Test case prioritization using Cuscuta search

Mukesh Mann, Om Prakash Sangwan
School of ICT, Gautam Buddha University, Greater Noida, 201308, India
E-mail: Mukesh.gbu@gmail.com

Received 19 September 2014; Accepted 6 October 2014; Published online 1 December 2014

Abstract
Most companies are under heavy time and resource constraints when it comes to testing a software system. Test prioritization technique(s) allows the most useful tests to be executed first, exposing faults earlier in the testing process. Thus makes software testing more efficient and cost effective by covering maximum faults in minimum time. But test case prioritization is not an easy and straightforward process and it requires huge efforts and time. Number of approaches is available with their proclaimed advantages and limitations, but accessibility of any one of them is a subject dependent. In this paper, artificial Cuscuta search algorithm (CSA) inspired by real Cuscuta parasitism is used to solve time constraint prioritization problem. We have applied CSA for prioritizing test cases in an order of maximum fault coverage with minimum test suite execution and compare its effectiveness with different prioritization ordering. Taking into account the experimental results, we conclude that (i) The average percentage of faults detection (APFD) is 82.5% using our proposed CSA ordering which is equal to the APFD of optimal and ant colony based ordering whereas No ordering, Random ordering and Reverse ordering has 76.25%, 75%, 68.75% of APFD respectively.

Keywords Dodder (Cuscuta sp.); prioritization; Cuscuta Search Algorithm (CSA); Ant Colony Optimization (ACO).

1 Introduction
As specified by G.J. Mayers (1997) “Testing is the process of executing a program with the intent of finding faults”. It focuses on the process of testing the newly developed / under development software system, prior to its use. Regression testing is primarily a maintenance activity that is performed frequently to ensure the validity of the modified software (Singh et al., 2010) and due to time and cost constraints, the entire test suite during regression testing cannot be run. Thus, it becomes essential to prioritize the tests in order to cover maximum faults in minimum time. Graphical User Interface (GUI) Test case prioritization was proposed (Sangwan et al., 2012) using fuzzy logic model by assigning weight value on the basis of multiple factors such as type of event, event interaction and Count as one of the criteria for test case prioritization for GUI based...
software. Apart from Fuzzy logic, Ant colony optimization is used as a new way to solve time constraint prioritization problem. The paper (Singh et al., 2010) presents the regression test prioritization technique to reorder test suites in time constraint environment along with an algorithm that implements the technique.

In the paper of Malhotra et al. (2010), regression testing is defined as the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications. A regression test selection technique selects an appropriate number of test cases from a test suite that might expose a fault in the modified program. The technique uses two algorithms one for “modification” and the other for “deletion.” The “modification” portion of the technique is used to minimize and prioritize test cases based on the modified lines of source code. The “deletion” portion of the technique is used to (i) update the execution history of test cases by removing the deleted lines of source code (ii) identify and remove those test cases that cover only those lines which are covered by other test cases of the program. Thus Regression testing is a very costly process performed primarily as a software maintenance activity.

During past few years Ant Colonies (AC’s) have been used as a general purpose heuristics to solve combinatorial optimization problem like classic travelling salesman problem, data mining, telecommunication networks, vehicle routing (Ayari et al., 2007; Caro et al., 1998; Dorigo et al., 1996; Gomez et al., 2005; Huaizhong et al., 2005; Li et al., 2008; Parpinelli et al., 2002; Zhao et al., 2006; Zhang, 2013a, 2013b). Srivastava et al. (2009) presented a simple and novel algorithm with the help of an ant colony optimization for the optimal path identification and prioritization by using the basic property and behavior of the ants. This novel approach uses certain set of rules to find out all the effective/optimal paths via ant colony optimization (ACO) principle. The method concentrates on generation of paths, equal to the cyclometric complexity. This algorithm guarantees Full path coverage and used an ACO technique to generate the optimal path suite and prioritize it according to path’s pheromone strength deposited by artificial ants. Control Flow Graph (CFG) diagram (Mathur, 2007) is used to generate optimal path. The benefit of approach (Srivastava et al., 2009) is that the manual generated test prioritized paths are not always reliable while automatic test prioritized paths are reliable, because humans are the most dynamic and error introducing entity.

