Test Case Prioritization using Ant Colony Optimization

A Thesis

Submitted in fulfilment of the requirements for the award of the degree of

Master of Science

in

Mathematics and Computing

Submitted by

Pawanpreet Kaur

(Registration No. 301003016)

Under the supervision of

Dr. Rajesh Kumar

School of Mathematics and ComputerApplications,

Thapar University,

Patiala –147004 (PUNJAB)

INDIA

JULY 2012

Certificate

I hereby certify that the work which is being presented in the thesis entitled Test Case Prioritization using Ant Colony Optimization" in partial fulfilment of the requirements for the award of degree of Master of science, School of Mathematics and Computer Applications, Thapar University, Patiala is an authentic record of my own work carried out under the supervision of Dr. Rajesh Kumar.

The matter presented in the thesis has not been submitted for the award of any other degree of this or any other university.

Pawanpreet Kaur

(Pawanpreet Kaur)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

11/07/2014

Dr. Rajesh Kumar
(Supervisor)
Associate Professor
SMCA, Thapar University
Patiala

Countersigned by:

Dr. S.S. Bhatia
(Professor & Head)
School of Mathematics & Computer Applications
Thapar University, Patiala

Dr. S.K. Mohapatra
Dean of Academic Affairs
Thapar University, Patiala

i

Acknowledgement

The key elements concentration, dedication, hard work and application are not the only essential factors for achieving the desired goals but also guidance, assistance and co-operation of people is necessary.

I would like to express my deep and sincere gratitude to my supervisor Dr. Rajesh Kumar, Associate Professor, School of Mathematics and Computer Applications (SMCA). His wide knowledge and logical way of thinking have been of great value for me. His understanding and personal guidance have provided a good basis for the present thesis. I could never imagine to have better mentor than him.

I owe my sincere thanks to Dr. S.S. Bhatia, Professor and Head of School of Mathematics and Computer Applications (SMCA), Thapar University, Patiala for providing facilities. I wish to express my sincere thanks to Mr. Manoj Pachariya, Assistant Professor, Galgotias University, Greater Noida, for the possible help extended especially, whenever needed. My thesis work would have incomplete without thanking my friend Mrs.HarpreetKaur.

I was very fortunate to have unconditional support from my family. I thank my parents, who gave me the courage to get my education, supported me in all achievements throughout my life. Without their encouragement and suggestion, this work would indeed have been very difficult for me to tackle. Last but not least, thanks to God for giving me inner peace and strength. May your name be exalted, honoured and glorified.

Pawanpreet Kaur

Pawanpreet Kaur

(301003016)

Contents

Abstract

Regression testing is primarily a maintenance activity that is performed frequently to ensure the validity of the modified software. In such cases, due to time and cost constraints, the entire test suite cannot be run. Thus, it becomes essential to prioritize the test cases in order to cover maximum statement coverage in minimum time. In order to solve time constraint, Test Case Prioritization Problem Ant Colony Optimization (ACO) is being used. Ant Colony Optimization is a meta-heuristic approach that has been applied for time constraint, test case prioritization problem considering the statement coverage and execution time of each test case. This work presents the regression test case prioritization technique that reorders test cases in a test suite in a time constraint environment. Simulation using MATLAB has also been done which gives near optimal solutions.

Chapter 1

Introduction

1.1 Software Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an independent view of the software to allow the business to appreciate and understand the risks of software implementation.

Software testing can be stated as the process of validating and verifying the software which:-

- Meets the requirements that guided its design and development.
- Works as expected.
- Can be implemented with the same characteristics.
- Satisfies the needs of stakeholders.

1.2 Types of Testing

Mainly there are four types of testing:-

a) **Regression Testing:** Re-running the test cases from the existing test suite to build or regain confidence in the correctness of the modified software is referred to as regression testing. The maintenance phase of a software product needs to go through the inevitable regression testing process. It is required to re-execute the existing test suite whenever any addition, deletion or modifications are made to the software.

b) **White Box Testing:** White Box Testing (WBT) involves looking at the structure of the code. When we know the internal structure of the product, tests can be conducted to ensure that the internal operations performed according to the specification, and all

the internal components have been adequately exercised. In others word WBT tends to involve the coverage of the specification in the code.

Code coverage is defined as follows [1]:-

- ***Segment Coverage***: Each segment of code between control structure is executed at least once.
- ***Branch Coverage***: Each branch in the code is taken in each possible direction at least once.
- ***Statement Coverage*:**Each statement in the code is executed at least once.
- ***Compound Condition Coverage***: When there are multiple conditions, we must test not only each direction but also each possible combinations of conditions, which is usually done by using a truth table.
- ***Data Flow Coverage(DFC)****:* We track the specific variables through each possible calculation, thus defining the set of intermediate paths through the code i.e those based on each piece of code chosen to be tracked. Even though the paths are considered independent, dependencies across multiple paths are not really tested. It tends to reflect dependencies but it is mainly through sequences of data manipulation. It tends to uncover bugs like uninitialized or undeclared variables.
- ***Path Coverage***: Path Coverage is where all possible paths through the code are defined and are executed at least once.
- ***Loop Coverage***: Loop Coverage includes single loops, concatenated loops and nested loops. Loops are fairly simple to test unless dependencies exist among the loop or between a loop and the code it contains.

c) Black Box Testing: Black Box Testing (BBT) is also known as behavioural testing. It is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.

Following approaches are used for designing the test cases for Black Box Testing :

- ***Boundary Value Analysis (BVA)***: This testing technique believes and extends the concept that the density of defect is more towards the boundaries. This is done due to the following reasons:
  - Usually the programmers are not able to decide whether they have to use ≤ operator or < operator when trying to make comparisons.

- Different terminating conditions of For loops, while loops and repeat loops may cause defects to move around the boundary conditions.
- The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

➢ ***Equivalence Class Testing***: The use of equivalence classes as the basis for functional testing is appropriate in situations like :

- When exhaustive testing is desired.
- When there is a strong need to avoid redundancy.

➢ ***Decision Table Based Testing***: Decision tables are a precise and compact way to model complicated logic. Out of all functional testing methods, the ones based on decision tables are the most rigorous due to the reason that the decision tables enforce logical rigour. Decision tables are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions.

➢ ***Cause-Effect Graphing Technique***: This is basically a hardware testing technique adapted to software testing. It considers only the desired external behaviour of a system. This is a testing technique that aids in selecting test cases that logically relate causes (inputs) to effects (outputs) to produce test cases.

d) Grey Box Testing: It is a software testing method which is a combination of black box and white box testing method. In black box testing, the internal structure of the item being tested is unknown to the tester and in white box testing the internal structure is known. In grey box testing, the internal structure is partially known. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing is done at the user level.

