[[1]](#footnote-1)

**EFFICIENT SOFTWARE TESTING METHODS TO INCREASE CODE RELIABILITY**

**Arunim Roy Arpit Rathi Parmeet Singh**

**ABSTRACT—** Software Testing is an integral part of Software Development. It is the method used to provide the stakeholders with the quality and efficiency of the software being developed. It also provides with a better and efficient code that is more reliable. The study of software testing methods can help a student to produce more reliable codes. Thus we must find a way to create more efficient software testing methods, the most common method being Automated Software Testing. The first step towards this approach is to find a way to generate more and accurate test cases automatically. In this project, we are going to see many methods for this task and compare and contrast them to find out the most efficient and fastest method automated test case generation.

We will basically look into two methods: Artificial Bee Colony Method and Test Case Generation by UML Diagram by modified Depth First Search.

We will compare the two methods on the basis of test-case prioritization techniques and finally will come out with the most optimal method for test case generation.

**INTRODUCTION—** SoftwareTesting is a process used to help identify the correctness, completeness and quality of developed computer software. With that in mind, testing can never completely establish the correctness of computer software.

There are many approaches to software testing from using alm tools to automated testing, but effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following rote procedure.

The quality of the application can and normally does vary widely from system to system but some of the common quality attributes include reliability, stability, portability, maintainability and usability.

**Software testing** is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.

Software testing can also be stated as the process of validating and verifying that a software program/application/product:

* meets the business and technical requirements that guided its design and development;
* works as expected; and
* can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Testing can never completely identify all the defects within software. Instead, it furnishes a criticism or comparison that compares the state and behaviour of the product against oracles—principles or mechanisms by which someone might recognize a problem. Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing is the process of attempting to make this assessment.

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing tends to answer such questions as "how many people can log in at once".

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, spread sheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

**IMPLEMENTATION-**

**ARTIFICIAL BEE COLONY**

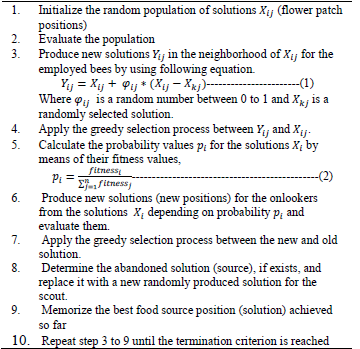


Figure 2. Artificial Bee Colony algorithm

For test data generation, initially a random population of candidate solutions is generated from the inputs’ domains, with reference to the flower’s positions. The optimum positions of flower patches are searched out in such a way so that positions of these can satisfy the targeted path constraint system. With respect to each flower patch, its profitability is measured. In computer modelling, this profitability is represented by the fitness of the position. The ABC algorithm works in three phases. First phase is

called the employed phase, where employed bees modify the position of elite flower patches (where profitability is higher) in neighborhood. The second phase is executed by onlookers, who modify their patches’ positions with influence from elite positions. These positions are known as selected patches. After every phase a greedy selection process is repeated, where solutions (flower patch

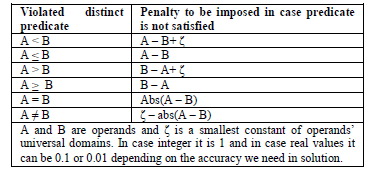
positions) compete themselves for retention in the elite or selected patches based on their fitness. In this process some of the sources may migrate from one category to another or some may be abandoned in favor of randomly generated sources, which are simulated by the scout phase of the algorithm. Subsequently, these search processes of the employed, onlooker, and scout phases are repeated in cycles until the termination criterion is met. The ABC algorithm doesn’t have much flexibility in tuning its parameters for the best results other than the size of colony and number of bees’ allocation for the elite patches.

Fitness Function

For path testing criterion, in order to traverse a

feasible path, the control must satisfy the entire branch predicates, which falls on that particular path. So, corresponding to each path a compound predicate (CP) is made by ‘***anding’*** each branch predicate of the path. The CP must be evaluated to true by a candidate solution in turn to become a valid test case. The ABC generates population of candidate solutions and these are used to evaluate the CP. If the CP is not evaluated to true by an individual then all the constraints of particular path are split into distinct predicates (DP) and one by one each DP is evaluated by taking values of its operands from candidate solutions. A DP contains only one operator (a constraint with modulus operator is the exception) and can be expressed in the form of expression A op B where A and B are LHS and RHS of an expression which is made of one or more operand(s) and op is a relational operator. For determining the fitness of candidate

solution following rule is followed. If the DP is satisfied then no penalty is imposed on the candidate solution, otherwise the candidate solution is penalized on the basis of branch distance concept rules as shown in table 1:



After this, integrated fitness due to whole of CP is

determined by adding penalty values of two DPs, if they are connected by a conditional ‘and’ operator. If two DPs are connected by a conditional ‘or’ operator then the minimum penalty of two DPs is considered for the evaluation of whole CP fitness. If the integrated fitness is zero then the CP is called evaluated or satisfied by the individual, whose values are replaced in the CP and the search process for particular path is terminated otherwise search is allowed to proceed further.