Krishnamoorthi et al. (2009) focused on test case prioritization. The authors proposed a new test case prioritization technique using Genetic Algorithm (GA). The proposed technique prioritizes subsequences of the original test suite so that the new suite, which is run within a time-constrained execution environment. A superior rate of fault detection when compared to rates of randomly prioritized test suites has been achieved. Test case prioritization techniques schedule test cases in an execution order according to some criterion. The purpose of this prioritization is to increase the likelihood that if the test cases are used for regression testing in the given order, they will more closely meet some objective than they would if they were executed in some other order.

2 Cuscuta Search: A plant Intelligence

What is intelligence? There has been a long debate to find the definition of intelligence which is still in its premature stage. Some scholars define it as the ability to learn in complex situations, to make thought and reason, to bring out profit from experience. Intelligence is more than memory or learning and one definition (Stenhouse, 1974) defines intelligence as, “Adaptively variable behavior during the lifetime of an individual”.

Till now we have strong observations and formulation about the animal’s intelligent behavior such as ant colony, bee colony (Tereshko, 2000; Zhang, 2013a, 2013b). They involve foraging for food not by simple but by collective intelligence behavior. The plant’s foraging has been the least studied area in computational intelligence. Not only animals as described above forage for food intelligently but the same have been done by the plants too.
One such example is the dodder (Cuscuta sp.), a parasitic plant (Fig. 1) which attack it by prospective host through some host-plant clue. If the host is found unsuitable the Cuscuta sp. continue its search but once selection is made the Cuscuta sp. coil around its selected host in a specific manner (anticlockwise) to transfer resources from the host plant.

A recent study (Runyon et al., 2006) has reported that seedlings of C. pentagona (Cuscuta) use host-plant volatiles to guide host location and selection which was assumed a random (Dawson et al., 1994) phenomena before. The seedlings of C. pentagona orient their growth to various light cues associated with the presence of host plants (Benvenuti et al., 2005).

Research (Runyon et al., 2009) has found that Cuscuta sp. seedlings show directed growth toward tomato volatiles experimentally released in the absence of any other plant-derived cues. Furthermore, volatile cues are used by the seedlings to “choose” tomatoes, a preferred host, over non host wheat. This is because several individual compounds from the tomato volatile blend were attractive to Cuscuta sp. seedlings but out of this blend, three compounds individually elicit directed growth of Cuscuta: (A) β-phellandrene, (B) β-myrcene, and (C) α-pinene (Runyon et al., 2006) while one compound from the wheat blend, (Z)-3-hexenyl acetate, had a repellent effect. A typical attack of Cuscuta is shown in Fig. 1.

Fig. 1 A Dodder (Cuscuta sp.) (Light yellow) coiling around its host. Cuscuta sp. has the ability to assesses its prospective host before coiling around the host plant (Kelly, 1990) and thus it does not coil around every host with which it comes in contact. If the prospective host is found to be unsuitable the parasitic plant continues its search for other hosts. Photo by Mukesh Mann and Om Prakash Sangwan at Gautam Buddha University, Greater Noida, India; http://www.gbu.ac.in.

A key point of observation is that Cuscuta somehow knows its starvation i.e. if the same cues (α-pinene, β-myrcene, and β-phellandrene) would have been coming from the wheat, the bend will be towards the wheat rather than tomato. Considering this dynamics we can say Cuscuta search for its food from its current need.
(starvation) and will continue to attack till its starvation get complete. As soon as its starvation is completed a new branch will evolve. The evolution of new branch is considered as the completion of search i.e. no left starvation. The new branch will again repeat the same process until all plants nutrients are been taken by Cuscuta i.e. at short of dead host.

3 Modeling Test Case Prioritization Using Cuscuta Foraging
In order to model the intelligent behavior of Cuscuta we make the following assumption.
1. The Cuscuta knows its initial starvation.
2. The host is chosen which fulfills its maximum starvation from its current starvation.
3. At each attack the Cuscuta need for nutrients get fulfilled and the next attack is totally governed by the left starvation.
4. A new leave will grow as soon as the starvation get completed, this indicate the completion of one iteration / stopping criteria.