1.3 Levels of Testing

The system is tested in steps, in line with the plan and release strategies, from individual units of code through integrated subsystems to the deployed releases and to the final system. Testing proceeds through various physical levels of the application development lifecycle. Each completed level represents a milestone on the project plan

and each stage represents a known level of physical integration and quality. These stages of integration are known as test levels. Levels of testing include the following:

a) **Unit Testing**: This verifies the program specifications to the internal logic of the program or module and validates the logic.

b) **Integration Testing**: This verifies proper execution of application components including interfaces. Communication between modules within the sub-system is tested in a controlled and isolated environment within the project.

c) **System Testing**: It verifies the proper execution of the entire application components including interfaces to other applications. Both functional and structural types of tests are performed to verify that the system is functionally and operationally sound.

d) **System Integration Testing**: It verifies the integration of all applications, including interfaces internal and external to the organization, with the hardware, software and infrastructure components in a production like environment.

e) **User Acceptance Testing**: It verifies that the system meet users requirements as specified. It simulates the user environment and emphasizes security, documentation and regression tests.

f) **Operability Testing**: It verifies that the application can operate in the production environment. Operability tests are performed after, or concurrent with user acceptance tests.

1.4 Test Case Optimization

Test Case: A test case is a input on which the program under test is executed during testing. A test case is a set of conditions or variables under which a tester will determine whether an application or a software system is working correctly or not.

Test Case Optimization: Test case optimization is a problem of finding the best subset of test cases from a pool of test cases to be audited. Following are the steps of test case optimization:

a) **Test Case Minimization**: Test cases minimization is a selection of smallest subset of test cases from a pool of test cases to be audited for a program. It covers as many program elements as the entire pool does. This technique seeks to reduce the effort required for regression testing by selecting an appropriate subset of test cases. Mathematically, it is stated as:-

   Let T be a set of test cases and R be a set of test objectives. M is a test case minimization of T with respect to R if and only if M is a subset of T that maximizes R(T).

b) **Test Case Selection**: It is also finding minimal cardinality subset of test cases from the pool of test cases. One major difference between test cases minimization and test case selection is that the test case selection chooses a temporary subset of test cases, whereas test case minimization reduces the test case permanently based on some external criterion such as structural coverage.

c) **Test Case Prioritization**:Test Case Prioritization techniques try to find an ordering/ranking of test cases, so that the some adequacy can be maximised as early as possible.

d) **Test Case Filteration**: It is to chunk out subset of closely related test cases, so that large portion of defects would be found as if the whole test suite was to be used.It is often desirable to filter a pool of test cases for program in order to identify a subset that will actually be executed and audited at a particular time.

1.5 Test Data Adequacy Criteria

Software test adequacy criteria are the rules to determine whether a software system has been adequately tested, which points out the central problem of software testing. Number of test data adequacy criteria has been proposed and investigated in the literature like code coverage or control flow-based test adequacy criteria, data flow based adequacy criteria, fault-based adequacy criteria, and error-based criteria. The control-flow based adequacy criteria includes statement coverage, branch coverage, path coverage, Length-i path coverage, loop coverage, relational operator coverage, table coverage.

A Test Data Adequacy Criterion C is a function C: $P \times S \times T \rightarrow \{true, false\}$. C(p, s, t)= true means that t is adequate for testing program p against specification s according to the criterion C, otherwise t is inadequate. There are various ways to classify adequacy criteria. Some of them are described as follows:

a) **Specification Based**: It specifies the required testing in terms of identified features of the specification or the requirements of the software, so that a test set is adequate if all the identified features have been fully exercised. In software testing literature it is fairly common that no distinction is made between specifications and requirements. It includes coveragibility of critical and non-critical requirements.
b) **Program Based**: It specifies testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised.

Another way of categorizing test adequacy criteria is as follows:

a) **Structural Testing**: It specifies testing requirements in terms of the coverage of the particular set of elements in the structure of the program or the specification.
b) **Fault Based Testing**: It focuses on detecting faults in the software. An adequacy criterion of this approach is some measurement of the fault detecting ability of test sets.
c) **Error Based Testing**: It requires test cases to check the program on certain error-prone points according to our knowledge about how program typically depart from their specification. Following table describes the adequacy criteria's for test case optimization.

Table 1: Adequacy Criteria's for Test Case Optimization

S.no	Adequacy criteria	Sub criterion
1	Fitness Oriented Adequacy Criterion	Code Coverage Data Flow Based Control Flow Based Execution Time Fault Detection Capability Client Requirement Fault Localization
2	Cost Oriented Adequacy Criterion	Cost Benefit Analysis Execution Cost Design Cost Setup Cost Data Access Code Third Party Cost
3	Project Change Volatility Adequacy Criterion	Rate of Change in Requirements Increase in Testing Efforts

Chapter

2

Literature Review

Ant Colony Optimization (ACO) [2] is a meta-heuristic approach used for solving Combinatorial Optimization Problems (COP's). It has also been successfully used to solve many NP hard optimizations problem. Artificial ants have now been successfully applied on a considerable number of applications leading to world class performances for problems like vehicle routing, quadratic assignment, scheduling, sequential ordering, routing in internet like networks and more.

Rothermel*et al.* [3] has addressed the issues related to prioritization for large software development environments. He made an empirical study for analysing the impact of size reduction test suite on fault detecting capability of test case. They used Siemens suite in their study and later expanded this to space. Prioritization of test cases based on historical execution of test data has been proposed by Kim *et al.* [4] to reduce the cost of regression testing. If the number of test cases selected by a Regression Test Cases Selection (RTS) is too large or if the execution cost are too high, then the selected test cases may have to be further prioritized. Under certain conditions, some can even guarantee that the selected test cases perform no worse than the original test suite. They prioritized the test cases and exercised only that fit within existing constraints. They pointed out that the existing prioritization techniques are memory less, implicitly, local choices. Li *et al*. [5] applied various meta-heuristics for test case prioritization, hill climbing algorithm, a genetic algorithm, a greedy algorithm, the additional greedy algorithm and a two-optimal greedy algorithm. There are several works that address the test case prioritization problem with search based algorithms, but there are few studies related to the application of ACO to this problem.

Walcott *et al*. [6] proposed a technique for test case prioritization based on genetic algorithm that reorders the test suite considering two objectives: time constraints for testing and code coverage. The objective function implements Block Average Percentage Coverage (APBC) metric. The proposed technique performed better than other existing techniques for a case study. For another case study, this did not happen because the test cases of this case study were more interchangeable. Walcott *et al*. [6] proposed algorithm with algorithm greedy, additional greedy, 2-optimal, hill climbing and genetic algorithm for test case optimization. They considered the following three coverage metrics separately (three single objective approaches): Average Percentage Block Coverage (APBC), Average Percentage Decision Coverage (APDC) and Average Percentage Coverage Statement (APCS). They concluded that genetic algorithm is better than others for small programs and additional greedy, two-optimal algorithm are good for large programs.