Experimental setup

In order to carry out procedures of the ABC method, we have to carry out tests in ten real world problems. The aim is to automatically generate test cases from control flow diagram using ABC method. The CFG of programs are automatically constructed from source codes and paths identified manually. The ABC algorithm is implemented in MATLAB environment. Performance algorithms are measured using Average Test cases generated per path(ATCPP) and Average Percentage Coverage(APC) metrics. A high value of ATCPP signifies the difficulty in search method for test case generation. Experiment is conducted 10 times for averaging the results. Each time the ABC is iterated for 100 generations for each of the 10 runs. In each run, the first- generation population is seeded with the best solution from the previous run.

We will conduct the ABC test on 5 of the real world problems which are known as test objects.

* Triangle classifier(TC) is one of the most used programs for test case generations. It accepts the sides of a triangle and decides whether the sides form a triangle or not and of what type. The program contains 7 feasible paths of which 4 involves equality constraints.
* Line rectangle classifier program finds out whether a line cuts a rectangle or lied completely outside or lies completely inside of the rectangle. In this program 8 inputs are accepted, 4 for the rectangle and 4 for the line. The nodes of CFG of this program involve very high nesting and henceforth, using this program can help find difficulty of testing a nested structure.
* Quadratic equation program finds the roots of a quadratic equation.
* Min Max program finds minimum and maximum value from an array. In this program loop is allowed to execute 5 times.
* Isprime program is used to check whether an integer is a prime number or not.

**ANT COLONY OPTIMIZATION ALGORITHM**

BASIS OF THE ALGORITHM

This algorithm is inspired by the ants and how they increase their efficiency to collect food.

The ants are very intelligent creatures. An ant searching for food lays along a certain amount of pheromone trail. This fulfils two main purpose :

1. It provides a path for the other randomly moving ants to follow and hence decreases their random behavior.
2. It provides a path for the ants to get back to the source.

The main advantage of this method that the ants follow is that as the time progresses, more and more ants travel along different paths and at last if a new ant comes, it can easily decide its path by the amount of pheromone on each path and hence can choose the best suited path.

This process of ants moving back and forth from food source to its colony is known as Ant Colony Optimization.

Ant Colony Optimization is a probability based computation problem algorithm which generates solutions traversing a graph consisting of different states of system. This algorithm is used in software testing to generate test sequences. The algorithm takes a control flow graph as input and traverses nodes to find optimized paths in the graph.

SETS AND VARIABLES :

* Node Set – it represents all the nodes of the Activity Graph.
* Edge Set- it represents the edges between two consecutive nodes.
* Feasible Track – F­­XY =1, path exists else 0, no path exist.
* Pheromone Value- helps other ants to find and select a path.
* Heuristic Value – The visibility of a path is denoted by it.
* Visited Status- tells if a node is visited or not.
* Probability value- the heuristic, feasibility and pheromone values are together used in calculating the probability of a node to be visited.

ALGORITHM :

Run the Algorithm

2.1 W**hile (sum>0)**

2.1.1 **Initialize:** x= start, weight=0, visit=0.

The variable Weight is used for calculating

the strength of the path, Later on the

variable Strength is used for prioritizing

the paths. Next variable Visit is used to

discard the redundant paths.

2.1.2 **Update the Visited status:** If Ant has

traversed any node ‘x’ then the value of the

visited status of that vertex ‘x’ is updated.

If the visited status for the any

vertex ‘x’ is (Vs[x]=0) then it will be

updated to (Vs[x]=1), also the value of

variable ‘visit’ is updated to Visit=Visit+1.

2.1.3 **Evaluate Feasible track set:** The feasible

path F(q) for the current vertex ‘x’ is

determined, This part also evaluates the

available possible routes from the current

vertex ‘x’ to all the adjacent vertices using

Activity Graph. If no feasible path exists

then go to step 3.

