Tabu” list (ST) and the long-term “Tabu” list (LT). If a generated test is not “Tabu”, the instrumented program under test is executed to check which branches (nodes) it has covered and the cost incurred by said test. However, if a generated test is “Tabu”, it will be rejected. During the search process, the best solutions found are stored together with their costs in the CFG. Thus, when an executed test has a lower cost in a CFG node than the current cost stored in that node, that test is stored as the best solution for that node. This process is repeated until all the nodes become the subgoal node hence the branches are covered with optimal number of test cases. Algorithm for generating test cases is given below.

\[
\text{Begin} \\
\begin{align*}
&\text{Initialize the subgoal node status of all nodes to false.} \\
&\text{Initialize Current Solution} \\
&\text{Calculate the cost of Current Solution} \\
&\text{Store Current Solution in CFG} \\
&\text{Add Current Solution, its cost and path string to Tabu list ST} \\
&\text{Select a subgoal node to be covered} \\
&\text{Do} \\
&\text{Calculate neighborhood candidates} \\
&\text{Calculate the cost of candidates: each non Tabu candidate is executed and their path string generated.} \\
&\text{for each candidate do} \\
&\text{if a candidate triggers a node whose subgoal status is false then change subgoal status to true and add the candidate in CFG in that node.} \\
&\text{End for} \\
&\text{if (sub goal node not covered) then Add Current Solution to Tabu list ST} \\
&\text{else Delete Tabu list ST} \\
&\text{Endif} \\
&\text{Select a subgoal node to be covered} \\
&\text{Select Current Solution using the CFG} \\
&\text{if (Current Solution is depleted) then} \\
&\text{Add Current Solution to Tabu list LT} \\
&\text{Apply a backtracking process: new Current Solution and maybe new subgoal node} \\
&\text{Endif} \\
&\text{while (some node has subgoal status as false AND number of iterations < MAXIT)} \\
&\text{End} \\
\end{align*}
\]

After executing this algorithm we get the test cases with their cost and the path string which is used later in the test case prioritization. The time complexity for
“Tabu” search method is $O(t \cdot h)$ where $t$ is the maximum number of attempts in finding the neighbors and $h$ is the height of CFG. This “Tabu” search pseudo code can be used to generate the path taken by a test case during its execution in form of path string. The test cases generated along with the path string is an integral part of test case prioritization and since no extra effort is needed for the generation of the path string hence the prioritization becomes very efficient and easy. Next section gives the detailed explanation of the procedure of the test case prioritization developed.

4 Test Case Prioritization (TCP)

In the previous section we have described the “Tabu” search algorithm which we have used to generate test cases for testing purpose. The test cases generated along with the path string forms the integral part of the test case prioritization algorithm developed by us. TCP algorithm arranges the developed test cases in the order of their importance in software testing. The main stress for TCP in our approach is to prioritize test cases based on their dissimilarity. This approach allows the tester to detect all the bugs and faults in the software under test using minimum number of test cases, thus minimizing the time and effort required in testing. This is being made possible by maximum branch coverage in least number of test cases generated. The test cases generated by “Tabu” search method acts as an input to this module. The test cases are prioritized by clustering, that is grouping similar test cases together. Clustering can be done in two ways:

4.1 By specifying the number of clusters (K-means Clustering). K-means clustering groups the test cases in $k$ groups based on their dissimilarity, where $k$ is the input number [15].

4.2 By specifying the dissimilarity measure (Dynamic Clustering). Dynamic clustering groups the test cases into optimal number of clusters based on the extent of dissimilarity between them. If the difference between two test cases is more than a given threshold then they are grouped into different clusters [16].

The important terms involved in the TCP algorithm developed are:

Path String - It is a binary string representing the path taken by the test case in the CFG. Every path in a CFG is denoted by a binary number string containing 0’s and 1’s. As the test case moves along the control flow graph bit 0 is assigned for left branch of the node and bit 1 for right branch of the node. Thus, every test case that follows some particular path in the CFG has a unique path string. We use this path string for clustering and prioritization of the test cases based on the weights assigned to each test case as defined below.

$$W_1 = \frac{\sum \text{(Number of test cases through a particular path)}}{\text{(Total number of test cases)}}$$
Calculated for every branch that is covered by the test case in the CFG. This weight function gives more priority to the link that has been traversed by less number of test cases.

\[ W_2 = \frac{\text{Number of statements executed}}{\text{Total number of statements}} \]

This weight function will give more priority to a test case which has executed more statements.