Singh *et al.* [7] present an approach to the test case prioritization problem based on run time and fault detection rate. In this approach, there is a time constraint which implies that this approach relates to the test case selection problem, not the test case prioritization problem. The authors considered a set of test cases and a set of test faults, where each test case covers one or more faults. The proposed ACO based algorithm consider n ants, where n is a number of test cases. The initial vertex is chosen randomly and the edges of the graph to be covered are randomly chosen by the ants among ones having maximum pheromone. The results obtained by the proposed approach are similar to the technique called optimal fault coverage, and superior to random order, reverse order and no order techniques.

Suri*et al*. [8] also considered the runtime information and fault detection of test cases. The number of ants is the number of test cases and each ant is placed at different vertex (test case) at the beginning of the execution. After initialization, the next vertex is chosen probabilistically according to the heuristic function (maximize the number of faults detected by the test case and minimize the execution time) and pheromone previously deposited by ants. The selection process of vertices is performed until all faults are covered by test cases already in the test suite. After updating the pheromone and choosing the best solution, the algorithm checks the runtime restriction. If the solution is not valid, the whole process is performed again. The proposed approach has reduced by 62.5% the size of the test suites to be performed, because it was considered that a fault can be found by one or more test cases. The proposed approach was not compared with other approaches.

There are two common features to the last two approaches described above: the number of ants and the fact that they do not consider the precedence among the test cases. When considering the number of ants as the number of test cases, the execution of the algorithm may become slow as the number of test cases increases, since there are more ants sharing pheromone information. Moreover, the approaches do not address the precedence among test cases, which is common in real world applications.

Chapter

3

Gaps and Objectives

3.1 Gaps in Present Study

Test case prioritization is a critical problem of software testing. Since several factors may be considered in order to find the best order for test cases, search based techniques have been applied to find solutions for test case prioritization problem. Some of these works apply ant colony based algorithm, but the Statement Coverage and Precedence of test cases were not considered. Previous work on regression test case prioritization has focused on greedy algorithm. However, it is known that these algorithms may produce suboptimal results because they may construct results that denote only local minima within the search space. Meta-heuristic and evolutionary search algorithms aims to avoid such problems.

Genetic algorithm to reorder test cases in a test suite using execution time as a constraint had shown that prioritization technique is appropriate for man regression testing environment and explains how the baseline approach can be extended to operate in additional time constrained testing circumstances.

Most of the researchers had explored genetic algorithm, ant colony optimization, linear programming etc. based approaches to find out the subset of test cases from a large pool of test cases. On the basis of fault detecting capability many interesting results have been received but the test case prioritization based on statement coverage using Ant Colony Optimization technique has not been explored. So there is still space for the researchers to experiment and validate the ant colony optimization based approach to find out the order of test cases on the basis of statement coveragibility.

3.2Objectives

On the basis of gaps identified, following are the objectives of my thesis:-

- To study the various existing techniques for test case prioritization.
- To investigate Ant Colony Optimization technique for software test case prioritization.
- Propose framework for test case prioritization based on statement coverage using ACO algorithm.

Chapter

4

Test Case Prioritization using Ant Colony Optimization

4.1 Ant Colony Optimization (ACO)

ACO technique is a set of instructions based on search algorithms of artificial intelligence for near to optimal solutions, here the iconic member is ANT system, as proposed by Dorigo, Colorni*et al.* and Maniezzo*et al.* [9,10,11]. Ants are blind and small in size and still are able to find the shortest route to their food source. They make the use of chemical substances, secreted by them known as pheromones and antennas to communicate with each other. ACO inspired from the behaviour of live ants which is capable of synchronization with searching solutions for local optimization problem by maintaining array list for maintaining previous information gathered by each ant.

ACO mainly deals with two important processes namely, pheromone deposition and trail pheromone evaporation [12]. Initially pheromone deposition is taken arbitrarily and pheromone evaporation is taken as per equation (3). Pheromone deposition is a phenomenon of ants adding the pheromone on all paths they follow. Pheromone trail evaporation means decreasing the amount of pheromone deposited on every path with respect to time. Updation of the trail is performed when ants either complete their search or get the shortest path to reach the food source. Each combinatorial problem defines its own updating criteria depending upon its own local search and global search respectively.

Artificial ants leave a virtual trail accumulated on the path segment they follow. The path for each ant k is selected on the amount of pheromone trail present on the possible paths starting from the current node of the ant as per equation (1).

$$\rho_{ij}^{k}(t) = \tau_{ij}^{\alpha} \,.\eta_{ij}^{\beta} \;/\; \sum \tau_{ij}^{\alpha} \,.\eta_{ij}^{\beta} \;\; ; \;\; j \notin \zeta \qquad \textbf{. . . (1)}$$

$$\rho_{ij}^{k}(t) = 0 \;\; ; \text{otherwise}$$

where,

τ_{ij}^{α} is the pheromone level on the path (i , j) of the graph .

η_{ij}^{β} is the heuristic function

ζ is the array list (memory of ant) containing all the trails that ant has already passed and must not be chosen again .Heuristic function η is used to search for the optimal path with the help of useful information, and the array list ζ stores the previous paths visited in the memory. Heuristic information h_{ij} can be calculated using equation (2).

$$h_{ij} = 1 \Big/ \eta_{ij}^{\beta} \quad \textbf{. . . (2)}$$

Updating the trail is performed when ants complete their search of food source. The updating of pheromone is achieved by using equation (3).

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho \sum_{k=1}^{g} \Delta_{ij} \quad \textbf{. . . (3)}$$

Where ρ is pheromone trail evaporation that lies in the interval [0, 1) and $\sum_{k=1}^{g} \Delta\tau_{ij}$ are pheromone deposited in the trail (i, j) followed by g ants after completing a tour. In case of equal or no pheromone on adjacent paths, ants randomly choose the path. Pheromone trail on a path increases the probability of the path being followed. Ant then reaches the next node and again does the path selection process as described above. This process continues till the optimal path is achieved. These final tours give the solution for shortest or best path which can be analysed for optimality.

Thus, ants are able to find shortest path on the basis of:

- The pheromone laid on ground by other ants of same colony which represents the experience of the colony.
- Memory and heuristic information which represents useful knowledge about the particular problem the ants are solving.

4.2 Test Case Selection and Prioritization using ACO

The proposed test case prioritization technique using ant colony optimization within a time restricted frameworkis implemented and evaluated. The technique uses the statement coverage and execution time information of the regression test suite as an input. In the proposed algorithm statement coveragability acts as a cost of execution.

The basic block diagram for the ACO_TEST (Ant Colony Optimization for Test Case Selection and Prioritization system (ACO_TEST) is shown in Figure 1. The inputs to the system include details of the test suite i.e the test cases along with the statement covered by them and their execution time. These inputs are generally tabulated and are entered by the tester. The output then produced has path details for each iteration, pheromone details, best path details and the final selected and prioritized test suite.