2.1.4 **Observe the path:** The probability of each

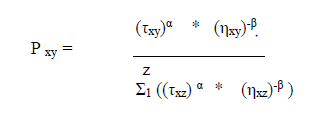
node is calculated in the feasibility set F(q)

to observe the path. Now the probability for

each non-zero element belongs to feasible

set F(q) is calculated, with the help of

following formula.



where every z belongs to feasible set F(q).

2.1.5 **Traverse to next vertex:** for visiting the

next vertex use the following rules:-

**R1:** Choose the path(xy) having maximum

probability (Pxy).

**R2:** If Probability of two or more paths are

equal e.g., for paths xy and xz , (Pxy

= Pxz) then rule 2.1is being followed.

**R2.1:** In case of self transition choose that path;

otherwise below:

**R2.2 : E**ach feasible set node is compared with

the end node, if (feasible set node = =

end\_node) then choose end\_node as the

next node otherwise follow rule below:

**R2.3 :** Select that node with visiting status,Vs=0

that means a node which has not been

visited yet. Choose the nodes randomly

in case two or more states are having

same visited status i.e. Vs[y]=Vs[z] .

2.1.6 **Update pheromone value:**

Pheromone value is updated for the path

(x->y) through the following rule:

**(τxy) = (τxy)α + (ηxy)-β**

2.1.7 **Update Heuristic value:**

Heuristic information for an ant is being

kept up to date by the formula:

ηxy = 2\* (ηxy)

2.1.8 **Evaluate strength:** This factor contains

the values related with each path and it is

calculated as:-.

weight = weight + τxy

strength [sum]=weight.

start = next\_vertex.

2.1.9 If (start != end\_node) then

go to step 2.1.3

else

if (visit = = 0) then reject the path it is

superfluous path otherwise add new path.

Sum=Sum-1(decrement sum by one each

time).

3. END //end of Algorithm

The proposed algorithm is good

1) at removing redundant paths

2) it prioritizes the paths

3) covers full path

4) capable of finding errors at earlier stage of testing.

**GENETIC ALGORITHM**

A Genetic Algorithm starts with guesses and attempts to improve the guesses by evolution. A GA will typically have five parts:

(1) a representation of a guess called a chromosome,

(2) an initial pool of chromosomes,

(3) a fitness function,

(4) a selection function and

(5) a crossover operator and a mutation operator.

A chromosome can be a binary string or a more elaborate data structure. The initial pool of chromosomes can be randomly produced or manually created. The fitness function measures the suitability of a chromosome to meet a specified objective: for coverage based ATG, a chromosome is fitter if it corresponds to greater coverage. The selection function decides which chromosomes will participate in the evolution stage of the genetic algorithm made up by the crossover and mutation operators. The crossover operator exchanges genes from two chromosomes and creates two new chromosomes. The mutation operator changes a gene in a chromosome and creates one new chromosome. GA has well-defined steps.

Basis of the Algorithm

The pseudo code for GA is:

Initialize (population)

Evaluate (population)

While (stopping condition not satisfied)do

{

Selection (population)

Crossover (population)

Mutate (population)

Evaluate (population)

}

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

Procedure-

Input: CFG of the code

The first step of algorithm is assigning weights to CFG.

An initial credit is taken (100 or 10), if CFG is dense i.e. large numbers of edges are there than initial credit should be taken as 100 and if CFG is sparse (small codes) then it can be taken as 10.

At each node of CFG the incoming credit (sum of the weights of all the incoming edges) is divided and distributed to all the outgoing edges of the node. Distribution of weights is done as follows: Take ‘n’ to be the number of outgoing edges.

We will considered an 80-20 rule. 80 percentage of weight of the incoming credit is given to loops and branches and the remaining 20 percentage of the incoming credit is given to the edges in sequential path. From each node if n1 is the number of edges in sequential path and n2 is the number of edges in looping and branching paths, then n1 edges are given 20 percentage of incoming weight and then divided equally amongst them and the remaining 80 percentage is given to n2 edges. If there is only one outgoing edge from a particular node than the incoming weight is assigned to the outgoing edge.

Selection

The selection of parents for reproduction is done according to a probability distribution based on the individual’s fitness values. First the fitness value is calculated using the Fitness function proposed in the algorithm. Weights are used to determine the relative contribution of a path to the fitness calculation.

The fitness function we are using here is



Where, wi = weight assigned to i-th edge on the path under consideration.