**Weight Function (WtFunc)**

\[ \text{WtFunc} = W_1 + W_2; \]

**Prioritization through clustering**

The Path string and weights of all the test cases are calculated using the above functions and their clustering is done using the techniques as specified above. After the test cases are clustered the algorithm applied for prioritization of test cases is shown below.

**Begin**
**While** all clusters are not empty
**For**
i equal 0 to i less than number of clusters
Choose the test case with highest weight from
Cluster i.

This prioritization scheme will result in quick and efficient prioritization of the test case generated. Prioritization can be done in \( O(n^2) \) time complexity where \( n \) is number of clusters in which the test cases are clustered.

So once the test cases are generated using the modified “Tabu” search algorithm as explained in the previous section, we can use the TCP algorithm to prioritize the test cases and thus making software testing procedure more effective and efficient. Next section describes the working of the above explained algorithms through two different case studies in which generation and prioritization of test cases is done using the “Tabu” search and clustering algorithm.

**5 Case Study**

In this section we have explained in detail two case studies, case (a) and case (b). We have demonstrated the working of “Tabu” search algorithm for test case generation and TCP algorithm for prioritizing the test cases.

**Case a:**
Consider the code fragment shown below, which takes five numbers as input and then gives the highest number of the numbers.

```c
1) int main()
2) {
3) int a,b,c,d,e;
4) int greatest;
```
5) if(b>a)  
6) {  
7) greatest = b;  
8) }  
9) else  
10) {  
11) greatest = a;  
12) }  
13) if(c>greatest)  
14) {  
15) greatest = c;  
16) }  
17) if(d>greatest)  
18) {  
19) greatest = d;  
20) }  
21) if(e>greatest)  
22) {  
23) greatest = e;  
24) }  
25) return greatest;  
26) }

We first make the control flow graph of the code fragment. Each node of the control flow graph represents number of lines present in the code. Every logical statement in the program leads to branching in the CFG. Thus nodes are obtained in the CFG equal to the number of logical statements present in the code fragment. Control flow graph for the above example code is shown in figure 1.1. The statements present in each node are indicated near the node of the CFG. Now the “Tabu” Search technique gives the test cases which cover all the branches of the CFG. We generate the random test case and then modify the test case to generate the other neighboring test cases. We take the value of $\omega = 10$ $\mu = 25$ and calculate the value of other test cases using the above mentioned procedure. We assume that the domain of all the values is between [0,100].
1) Let the first test case be-
   a = 15, b = 20, c = 35, d = 50, e = 60.
   The path string obtained by the control flow graph is ABDEFGHIJ and corresponding binary string is 00000000. [W1 = 4.33, W2=0.9 W=5.23]
   The test cases generated by changing the value of ‘a’ we get following test cases:

(i) a = 25, b = 20, c = 35, d = 50, e = 60. (Triggers the subgoal node C.)
(ii) a = 5, b = 20, c = 35, d = 50, e = 60. (redundant because we get the same CFG as in the first test case.)
(iii) a = 40, b = 20, c = 35, d = 50, e = 60. (triggers the subgoal node C.)
(iv) a = -10, b = 20, c = 35, d = 50, e = 60. (rejected as the value of ‘a’ is negative.)
So now we consider the test cases ‘(i)’ and ‘(iii)’ as they trigger the subgoal node C for the next test case. We take ‘(i)’ as the next case.

1)  

\[ a = 25, \ b = 20, \ c = 35, \ d = 50, \ e = 60 \]  
The path string obtained by the control flow graph is ACDEFGHIJ and corresponding binary string is 10000000. [W1 = 4.33, W2=0.9, W=5.23]  
The test cases generated by changing the value of ‘b’ we get following test cases-

(i)  
\[ a = 25, \ b = 10, \ c = 35, \ d = 50, \ e = 60. \]  
(redundant because it has the same path covered as done by test case 2.)

(ii)  
\[ a = 25, \ b = 30, \ c = 35, \ d = 50, \ e = 60. \]  
(redundant because it covers the same path as 1.)

(iii)  
\[ a = 25, \ b = 45, \ c = 35, \ d = 50, \ e = 60. \]

(iv)  
\[ a = 25, \ b = -5, \ c = 35, \ d = 50, \ e = 60. \]  
(redundant because it has the negative value of b.)