4 Defining Software Test Prioritization Problem
Prioritization is the process of scheduling test cases in an order to meet some performance goal. We define a test suite T as a tuple of test cases Ti from i=1 to n as (T1, T2,……,Tn). The goal is to execute Ti in order to meet some performance goal. With Knapsack problem, the minimum time in which we can prioritize the test cases is the maximum output of knapsack, i.e. Test cases are knapsack items, having total maximum capacity equal to total number of faults to be covered. The numbers of faults covered by each test case represent its weight and the total time to execute a test case to find the particular number of faults represent the time to put the item (test case) into the knapsack. The knapsack 0/1 algorithm outputs prioritized list in minimum ejection time.

Formally, 0/1 knapsack in terms of test suite prioritization is defined as (Alspaugh et. al., 2007)
Maximize: \( c_i x_i \)
Subject to: \( \min (t_i x_i) , x_i = 0 \) or 1
where, \( c_i \) is fault coverage, \( t_i \) is execution time of test case Ti. Thus, the 0/1 knapsack problem is an NP-complete problem (Rothermel et. al., 2001). All NP complete problems are NP hard. In this paper we use Cuscuta search method for solving this hard combinatorial optimization problem.

5 Test Suite Selection and Prioritization using Cuscuta Search
For a given test suite, the problem of selection and prioritization of test cases can be stated as follows:-
1. Given \( T = t_1, t_2, t_3, \ldots, t_n \) where \( T \) is original test suite.
2. Obtain \( m \in T \) such that \( m \leq n \) where \( m = \) number of test cases in test suite \( T \) and
3. Select \( m \) and prioritize them on the basis of maximum fault coverage in minimum time.

6 Proposed Cuscuta Search Algorithm (CSA)
1. Count total number of faults (TS) in given test prioritization problem.
2. Initialize position of each fault suite (fs) as the chemical clue randomly.
3. Place each test case (Cuscuta) [T1, T2, T3, …., TN] at the position corresponds to the position of fs
4. For each Cuscuta Ti , where \( i \in [1 \text{ to } n] \)
a) Starvation $ST_i = TS$. /* initialize the current starvation

b) Initial position $PT_i = f_{si}$. /* initialize position of current test case (Cuscuta) equal to position of its corresponding
/ * fault suite ($f_{si}$).

5. WHILE ($ST_i != 0$) {
   a) Current starvation $CS_i = ST_i$

6. For each Cuscuta $T_j$ where $j \in [1$ to $n]$ {
   /* find test case (Cuscuta) out of all available test case which fulfills maximum starvation of the current test suite as per the
   need of its current starvation.
   a) Find $T_j$ such that $T_j = \max (fs)$ as per $CS_i$.
      /* Replace current starvation with the test suite which provide maximum fault as per current starvation of Cuscuta.
   } /* end of second for loop
   b) Seq_array[n] = $T_j$ /* Store value of test case in an array of size n.
   c) Find $f_s$ corresponding to $T_j$.
      a) $ST_i = \text{sub}(CS_i, f_s)$ /* calculate left starvation

7. Find execution time associated with the $T_j = \text{exc}_\text{Time}[T_j]$. 
8. Total time[i] = total time[i] + $\text{exc}_\text{Time}[T_j]$.
   } /*end of while statement
   Print Total time[i].
   Print Seq_array[n].
   } /* end of first for loop

9. Find sort_ascend (Total time[i]) /*Arrange Test cases in order of increasing execution time
10. Print “The Prioritization order will be the corresponding array index Seq_array[n] in order of sort_ascend
    Total time[i]”.

7 Evaluation Metrics
In order to evaluate the performance of various test case prioritization schemes, prior knowledge of faults
within the given program is assumed along with execution time to run the test cases as shown in Table no.1
and Table no.2. Test suite can be evaluated empirically based on average percentage of fault detected (APFD,
for short) over the life time of the test suite. A higher preference will be given to the prioritization scheme
having higher APFD value. APFD (Krishnamoorthi et al., 2009) is defined as

$$\text{APFD} = \left[ 1 - \Sigma_{i=1}^{g} \frac{\text{reveal}(i, T)}{ng} \right] + \frac{1}{2n}$$

where, $T =$ test suite, $g =$ number of faults in program under test, $n =$ number of test cases, $\text{reveal}(i, T) =$
position of the first test in $T$ that exposes fault $i$.