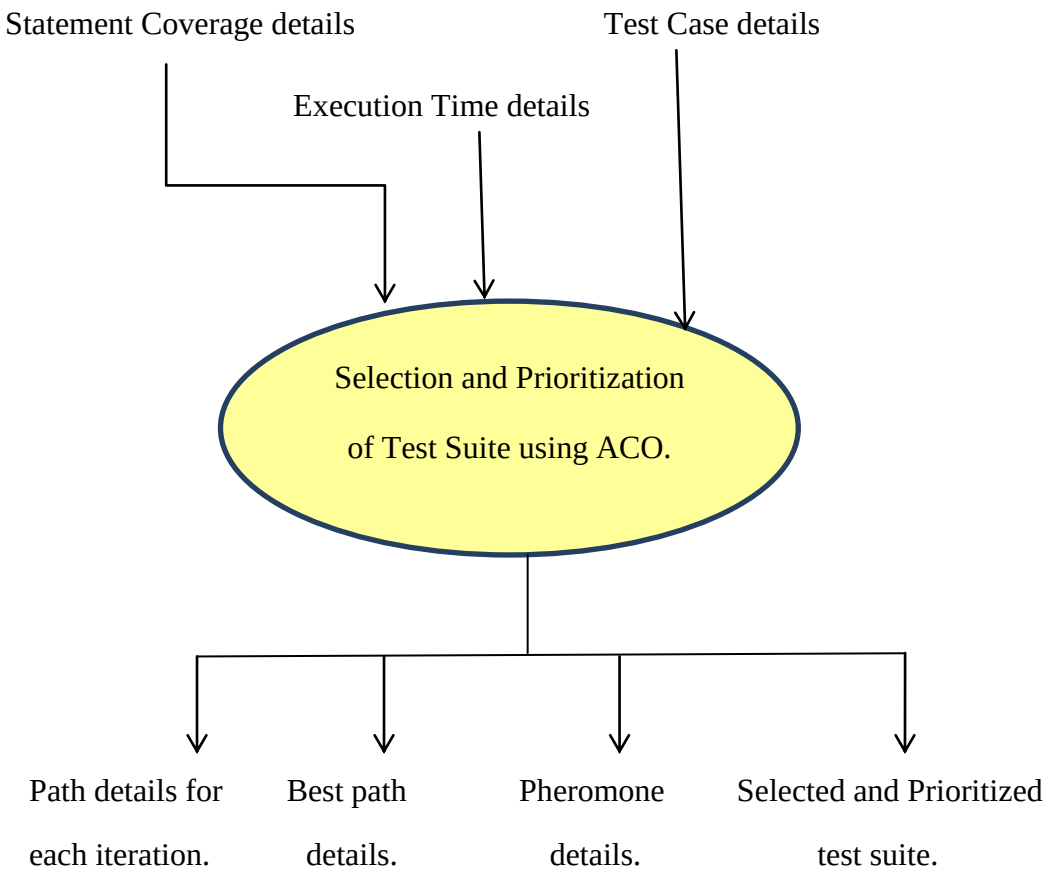

Figure 1: Block Diagram of ACO_TEST.

4.3 Statement of the Problem

Let T be a test suite containing n elements and T= $\{t_1, t_2, t_3, \ldots\ldots t_n\}$ be a sequence on T. let F be a function from test suite elements to some domain on which the relation $\geq$ imposes a total order. T' is a test case prioritization of T with respect to F if and only if for all i, $1 \leq i \leq n\text{-}1$ such that $F(t_i) \geq F(t_{i+1})$, that is F is monotonic over T.

4.4 Problem Formulation

A selective regression testing technique chooses a subset of test suite that was used to test the software before modifications were made and then uses this subset to test the modified software. Prioritization is the process of scheduling test cases in an order to meet some performance goal.

We define a test suite T as a n-tuple of test cases t_i, where i = 1to n and T= { t_1, t_2, t_3,……t_n}. The prioritization of T is denoted as T'. In the problem formulation, the maximum time within which a prioritize test suite must execute is the maximum capacity of test suite, execution time of each test case is its weight and its value is percentage of statement coverage. The output of algorithm is prioritized list that fits the required time limit. Formally test case prioritization is defined as equation (4).

Maximize: $\sum_{i=1}^{i=n} s_i x_i$

Subject to: $\sum_{i=1}^{i=n} et_i x_i <= et_{max}$; $x_i = 0 \text{ or } 1$ **. . . (4)**

where,

s_i: Statement coverage.

et_i: Execution time of test case t_i

et_{max}: Maximum time allowed for the execution of the prioritization.

Since Test Case Prioritization is an NP-complete problem [13] and all NP complete problems are NP hard. Hence, Ant colony optimization is used as a method for solving hard combinatorial optimization problems (COP's) [14].

Following are the assumptions made in the problem stated above

- Original test suite is taken as T={ t_1 , t_2 , t_3 , ……. , t_n}
- Set of all statements is defined as St ={s_1 , s_2 , s_3……. ,s_x}
- Each test case t_i, where i=1to n in the original test suite covers some or all the statements.
- Initial weight is taken arbitrary.
- Number of artificial ants to search through the test case space is n (number of test cases).
- For each ant j, a list that contains selected test cases (and thus the statements covered on the current path) is represented as S_j = { s_1 , s_2 , s_3 …….. ,s_m} where, m ≤ n.
- w_i is the weight of each edge i, which is assumed to be the amount of pheromone deposited on the edge.
- Assume pheromone deposition rate to be +1 or 100% for each ant that has crossed the edge on a best path.
- Assume pheromone evaporation rate to be k% of w_i (current weight of i^{th} edge) to be reduced for each edge after each iteration of the loop.

4.4.1 Problem Representation

The problem can be represented in the form of a undirected graph G (V, E) where V is the set of vertices and E is the set of edges in the graph. V≡T i.e, test cases are represented by vertices in the graph. Each edge in the graph represents the pheromone trail associated with the edge $e_i \in E$, which reflects the amount of statement coverage $s_i$on the chosen path within time constraint, TC.