After all the fitness function values are calculated, the probability of selection pj for each path j, so that



Then cumulative probability ck is calculated for each path k with equation:



REPRODUCTION (CROSSOVER)-

In one-point (or single) crossover, two input data selected as potential parents by selection process exchange substring information at a random position in the data to produce two new data. Crossover happens according to a crossover probability pc, which is an adjustable parameter. For each parent selected, generate a random real number r in the range [0, 1]; if r < pc then select the parent for crossover. After that, the selected data are formatted randomly. Each pair of parents generates two new paths, called offspring. The crossover ­­­­technique used is one point crossover done at the midpoint of the input bit string. In this technique, right half of the bits of one parent are swapped with the corresponding right half of the other parent.

MUTATION-

Mutation is performed on a bit-by-bit basis. Every bit of every chromosome in the offspring has an equal chance to mutate (change from ‘0’ to ‘1’ or from ‘1’ to ‘0’), and the mutation occurs according to a mutation probability pm, which is also an adjustable parameter. To perform mutation, for each chromosome in the offspring and for each bit within the chromosome, generate a random real number r in the range [0, 1]; if r < pm then mutate the bit.

These major components including the fitness function will evolve test data to better ones, trying to find a candidate that covers the target path. The crossover process tries to create better test data from fitter ones, while mutation introduces diversity into population, avoiding getting stuck at local optima solutions.

**TESTING STRATEGY—**

Test Strategy document is a high level document and is usually developed by a project manager. This document defines “Software Testing Approach” to achieve testing objectives. Test Strategy document is a static document meaning that it is not often updated.

There are different types of techniques and methodologies involved in testing:

* Functional Testing
* Unit Testing
* Integration Testing
* System Testing
* Acceptance Testing
* Non-Functional Testing
* Performance, Load, Stress Testing
* Security and Vulnerability Testing

### **Structural vs Functional Testing[9]**        Structural testing is considered white-box testing because knowledge of the internal logic of the system is used to develop test cases. Structural testing includes path testing, code coverage testing and analysis, logic testing, nested loop testing, and similar techniques. Unit testing, string or integration testing, load testing, stress testing, and performance testing are considered structural.       Functional testing addresses the overall behavior of the program by testing transaction flows, input validation, and functional completeness. Functional testing is considered black-box testing because no knowledge of the internal logic of the system is used to develop test cases. System testing, regression testing, and user acceptance testing are types of functional testing.  Both methods together validate the entire system. For example, a functional test case might be taken from the documentation description of how to perform a certain function, such as accepting bar code input.     A structural test case might be taken from a technical documentation manual. To effectively test systems, both methods are needed. Each method has its pros and cons, which are listed below:  **Structural Testing** Advantages The logic of the software’s structure can be tested. Parts of the software will be tested which might have been forgotten if only functional testing was performed.

### Disadvantages Its tests do not ensure that user requirements have been met. Its tests may not mimic real-world situations.

### **Functional Testing** Advantages Simulates actual system usage. Makes no system structure assumptions.

### Disadvantages Potential of missing logical errors in software. Possibility of redundant testing.

**Static vs Dynamic Testing[10]**

Static Testing is type of testing in which the code is not executed. It can be done manually or by a set of tools. This type of testing checks the code, requirement documents and design documents and puts review comments on the work document. When the software is non –operational and inactive, we perform security testing to analyse the software in non-runtime environment. With static testing, we try to find out the errors, code flaws and potentially malicious code in the software application. It starts earlier in development life cycle and hence it is also called verification testing. Static testing can be done on work documents like requirement specifications, design documents, source code, test plans, test scripts and test cases, web page content.

* **Inspection**: Here the main purpose is to find defects. Code walkthroughs are conducted by moderator. It is a formal type of review where a checklist is prepared to review the work documents.
* **Walkthrough**: In this type of technique a meeting is lead by author to explain the product. Participants can ask questions and a scribe is assigned to make notes.
* **Technical reviews**: In this type of static testing a technical round of review is conducted to check if the code is made according to technical specifications and standards. Generally the test plans, test strategy and test scripts are reviewed here.
* **Informal reviews**: Static testing technique in which the document is reviewed informally and informal comments are provided.

Dynamic testing is done when the code is in operation mode. Dynamic testing is performed in runtime environment. When the code being executed is input with a value, the result or the output of the code is checked and compared with the expected output. With this we can observe the functional behaviour of the software, monitor the system memory, CPU response time, performance of the system. Dynamic testing is also known as validation testing, evaluating the finished product. Dynamic testing is of two types: Functional Testing and Non-functional testing.