Now we have (iii) as the new path covering test case.

2)  

\[ a = 25, \ b = 45, \ c = 35, \ d = 50, \ e = 60. \]  
The path string obtained by the control flow graph is ABDFGHIJ and corresponding binary string is 0010000. [W1 = 3.83, W2=0.8, W=4.63]  
The test cases generated by changing the value of ‘c’ we get following test cases-

(i)  
\[ a = 25, \ b = 45, \ c = 25, \ d = 50, \ e = 60. \]  
(redundant as it covers the same path as test case 3.)

(ii)  
\[ a = 25, \ b = 45, \ c = 45, \ d = 50, \ e = 60. \]  
(redundant as it covers the same path as test case 3.)

(iii)  
\[ a = 25, \ b = 45, \ c = 10, \ d = 50, \ e = 60. \]  
(redundant as it covers the same path as test case 3.)

(iv)  
\[ a = 25, \ b = 45, \ c = 60, \ d = 50, \ e = 60. \]

Now we take the (iv) as the next test case in consideration. Applying the similar approach for the next test cases we obtain the following test cases.

3)  

\[ a = 25, \ b = 45, \ c = 60, \ d = 50, \ e = 60. \]  
The path string obtained by the control flow graph is ABDFHJ and corresponding binary string is 000011. [W1 = 2.83, W2=0.7, W=3.53].

4)  

\[ a = 40, \ b = 20, \ c = 35, \ d = 50, \ e = 60. \]  
The path string obtained by the control flow graph is ACDFGHIJ and corresponding binary string is 1010000. [W1 = 3.33, W2= 0.7, W=4.03].
5) \( a = 60, \ b = 20, \ c = 40, \ d = 30, \ e = 10. \) The path string obtained by the control flow graph is ACDFHJ and corresponding binary string is 10111. [W1 = 2.33, W2= 0.6, W= 2.93].

Now we cluster the above test cases using k-means clustering where \( k = 2. \) We select the test cases on the basis of weights. The highest weight test case is obtained from each cluster and thus we obtain the prioritized test cases in decreasing order of

![Figure 1.2 Clusters for the test cases in the case study ‘a’.

Priority. In case of same priority test case we randomly select one of the clashing test cases. Thus the prioritized test cases for the above example are TC2 > TC1 > TC3 > TC5 > TC4 > TC6.

**Case b:**

Another example takes into account a code fragment which is a function to find the least common multiplier (LCM) of two input numbers.

1) \( \text{Int LCM(int a, int b)} \)
2) \{ \)
3) \( \text{int a;} \)
4) \( \text{if(a>b)} \)
5) \{ \)
6) \( \text{temp = a;} \)
7) \( \text{a = b;} \)
8) \( \text{b = temp;} \)
9) \} \)
10) \( \text{if(b==a)} \)
11) \} \)
12) \( \text{return b;} \)
13) \}
14) \}
15) \{
16) \( \text{for(i=b ; i < b*a ; i++)} \)
17) \{
18) \( \text{if(i\%b == 0 && i\%a ==0)} \)
19) \{
20) \( \text{break;} \)
21) \}
22) \}
23) \( \text{return i;} \)
24) \} \)
CFG for the above example code is obtained using the same method as explained for the case study ‘a’. CFG for case study ‘b’ is shown in figure 2.1. “Tabu” search technique gives us the test cases which cover all the branches of the control flow graph. Proceeding in the same manner as previous example we take the value of $\omega = 1$ and $\mu = 4$ and calculate the value of other test cases using the above mentioned procedure. We assume that the domain of the values is between $[0, 10]$ in this case.

Figure 2.1 CFG for the code fragment in the case study ‘b’.

1) Taking initial test case as—
   \[ a = 5, \ b = 10 \]
   The path string obtained by the control flow graph is ACEFHIJ and corresponding binary string is 110100. [W1 = 3.8, W2 = 0.7, W = 4.5]
   The cases generated by changing value of ‘a’ are-
   (i) $a=6, b=10$. (subgoal node G is triggered.)
   (ii) $a=4, b=10$. (redundant since no subgoal node triggered.)
   (iii) $a=9, b=10$. (redundant case.)
(iv) \(a=1, b=10\). (Useful since it covers a different path.)

Thus, only (i) and (iv) can be considered and we take up the cases as the next case.

1) \(a = 6, b = 10\).