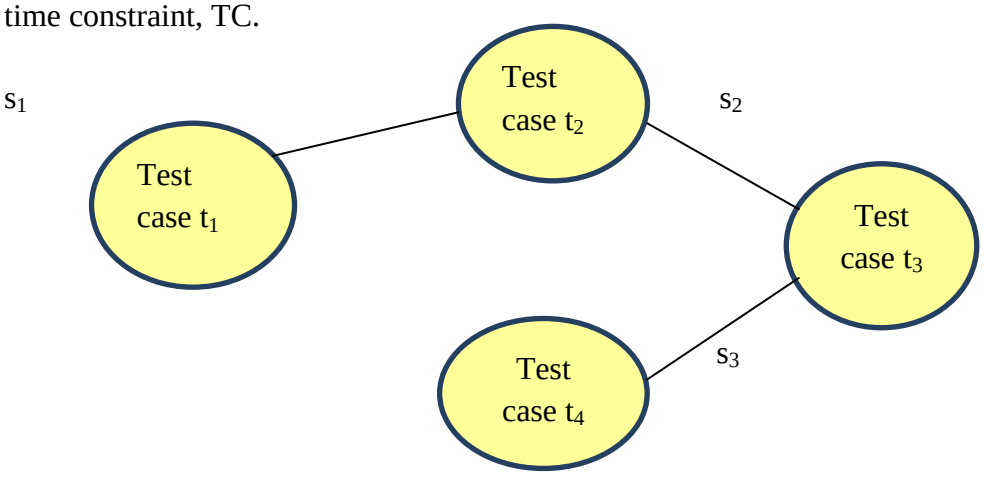

Figure 2: Problem Representation

4.4.2 Process diagram for the ACO_TEST system

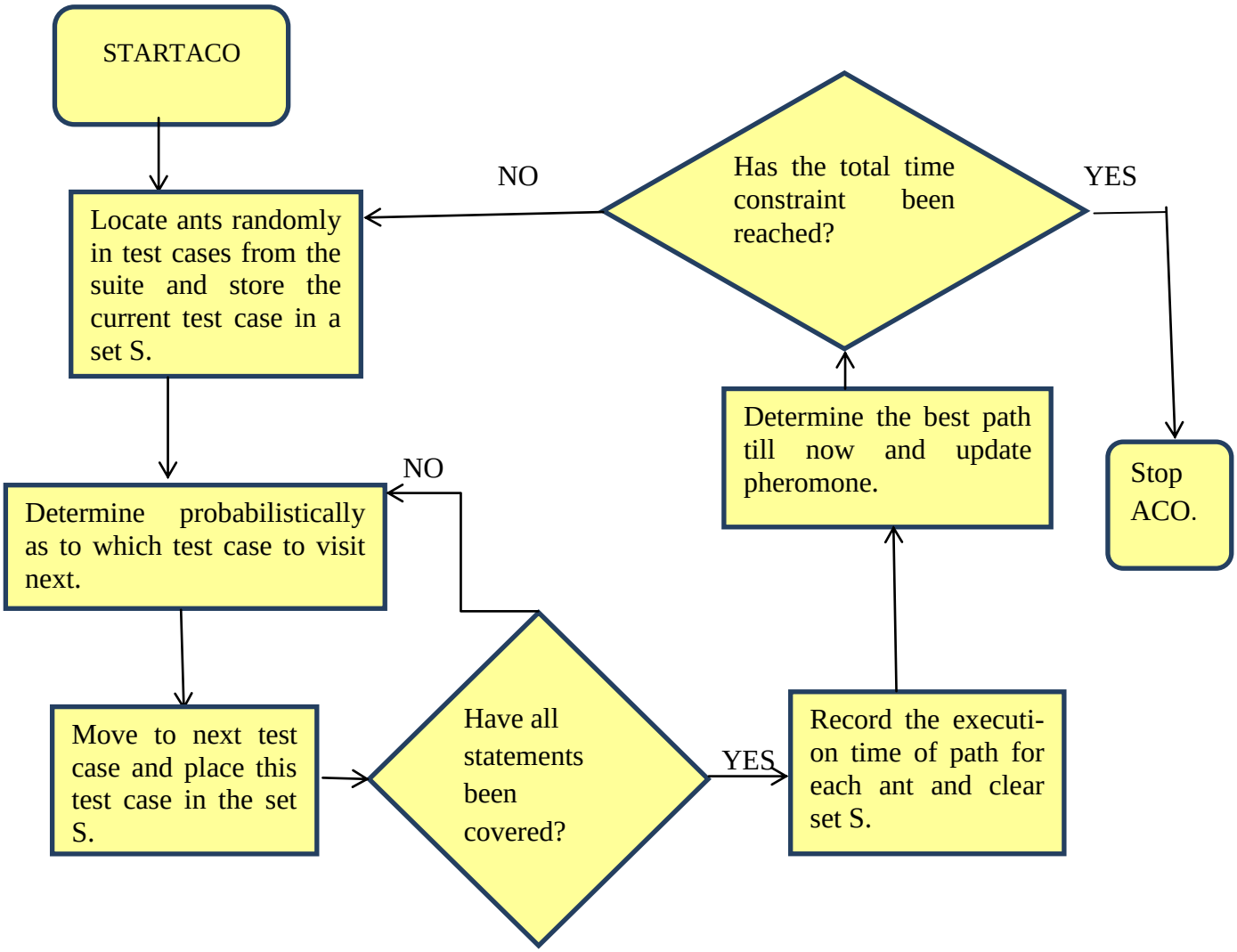

Figure 3: Flow Chart for ACO_TEST System

In, the above process, ACO has four modules namely ants_cost, ants_primary placing, ants_traceupdating and ants_cycle. ants_information provide initialization of the variables, ants_primaryplacing initially places ants randomly in the test cases, ants_cost selects edges on the basis of attractiveness which depends upon the density of pheromones and statement coverage and ants_traceupdating updates the pheromone trails.

4.5 Case Study

Here to explain the process we consider a test suite with 7 test cases in it, covering 163 statements of program (printtokens2) under test. Test prioritization schemes typically create reordering of the test cases in the test suite that can be executed after many subsequent changes to the program under test.

Each test case covers the statements as shown in table below:

Table 2: Sample Test Cases and Statements executed in Program under Test.

Test Case/Statement	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S1	1	1	1	1	1	1	1
S2	1	1	1	1	1	1	1
S3	1	1	1	1	1	1	1
S4	1	1	1	1	1	1	1
S5	1	1	1	1	1	1	1
S6	1	1	1	1	1	1	1
S7	1	1	1	1	1	1	1
S8	1	1	1	1	1	1	1
S9	1	1	1	1	1	1	1
S10	1	1	1	1	1	1	1
S11	1	1	1	1	1	1	1
S12	1	1	1	1	1	1	1
S13	1	1	1	1	1	1	1
S14	1	1	1	1	1	1	1
S15	1	1	1	1	1	1	1
S16	1	1	1	1	1	1	1
S17	1	1	1	1	1	1	1
S18	1	1	1	1	1	1	1
S19	1	1	1	1	1	1	1
S20	1	1	1	1	1	1	1
S21	1	1	1	1	1	1	1
S22	1	1	1	1	1	1	1

S23	1	1	1	1	1	1	1
S24	1	1	1	1	1	1	1
S25	1	1	1	1	1	1	1
S26	1	1	1	1	1	1	1
S27	1	1	1	1	1	1	1
S28	1	1	1	1	1	1	1
S29	1	1	1	1	1	1	1
S30	1	1	1	1	1	1	1
S31	1	1	1	1	1	1	1
S32	1	1	1	1	1	1	0
S33	1	1	1	1	1	1	1
S34	1	1	1	1	1	1	1
S35	1	1	1	1	1	1	1
S36	1	1	1	1	1	1	1
S37	1	1	1	1	1	1	1
S38	1	1	1	1	1	1	1
S39	1	1	1	1	1	1	1
S40	1	1	1	1	1	1	1
S41	1	1	1	1	1	1	1
S42	1	1	1	1	1	1	1
S43	1	1	1	1	1	1	1
S44	1	1	1	1	1	1	1
S45	1	1	1	1	1	1	1
S46	1	1	1	1	1	1	1
S47	0	0	0	0	0	1	0
S48	0	0	0	0	0	1	0
S49	0	0	0	0	0	1	0
S50	1	1	1	1	1	1	1
S51	1	1	1	1	1	1	1
S52	1	1	1	1	1	1	1
S53	1	1	1	1	1	1	1
S54	1	1	1	1	1	1	1