Other method to calculate APFD is to find the area under the curve that represents the weighted percentage
of faults undetected over the corresponding fraction of the test suite (Gregg et al., 1999).
8 Example Validation
Consider a test suite with 8 test cases in it, covering a total of 10 faults (Singh et. al., 2010) and given their total time of execution as shown in Table no. 1 and Table no. 2. Our task is to prioritize these test cases in an order of maximum fault coverage with minimum test suite execution. The following mapping is considered between the proposed algorithm’s variables and given prioritization problem.

1. We assume that we had prior information about the original test suite \( T = \{t_1,t_2\ldots\ldots t_n\} \) and corresponding fault coverage (Yogesh Singh et. al., 2010) as shown in table 1 and the total execution time of each test case as shown in Table 2.

2. Number of Cuscuta plants search for food is equal to number of test cases.

3. The host is chosen which fulfills its maximum starvation from its current starvation.

4. At each attack the Cuscuta need for nutrients get fulfilled and the next attack is totally governed by the left starvation.

5. A new branch will grow as soon as the starvation get completed, this indicate the competitions of one iteration/stopping criteria.

<table>
<thead>
<tr>
<th>Test case/faults</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>T2</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>T4</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>*</td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>T8</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

**Table 1** Sample test cases vs. faults identified.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Fault suite</th>
<th>No. of faults covered</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>f1</td>
<td>2,4,7,9</td>
<td>7</td>
</tr>
<tr>
<td>T2</td>
<td>f2</td>
<td>1,3</td>
<td>4</td>
</tr>
<tr>
<td>T3</td>
<td>f3</td>
<td>1,5,7,8</td>
<td>5</td>
</tr>
<tr>
<td>T4</td>
<td>f4</td>
<td>2,4,9</td>
<td>4</td>
</tr>
<tr>
<td>T5</td>
<td>f5</td>
<td>3,6,10</td>
<td>4</td>
</tr>
<tr>
<td>T6</td>
<td>f6</td>
<td>1,7</td>
<td>5</td>
</tr>
<tr>
<td>T7</td>
<td>f7</td>
<td>3,6,8</td>
<td>4</td>
</tr>
<tr>
<td>T8</td>
<td>f8</td>
<td>2,10</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 2** Sample test cases, fault suite, faults identified and its execution time.
We start with test case 1 (T1) out of eight available test cases. T1 is positioned at its corresponding fault suite i.e. T1 is positioned at fault suite f1 (2, 4, 7, 9), T2 at positioned at fault suite f2 (1, 3) and so on. The initial starvation for each Cuscuta (T1, T2, T3, . . . T8) is 10. I.e. initial current starvation is set equal to total number of faults. Now the Cuscuta (test case T1) will search in its domain of eight test cases, where each test case release a definite amount of chemical clue (i.e., covers definite number of faults). The Cuscuta will attack on the test case which release maximum chemical clue (faults) as per current starvation of T1. The Current starvation of Cuscuta (T1) = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Mathematically, total number of test cases requirement to fulfill T1 current starvation is equal to 10.

Faults (starvation) covered by

1. T1 = 2, 4, 7, 9. I.e. favorable test case = 4, Thus probability = favorable test case / total number of test cases = 4/10 = 0.4.

2. T2 = 1, 3. I.e. favorable test case = 2, Thus probability = favorable test case / total number of test cases = 2/10 = 0.2.

3. T3 = 1, 5, 7, 8. I.e. favorable test case = 4, Thus probability = favorable test case / total number of test cases = 4/10 = 0.4.

4. T4 = 2, 4, 9. I.e. favorable test case = 3, Thus probability = favorable test case / total number of test cases = 3/10 = 0.3.

5. T5 = 3, 6, 10. I.e. favorable test case = 3, Thus probability = favorable test case / total number of test cases = 3/10 = 0.3.

6. T6 = 1, 7. I.e. favorable test case = 2, Thus probability = favorable test case / total number of test cases = 2/10 = 0.2.

7. T7 = 3, 6, 8. I.e. favorable test case = 3, Thus probability = favorable test case / total number of test cases = 3/10 = 0.3.

8. T8 = 2, 10. I.e. favorable test case = 2, Thus probability = favorable test case / total number of test cases = 2/10 = 0.2.