The path string obtained by the control flow graph is ACEFGFHIJ and corresponding binary string is 0100100. \([W1 = 3.6, W2 = 0.7, W = 4.3]\)

Changing values of ‘b’ in this case:
(i) \(a=6, b=11\). (redundant case again.)
(ii) \(a=6, b=9\). (redundant case as it does not covers any subgoal node.)
(iii) \(a=6, b=6\). (it is a useful test case as it covers node D.)
(iv) \(a=6, b=14\). (redundant test case.)

Next, we consider case (iii).

2) \(a=6, b=6\).

The path string obtained by the control flow graph is ACDJ and corresponding binary string is 100.

\([W1 = 1.0, W2 = 0.4, W = 1.4]\)

Next, we consider case (iv) of the modification from Case 1.

3) \(a=1, b=10\).

The path string obtained by the control flow graph is ACEFGIJ and corresponding binary string is 110000. \([W1 = 3.8, W2 = 0.6, W = 4.4]\)

Proceeding in the same fashion by getting neighboring cases, we obtain the following test case also which completes this generation study.

4) \(a=10, b=5\).

The path string obtained by the control flow graph is ABCEFHIJ and corresponding binary string is 0010100. \([W1 = 3.6, W2 = 0.8, W = 4.4]\)

Now we cluster the above obtained test cases using k-means clustering where \(k = 3\). Now on the basis of weights of the test cases we select the test case with the highest
weight from each cluster and thus obtain the prioritized test cases in decreasing order of priority.

![Figure 2.2 Clusters for the test cases in the case study ‘b’.

In case of same priority test case we randomly select one of the clashing test cases. Thus the prioritized test cases for the above example are TC1 > TC5 > TC2 > TC3 > TC4.

6 Analysis

Our research work presents a new approach combining the fields of data mining and software testing. Test Case Generation is achieved using the well established technique of “Tabu” search. “Tabu” search is used in this case because the Control Flow Graph is required in test case prioritization used further in our work. The use of “Tabu” Search greatly reduces the work in test case prioritization. After the test cases are generated, the algorithm for test case prioritization takes these test cases as its inputs and prioritizes them. Our analysis of the test case prioritization algorithm shows that the algorithm computes the priority in O (n) time where ‘n’ is the height of the control flow graph. The prioritization technique mainly uses clustering, which is a very efficient method for quickly organizing the test cases into “Congruent Modules” from where test cases can be quickly extracted. The extraction of test cases from clusters is also done based on very simple measures like the number of nodes covered by the test case and the number of statements present in each node of the CFG. The information about the number of statements is present in each node. Analyzing the case study described above, it shows that the test cases higher on the priority scale have a higher chance of detecting faults in the code segment. Thus, we realize the main aim of test case prioritization- increase the rate of fault detection in
lesser time. Also, it can be seen that the test cases clustered are similar and so it validates the clustering approach used here. Giving a complete picture of the algorithm, the techniques used for generation and prioritization complement themselves. Also, the use of clustering makes it easier for the tester when the number of test cases is large. The algorithm is open to improvement in terms of code optimization and also to refinement in some areas. For example, more efficient techniques can be used for clustering and it will improve the performance of the algorithm. This new approach provides a new and radical way for handling the problem of increasing fault rate detection. “Tabu” search is also an important part of the whole algorithm and though other case generation techniques may be used, Tabu search technique helps in prioritization by generating the control flow graph. Thus, the algorithm is easily implementable and is open for further optimization. The binary string generated also does not take time to parse. Other prioritization techniques usually take $O(n \log n)$ time where $n$ is the number of branches. This analysis shows that our algorithm is at par with other prioritization algorithms in the effectiveness scale.

7 Conclusion

Giving a complete picture of the algorithm, the techniques used for test case generation and prioritization complement themselves providing a unified model for testing software. The designed algorithm gives new approach which has generation and prioritization embedded. Also, the use of clustering makes it easier for the tester when the number of test cases is large and the use of binary string makes it computationally more efficient. The algorithm is open to improvement in terms of code optimization and also to refinement in some areas. For example, more efficient techniques can be used for clustering and which in turn will improve the performance of the algorithm. This new approach provides a new and radical way for handling the problem of increasing fault rate detection. “Tabu” search is also an important part of the whole algorithm and though other case generation techniques may be used, Tabu search technique helps in prioritization by generating the control flow graph. Thus, the algorithm is easily implementable and is open for further optimization.
References