S55	1	1	1	1	1	1	1
S56	0	0	0	0	0	1	0
S57	0	0	0	0	0	1	0
S58	0	0	0	0	0	1	0
S59	1	1	1	1	1	1	1
S60	1	1	1	1	1	1	0
S61	1	1	1	1	1	1	0
S62	1	1	1	1	1	1	0
S63	1	1	0	0	1	1	1
S64	1	1	0	0	1	1	1
S65	1	1	0	0	1	1	1
S66	1	1	1	1	1	1	1
S67	1	1	1	1	1	1	1
S68	1	1	1	1	1	1	1
S69	1	1	1	1	1	1	1
S70	1	1	1	1	1	1	1
S71	1	1	1	1	1	1	1
S72	1	1	1	1	1	1	1
S73	1	1	1	1	1	1	1
S74	1	1	1	1	1	1	1
S75	0	0	0	0	0	1	1
S76	1	1	1	1	1	1	1
S77	1	1	1	1	1	1	1
S78	1	0	1	1	1	1	1
S79	0	0	0	0	0	1	1
S80	1	1	1	1	1	1	1
S81	1	1	0	0	1	0	0
S82	1	1	1	1	1	1	1
S83	1	1	1	1	1	1	1
S84	1	1	0	0	1	1	0
S85	1	1	1	1	1	1	1
S86	0	1	0	0	1	0	0

S87	1	1	1	1	1	1	1
S88	1	1	1	1	1	1	1
S89	1	1	1	1	1	1	1
S90	1	1	1	1	1	1	1
S91	1	1	0	1	1	1	1
S92	1	1	1	1	1	1	1
S93	1	1	1	1	1	1	1
S94	1	1	1	1	1	1	1
S95	0	0	0	0	0	0	1
S96	1	1	1	1	1	1	1
S97	0	0	0	0	0	0	1
S98	1	1	1	1	1	1	1
S99	1	1	1	1	1	1	1
S100	1	1	1	1	1	1	0
S101	1	1	1	1	1	1	1
S102	1	1	1	1	1	1	1
S103	1	1	1	1	1	1	1
S104	1	1	1	1	1	1	1
S105	1	1	0	0	0	0	0
S106	1	1	1	1	1	1	1
S107	1	1	1	1	1	1	1
S108	1	1	1	1	1	1	1
S109	1	1	1	1	1	1	1
S110	1	1	1	1	1	1	1
S111	1	1	1	1	1	1	1
S112	1	1	1	1	1	1	1
S113	0	1	0	0	1	0	0
S114	0	1	0	0	1	0	0
S115	1	1	1	1	1	1	1
S116	1	1	1	1	1	1	1
S117	1	1	1	1	1	1	1
S118	1	1	1	1	1	1	1

S119	0	0	0	0	0	1	0
S120	0	0	0	0	0	1	0
S121	0	0	0	0	0	1	0
S122	0	0	0	0	0	1	0
S123	1	1	1	1	1	1	1
S124	1	1	1	1	1	1	1
S125	1	1	1	1	1	1	1
S126	1	1	1	1	1	1	1
S127	1	1	0	0	1	1	0
S128	1	1	0	0	1	0	0
S129	1	1	0	0	1	0	0
S130	1	1	0	0	1	1	0
S131	1	1	1	1	1	1	1
S132	0	1	1	0	1	1	0
S133	0	1	1	0	1	1	0
S134	0	0	0	0	1	0	0
S135	0	0	0	0	1	0	0
S136	0	1	1	0	0	1	0
S137	0	0	0	0	0	1	0
S138	0	0	0	0	0	1	0
S139	0	1	1	0	0	0	0
S140	0	1	0	0	0	0	0
S141	0	1	0	0	0	0	0
S142	0	1	1	0	0	0	0
S143	0	0	1	0	0	0	0
S144	0	0	1	0	0	0	0
S145	0	1	0	0	0	0	0
S146	0	1	0	0	0	0	0
S147	0	1	0	0	0	0	0
S148	0	1	0	0	0	0	0
S149	0	1	0	0	0	0	0
S150	0	1	0	0	0	0	0

S151	1	1	1	1	1	1	1
S152	1	1	1	1	1	1	0
S153	0	0	0	0	1	0	0
S154	1	1	1	1	1	1	1
S155	1	1	1	1	1	1	1
S156	0	1	0	0	0	0	0
S157	1	1	1	1	1	1	1
S158	1	1	1	1	1	1	1
S159	0	1	0	0	0	0	0
S160	1	1	1	1	1	1	1
S161	0	1	0	0	0	0	0
S162	1	1	1	1	1	1	1
S163	1	1	1	1	1	1	1

Regression test suite contains seven test cases with the initial ordering { t_1, t_2, t_3, t_4, t_5, t_6, t_7 }. Following table illustrates the number of statements covered by each test case and the execution time .

Table 3: Sample Test Cases and Statements Executed in the Program Under Test.

Test Case	Number of Statements Covered	Execution time (seconds)
t_1	123	4
t_2	142	3
t_3	120	5
t_4	113	4
t_5	130	3
t_6	136	3
t_7	116	3

Chapter

5

Experimental Study

5.1 Inputs for Experimental Study

For the experiment, out of the complete test pool randomly a test suite of fourteen test cases has been chosen. Here we calculate statement coverage percentage and execution time of each test case. Statement Coveragebility and execution time can be computed as follows:

gcov:gcov is a test coverage program. We compile the printtokens2 program (program under test) with the help of gcc compiler. We used gcov a profiling tool to help discover where our optimization efforts will best affect our code. We can also use gcov along with the other profiling tool such as gprof. Profiling tool helps us to analyse our code's performance. Using a profiler such as gcov, some basic performance statistics, are found such as:

- How often each line of code executes.
- What lines of code are actually executed.
- How much computing time each section of code consumes.

gcov creates a logfile called sourcefile.c.gcov which indicates how many times each line of a sourcefile.c has executed. gcov works only on code compiled with gcc. It is not compatible with any other profiling or test coverage mechanism. The .gcov files contain the ':' separated fields along with program source code. The format is as follow:

execution_count: line_number: source line text

Additional block information may succeed each line, when requested by command line option. If the execution_count is '5' it means the statement is executed 5times. Execution count for unexecuted lines are marked '#####' or '====='. When using gcov, we first compiled our program with two special gcc option '-fprofile-arcs' and '-ftest-coverage'. This

tells the compiler to generate additional information needed by gcov and also includes additional code in the object files for generating the extra profiling information needed by gcov. These additional files are placed in the same directory where the object file is located. Running the script file will cause profile output to be generated. For each source file compiled with –fprofile-arcs, an accompanying .gcda and .gcno file will be placed in the object file directory.

gcov data files:gcovuses two files for profiling. The names of these files are derived from the original object file by appending the file name suffix with .gcno or .gcda. All of these files are placed in the same directory as the object file, and contain data stored in a platform-independent format. It contains information to reconstruct the basic block graphs and assign source line numbers to blocks. It contains arc transition counts, and some summary information. The full details of the file format is specified in gcov-io.h, and functions provided in that header file should be used to access the coverage files.