* **Unit Testing:** Testing of individual modules by developers.. The source code is tested in it.
* **Integration Testing:**Testing the interface between different modules then they are joined..
* **System Testing:**Testing performed on the system as a whole.
* **Acceptance Testing:** Testing done from user point of view at user’s end.

**DIAGRAM—**

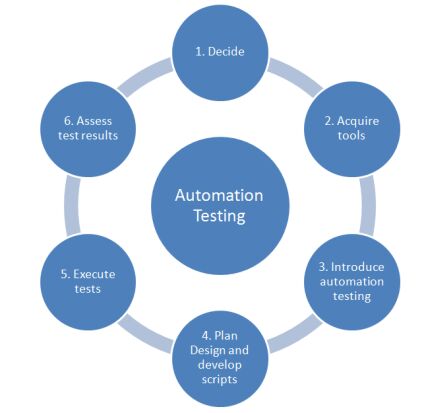


Figure 1 - AUTOMATED TESTING DIAGRAM

**LITERATURE SURVEY—**

In the testing phase of software development, the main motivation and function of search algorithms is to incorporate inputs from the input domain such that they are able to reveal faults in the software. There are several conventional software testing techniques available such as mutational testing, random testing and algorithmic searches, but these techniques are somewhat outdate and have disadvantages of slow search capability.[1]

It is a well-known fact that software testing is a resource- consuming activity. Studies show that testing constitutes more than 50% of the overall costs of software development[2] and with the increasing complexity of software, the proportion of testing costs will continue to rise unless more effective ways of testing are found. The main focus of investigation in industry should be for reducing cycle time and development costs, and at the same time increasing software quality and this is achieved only by improving their software testing processes.

Software testing is an important and critical phase that deals with software quality. However, software testing that consists of three phases (test case generation, test execution and test evaluation) is time consuming activity that requires a lot of resource. Therefore, automated testing is strived to save resource spent in the terms of time, cost and effort and to give more accurate result than manual testing that vulnerable to human error. Towards automated testing, automating test cases generation as the first testing process is being highlighted. Test cases can be generated automatically from source code or visual software model such as Unified Modelling Language (UML), Data Flow Diagram (DFD), or Entity Relationship Diagram (ERD). [3]

Artificial bee colony method[8], also known as ABC algorithm is a very recent mechanism of software testing developed by Dervis Karaboga in 2005, with reference to behaviour of honey bees. It uses common parameters such as colony size and maximum cycle number. It is extremely simple to implement the algorithm just like Particle Swarm Optimization (PSO) and Differential Evolution (DE) algorithm. This technique has successfully been employed on several engineering domains such as internet server allocation, job scheduling, pattern recognition, data clustering etc. but still it’s applications in testing domain haven’t been explored as much.

The ABC algorithm is biologically inspired technique of swarm intelligence for searching [10]. It is all about honey bees’ work distribution and collective foraging strategy to accumulate extra nectar for their survival in winter season. Seeley investigated the behavior of bees in distributing their work to optimize the collection of nectar Instead of initiating exploration by all bees, some dedicated explorer bees (scout bees) are appointed to explore the “profitability” of flower patches in the surrounding environment. This profitability accounts various parameters such as amount of nectar in flower patches, sugar content in nectar, distance of flower patches from the bee hive etc. If an explorer bee satisfies itself that there is sufficient profitability then it recruits unloader for unloading the nectar it has collected during exploration and dances (known as waggle dance) on dance floor (a designated place in beehive) to give feedback to foragers (observer or onlookers bees, which actually collect nectar from patches) about the quality of the flower patch, which they have recently searched out. The dance strength and its inclination with Sun determine the distance and the direction of the designated flower patch from beehive. The working of the honeybee colony is reported as robust and adaptive by [4]. [1]

**TEST CASE PRIORITIZATION—**

We formally define the test case prioritization problem as follows[5]:

Definition 1: The Test Case Prioritization Problem:

In this definition, PT represents the set of all possible prioritizations (orderings) of T, and f is a function that, applied to any such ordering, yields an award value for that ordering. (For simplicity, and without loss of generality, the

definition assumes that higher award values are preferable to lower ones.)

Goals with test case prioritization:

* To increase the rate of fault detection of a test suite, that is, the likelihood of revealing faults earlier in a run of regression tests using that test suite.
* To increase the coverage of coverable code in the system under test at a faster rate, thus allowing a code coverage criterion to be met earlier in the test process.
* To increase their confidence in the reliability of the system under test at a faster rate.
* To increase the rate at which high risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
* To increase the likelihood of revealing faults related to specific code changes earlier in the regression testing process.