So out of these probabilities test case = T1, T3 has highest probability to fulfill the current starvation of Cuscuta (T1). T1 is chosen because Cuscuta (T1) has already been placed on T1 initially. It should be noted that a random selection will be made when two or more than two test cases has same probability. But here we choose T1 because we initialize Cuscuta (T1) over test case T1.

Thus the left starvation after choosing test case (T1) by Cuscuta (T1) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] - [2, 4, 7, 9] = [1, 3, 5, 6, 8, 10]. So the current starvation now become equal to = [1, 3, 5, 6, 8, 10]. I.e. Total number of test cases = 6.

Faults (starvation) covered by

1. T1 = 0 (as all nutrients have been taken during previous attack). I.e. favorable test case = 0, Thus probability = favorable test case / total number of test cases = 0/6 = 0.

2. T2 = 1, 3. I.e. favorable test case = 2, Thus probability = favorable test case / total number of test cases = 2/6 = 0.33.

3. T3 = 1, 5, 7, 8. I.e. favorable test case = 3, Thus probability = favorable test case / total number of test cases = 3/6 = 0.5.
4. $T4= 2, 4, 9$. I.e. favorable test case = 0, Thus probability = favorable test case /total number of test cases = 0/6 = 0.

5. $T5= 3, 6, 10$. I.e. favorable test case = 3, Thus probability = favorable test case /total number of test cases = 3/6 = 0.5.

6. $T6= 1, 7$. I.e. favorable test case = 1, Thus probability = favorable test case /total number of test cases = 1/6 = 0.16.

7. $T7= 3, 6, 8$. I.e. favorable test case = 3, Thus probability = favorable test case /total number of test cases = 3/6 = 0.5.

8. $T8= 2, 10$. I.e. favorable test case = 2, Thus probability = favorable test case /total number of test cases = 2/6 = 0.33.

So out of these probabilities test case = $T3, T5, T7$ has highest probability to fulfill the current starvation (1, 3, 5, 6, 8, 10) of Cuscuta (T1). So a random selection will be made when two or more than two test cases have same probability. Lets Cuscuta chooses $T3$. Thus the left starvation after choosing test case (T3) by Cuscuta (T1) = [1, 3, 5, 6, 8, 10] - [1, 5, 7, 8] = [3, 6, 10]. So the current starvation now become equal to = [3, 6, 10], i.e. total number of test cases = 3.

In a similar manner the next attack by the Cuscuta (T1) will be on test case $T5$ and after attacking $T5$ the current starvation becomes zero, i.e. no further search or we can say a germination of a new leaf. Thus we have total of three moves by Cuscuta (T1) to fulfill its total starvation as shown in Fig. 2, with total time equal to sum of execution time of each test case, i.e. exce_time (T1) + exce_time (T3) + exce_time (T5) = 15.

![Fig. 2 Total number of attack (moves) by Cuscuta (T1) to fulfill its current starvation.](image)

In a similar manner the different attack (moves) by all Cuscuta have been shown in Table 3.

<table>
<thead>
<tr>
<th>Cuscuta</th>
<th>Initial Total starvation</th>
<th>Attack ( moves)</th>
<th>Final Starvation left</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>Attack( move) T1 T3 T5</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time 7 5 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chemical evaporated 2,4,7,9 1,5,7,8 3,6,10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>Attack( move) T2 T1 T3 T5</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time 4 7 5 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chemical evaporated 1,3 2,4,7,9 1,5,7,8 3,6,10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>Attack( move) T3 T5 T4</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time 5 4 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 Attack (moves) sequences by Cuscuta.
<table>
<thead>
<tr>
<th>Time</th>
<th>Chemical evaporated</th>
<th>Attack (move)</th>
<th>Chemical evaporated</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>T4 T5 T3</td>
<td>2,4,9 3,6,10 1,5,7,8</td>
</tr>
<tr>
<td>T5</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>T5 T3 T4</td>
<td>0 13</td>
</tr>
<tr>
<td>T6</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>T6 T1 T5 T4</td>
<td>20</td>
</tr>
<tr>
<td>T7</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>T7 T1 T3 T5</td>
<td>20</td>
</tr>
<tr>
<td>T8</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
<td>T8 T3 T1 T5</td>
<td>18</td>
</tr>
</tbody>
</table>

| Total execution time | 136 |

The CSA ordering is obtained from the Table 3 on the basis of execution time. The test case having minimum execution time is set at higher priority followed by next higher execution time. The different ordering scheme is shown in Table 4.