Time : Time command is used to find the execution time. Format of this command is as:

Time [options] command [arguments….]

The time command runs the specified program command with the given arguments. When command finishes, time writes a message to standard output giving timing statistics about the program run. These statistics consists of:

- The elapsed real time between invocation and termination.
- The user CPU time.
- The system CPU time.

5.2 Program Under Test

Program that is taken for experimentation was printtokens2. It has been taken from Software-Artifact Infrastructure Repository (SIR) and is a Lexical Analyser. It is a benchmark project and freely available at www.sir.unl.edu [15]. This program has been implemented in C and consists of 19 procedures, 570 Line of Code (LOC). It has been originally created by Tom Ostrand and colleagues at Siemens Coporate Research and its version is SIRV: 2.0. It consists of 4115 test cases, out of which 1000 test cases are used for implementation part.

5.3 Implementation

The algorithm has been named as ACO_TEST and is implemented in MATLAB programming according to the proposed algorithm. Program has been compiled and run with the inputs (Statement coveragability and execution time for each test case). We have taken hundred iterations in the program, but in order to explain the implementation part first and last iteration has been considered. It has main program and 6 modules namely ants_cycle( ), ants_information( ), ants_cost( ), ants_primaryplacing( ), ants_traceupdating and search. Main program calls ants_information( ) which provides initialization of variables of the algorithm. Subsequently, ants_primaryplacing( ) that initially places the ants randomly. Subsequently, it calls ants_cycle( ) that is used to select the edges on the basis of attractiveness which depends on the density of pheromones and statement coverage. Then ants_traceupdating is called whichs updates the pheromone trails. Implementation of code in MATLAB has led to the following results:

Table 4: Order of Test Cases in First and Last Iteration

Iteration	Order of Test Cases													
First	t_6	t_2	t_{12}	t_{14}	t_3	t_{11}	t_{10}	t_4	t_9	t_7	t_1	t_{13}	t_5	t_8
Last	t_3	t_1	t_{13}	t_5	t_{12}	t_6	t_2	t_8	t_{14}	t_{10}	t_{11}	t_9	t_7	t_4

As per the above table, pictorial representation for the first iteration is shown in Figure 4.

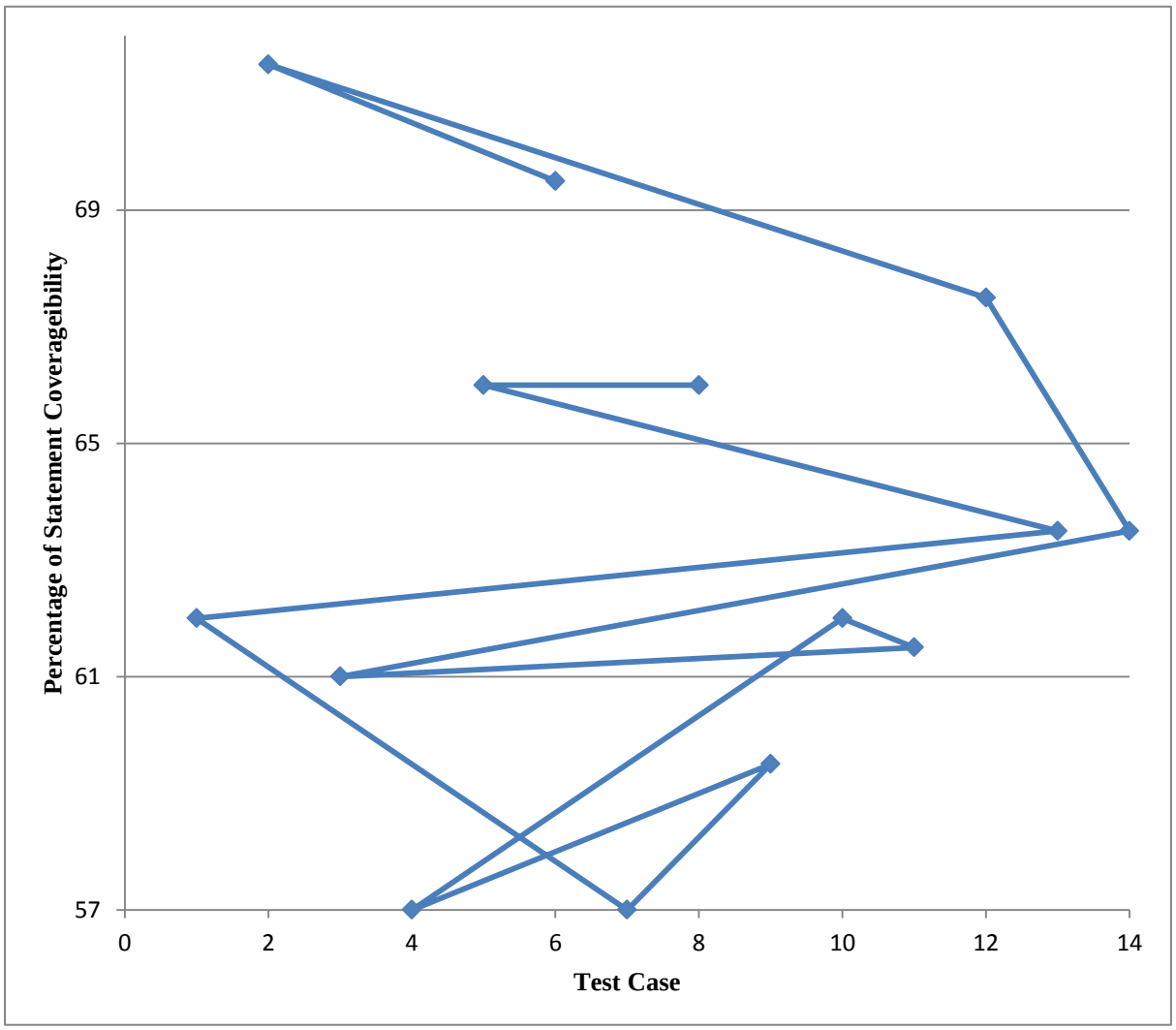

Figure 4: Test Case Prioritization for First Iteration.