These are qualitative ways to prioritize test cases.

We can also prioritize test cases quantatively.

We must first describe the goal quantatively.

1. We define one particular function f for use in quantifying the first of these goals.
2. Second, depending upon the choice of f, the test case prioritization problem may be intractable or undecidable.
3. Third, test case prioritization can be used either in the initial testing of software or in the regression testing of software. One difference between these two applications is that, in the case of regression testing, prioritization techniques can use information gathered in previous runs of existing test cases to help prioritize the test cases for subsequent runs.
4. Fourth, it is useful to distinguish two varieties of test case :
5. General test case prioritization –

In general test case prioritization,

given program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering of test cases that will be useful over a succession of subsequent modified versions of P. Thus, general test case prioritization can be

performed following the release of some version of the program during off-peak hours, and the cost of performing the prioritization is amortized over the subsequent releases.

It is hoped that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, on average over those subsequent releases.

1. Version specific test case prioritization.

In version-specific test case prioritization, given

program P and test suite T, we prioritize the test cases in T with the intent of finding an ordering that will be useful on a specific version P0 of P. Version-specific prioritization is performed after a set of changes have been made to P and prior to regression testing P0. Because this prioritization is accomplished after P0 is available, care must be taken to

keep the cost of performing the prioritization from excessively delaying the very regression testing activities it is intended to facilitate. The prioritized test suite may be more effective at meeting the goal of the prioritization for P0 in

particular than would a test suite resulting from general test case prioritization, but may be less effective on average over a succession of subsequent releases.

**Prioritization for Rate of Fault Detection:**

Here we are going to consider two heuristics methods for test case prioritization based on Fault Rate Detection[6].

1. **Total statement coverage prioritization**. By instrumenting a program, we can determine, for any test case, which statements in that program were exercised (covered) by that test case. We can then prioritize test cases in terms of the total number of statements they cover by counting the number of statements covered by each test case and then sorting the test cases in descending order of that number. (When multiple test cases cover the same number of statements, an additional rule is necessary to order these test cases; we order them randomly.)
2. **Additional statement coverage prioritization** iteratively selects a test case that yields the greatest statement coverage, then adjusts the coverage information on all remaining test cases to indicate their coverage of statements not yet covered and repeats this process until all statements covered by at least one test case have been covered. (When multiple test cases cover the same number of statements not yet covered, an additional rule is necessary to choose one of these test cases; we do this randomly.

**REFERENCES—**

[1] Dahiya, Surender Singh, Jitender Kumar Chhabra, and Shakti Kumar. "Application of artificial bee colony algorithm to software testing." *Software Engineering Conference (ASWEC), 2010 21st Australian*. IEEE, 2010.

[2] Harrold, M.J. , 20 0 0. Testing: a roadmap. In: Proceedings of the Conference on The Fu- ture of Software Engineering (ICSE’00). ACM, New York, NY, USA.

[3] Septian, Irwandhi, Ricky Setiawan Alianto, and Ford Lumban Gaol. "Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm." *Procedia Computer Science* 116 (2017): 629-637.

[4] T. Schmickl, R. Thenius and K. Crailsheim “Simulating swarm intelligence in honey bees: foraging in differently fluctuating environments,” GECCO'05, Washington, DC, USA, 2005, pp. 273-274.

[5] Srivastava, P. R. (2008). TEST CASE PRIORITIZATION. Journal of Theoretical & Applied Information Technology, 4(3).

[6] Elbaum, S., Malishevsky, A. G., & Rothermel, G. (2002). Test case prioritization: A family of empirical studies. IEEE transactions on software engineering, 28(2), 159-182.

[7] J. Wegener, A. Baresel and H. Sthamer, “Evolutionary test environment for automatic structural testing,” Information and Software Technology, 2001; Vol. 43, pp. 841–54.

[8] C. Chong, M. Low, A. Sivakumar and K. Gay, “A bee colony optimization algorithm to job shop scheduling,” Proceedings of the 37th Winter Simulation, Monterey, California, 2006, pp. 1954-1961

[9] Srivastava, P. R., Chis, M., Deb, S., & Yang, X. S. (2012). An efficient optimization algorithm for structural software testing. International Journal of Artificial Intelligence™, 8(S12), 68-77.

[10] Fairley, R. E. (1978). Tutorial: Static analysis and dynamic testing of computer software. Computer, 11(4), 14-23.

1. [↑](#footnote-ref-1)