**Table 4** Order of test cases for various prioritization approaches.

<table>
<thead>
<tr>
<th>No order</th>
<th>Random order</th>
<th>Reverse order</th>
<th>Optimal order</th>
<th>ACO order[Yogesh Singh et. al., 2010]</th>
<th>CUSCUTA order</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T5</td>
<td>T8</td>
<td>T1</td>
<td>T3</td>
<td>T3</td>
</tr>
<tr>
<td>T2</td>
<td>T7</td>
<td>T7</td>
<td>T1</td>
<td>T3</td>
<td>T5</td>
</tr>
<tr>
<td>T3</td>
<td>T1</td>
<td>T6</td>
<td>T5</td>
<td>T4</td>
<td>T5</td>
</tr>
<tr>
<td>T4</td>
<td>T3</td>
<td>T5</td>
<td>T4</td>
<td>T1</td>
<td>T1</td>
</tr>
<tr>
<td>T5</td>
<td>T6</td>
<td>T4</td>
<td>T6</td>
<td>T7</td>
<td>T8</td>
</tr>
<tr>
<td>T6</td>
<td>T2</td>
<td>T3</td>
<td>T7</td>
<td>T8</td>
<td>T2</td>
</tr>
<tr>
<td>T7</td>
<td>T4</td>
<td>T2</td>
<td>T8</td>
<td>T6</td>
<td>T6</td>
</tr>
<tr>
<td>T8</td>
<td>T8</td>
<td>T1</td>
<td>T2</td>
<td>T2</td>
<td>T7</td>
</tr>
</tbody>
</table>
Calculating Average Percentage of Faults Detected (APFD)

APFD depends on two things (i) calculation of total percentage of test suite executed and (ii) no of fault detected by each percentage of test suite executed. In this example we elaborate this calculation w.r.t CSA Scheme.

The CSA ordering as shown in table 4 is \( T_C = \{ T_3, T_4, T_5, T_1, T_8, T_2, T_6, T_7 \} \) i.e. a total of 8 test cases in test suite \( T_C \). Thus if we execute this sequence then percentage of test suite executed is calculated as

(i) \( T_3 = (1/8)*100 = 12.5\% \). Also, for execution of T4 it is necessary to execute T3 first. i.e

(ii) \( T_3, T_4 = (2/8)*100 = 25.0\% \). Also, for execution of T5 it is necessary to execute T3, T4 first.

(iii) \( T_3, T_4, T_5 = (3/8)*100 = 37.5\% \). The rest are calculated in similar manner as

(iv) \( T_3, T_4, T_5, T_1 = (4/8)*10 = 50.0\% \).

(v) \( T_3, T_4, T_5, T_1, T_8 = (5/8)*100 = 62.5\% \).

(vi) \( T_3, T_4, T_5, T_1, T_8, T_2 = (6/8)*100 = 75\% \).

(vii) \( T_3, T_4, T_5, T_1, T_8, T_2, T_6 = (7/8)*100 = 87.5\% \).

(viii) \( T_3, T_4, T_5, T_1, T_8, T_2, T_6, T_7 = (8/8)*100 = 100.0\% \).

Now we calculate number of faults detected for each percentage of test suite execution. In case of CSA, For 12.5\% test suite execution, number of participating test cases are \( \{ T_3 \} \) only, which covers \( \{ f_1, f_5, f_7, f_8 \} \) faults out of total ten faults. Thus number of fault detected by executing 12.5\% of test suite in case of CSA is 4/10 = 0.4. In similar manner Table 5 gives percentage of fault detected by executing various percentage level of test suite in case of CSA and other prioritization Schemes.

<table>
<thead>
<tr>
<th>Prioritization Scheme</th>
<th>Percentage of test suite executed</th>
<th>Number of participating test cases ((P_{st}))</th>
<th>Faults detected by ( P_{st} )</th>
<th>Total faults detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSA ordering</td>
<td>12.5</td>
<td>T3</td>
<td>( f_1, f_5, f_7, f_8 )</td>
<td>4/10 = 0.4</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>T3, T4</td>
<td>( f_1, f_2, f_4, f_5, f_7, f_8, f_9 )</td>
<td>7/10 = 0.7</td>
</tr>
<tr>
<td></td>
<td>37.5</td>
<td>T3, T4, T5</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>T3, T4, T5, T1</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>62.5</td>
<td>T3, T4, T5, T1, T8</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>T3, T4, T5, T1, T8, T2</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>87.5</td>
<td>T3, T4, T5, T1, T8, T2, T6</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>100.0</td>
<td>T3, T4, T5, T1, T8, T2, T6, T7</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td>Ant Colony based</td>
<td>12.5</td>
<td>T3</td>
<td>( f_1, f_5, f_7, f_8 )</td>
<td>4/10 = 0.4</td>
</tr>
<tr>
<td>ordering</td>
<td>25</td>
<td>T3, T5</td>
<td>( f_1, f_3, f_5, f_6, f_7, f_8, f_{10} )</td>
<td>7/10 = 0.7</td>
</tr>
<tr>
<td></td>
<td>37.5</td>
<td>T3, T5, T4</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>T3, T5, T4, T1</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>62.5</td>
<td>T3, T5, T4, T1, T7</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>T3, T5, T4, T1, T7, T8</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>87.5</td>
<td>T3, T5, T4, T1, T7, T8, T6</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>100.0</td>
<td>T3, T5, T4, T1, T7, T8, T6, T2</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td>Optimal ordering</td>
<td>12.5</td>
<td>T1</td>
<td>( f_2, f_4, f_7, f_9 )</td>
<td>4/10 = 0.4</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>T1, T3</td>
<td>( f_1, f_2, f_4, f_5, f_7, f_8, f_9 )</td>
<td>7/10 = 0.7</td>
</tr>
<tr>
<td></td>
<td>37.5</td>
<td>T1, T3, T5</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>T1, T3, T5, T4</td>
<td>( f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10} )</td>
<td>10/10 = 1.0</td>
</tr>
</tbody>
</table>
Thus for various ordering the APFD is calculated by finding the area under the curve enclosed by the solid line as shown in figure 3. We can also calculate the APFD using the formula given by Krishnamoorthi et al., 2009. Both methods result in the same output.

10 Comparison with Different Ordering
We compare the result of proposed CSA ordering with No order, Random order, Reverse order, optimal order, Ant colony order (ACO order) for the test cases order as shown in Table 4. The various approaches and their prioritization order as mentioned in Table 4 are compared by calculating their average percentage of faults detected (APFD). The comparison results are shown in Fig. 3.
Results obtained by measuring the Average Percentage of Faults Detected (APFD) shows that Cuscuta ordering has the same APFD as that of optimal ordering and ant colony optimization ACO (Singh et. al., 2010) but better than No order, Random order and Reverse order. The graph clearly shows the effectiveness of Cuscuta search in detecting average percentage of faults.

11 Application of the CSA

CSA can be used in large and complex test suite prioritization problems and thus saving bigger amount of time and cost during software development life cycle as compared to smaller ones. With this approach software testers can easily select and prioritize test cases with minimum execution time and higher percentage of fault detection.

12 Discussion

We have proposed a selection and prioritization technique based on Cuscuta search Algorithm to find the near
optimal solution. By calculating APFD (Average Percentage of Faults Detected) for each technique, we conclude that Cuscuta ordering gives same results as given by the optimal and ACO ordering but better than No order, Random order and Reverse order. Cuscuta is strong in its searching method as it knows its currents starvation during each attack over the host and hence lead to better solutions in optimal time.

This algorithm suggests a critical use of nature inspired approach in the field of software testing. The paper arguments about the intelligence of plants as like animals. A part from CSA, Particle Swarm Optimization (PSO), Artificial Bee Colony Optimization (ABC) and Genetic Algorithm are few other metahurestic inspired by animal’s intelligent behavior on which research can be carried out to exploit natural intelligence of species and to solve NP problems in software testing.

References
Ayari K, Bouktif S, Antoniol G. 2007. Automatic mutation test input data generation via ant colony. Proceeding of. 9th annual conference on Genetic and evolutionary computation, 1074-1081