Similarly, as per the above table, pictorial representation for the last iteration is shown in Figure 5.

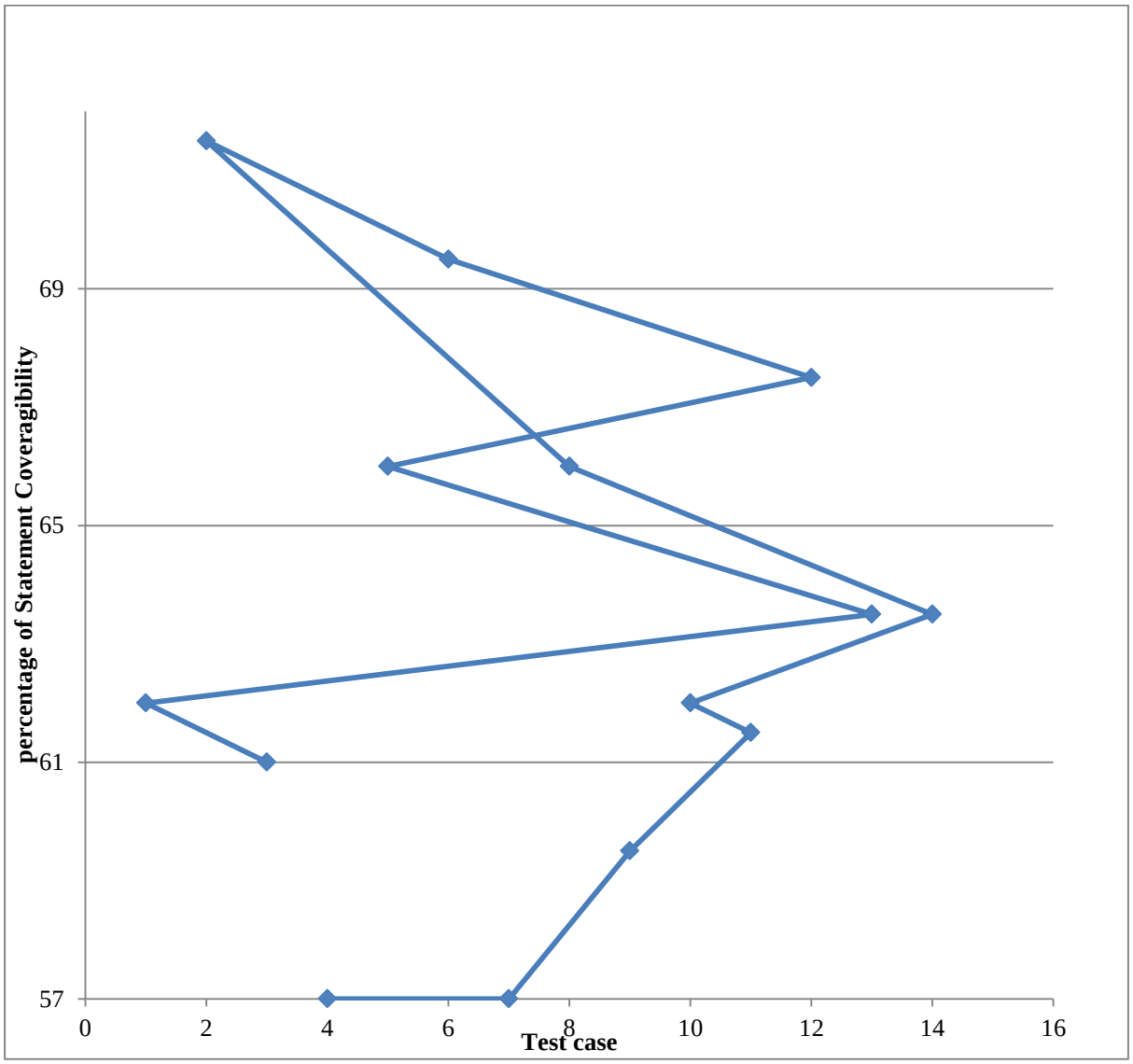

Figure 5: Test Case Prioritization for Last Iteration.

Chapter

6

Conclusion

We have proposed Selection and Prioritization approach based on ACO considering the Statement coveragability and execution time of test cases and obtained the solution that is nearest to optimal. Proposed work gave encouraging results. ACO is strong and robust as it involves positive feedback and parallel computations and hence, it can lead to better solutions in optimum time. Multi-objective Test Case Prioritization using Ant Colony Optimization will be explored in future.

References

[1] Aggarwal, K. K. and Singh, Y., Software Engineering, New Age International Publishers, 2005.

[2] Dorigo, M., Maniezzo, V. and Colorni, A., The Ant System: Optimization by a Colony of Cooperating Agents, IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, Vol. 26, pp:29-41, 1996.

[3] Rothermal, G., Untch, R.H., Chu, C. and Harold, M., Test Case Prioritization, IEEE Transactions on Software Engineering , Vol. 27, No:10, pp:928-948, Oct. 2001.

[4] Kim, J. M. and Porter, A., A History Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments, Proceedings of the 24th International Conference on Software Engineering, pp:119-129, 2002.

[5] Li, Z., Harman, M., and Hierons, R. M., Search Algorithms for Regression Test Case Prioritization, IEEE Transactions on Software Engineering, Vol. 33, No:4, April 2007.

[6] Walcott, K. R., Soffa, M. L., Kapfhammer, G. M., Roos, R. S., Time-Aware Test suite Prioritization, Proceedings of the International Symposium on Software Testing and Analysis, pp:1-12, 2006.

[7] Singh, Y., Kaur, A., Suri, B., Test Case Prioritization using Ant Colony Optimization, Software Engineering Notes (ACM SIGSOFT), Vol. 35, No:4, pp:1-7, July 2010.

[8] Suri, B., Singhal, S., Implementing Ant Colony Optimization for Test Case Selection and Prioritization, Proceedings of International Journal on Computer Science and Engineering, vol.3, No:5, pp:1924-1932, 2011.

[9] Dorigo, M., Optimization, Learning and Natural Algorithms, Ph. D. Thesis, Politecnicodimilano, Milano, 1992.

[10] Colorni, A., Dorigo ,M. and Maniezzo, V., Distributed Optimization by Ant Colonies, Proceedings of European Conference on Artificial Life (ECAL 91), Elsevier Publishing, Amsterdam, 1991.

[11] Manizzeo, V., Dorigo, M., and Colorni, A., The Ant System: an Autocatalytic Optimizing Process, Technical Report TR91-016, Politecnico di Milano, 1991.

[12] Alsapaughy, S., Walcotty, Kristen, R., Belanichz, M., Kapfhammerz, Gregory,M., and Soffay, Marry L., Efficient Time-Aware Prioritization with Knapsack Solyer, ACM , Newyork, USA, pp: 13-18, 2007.

[13] Rothermal, G., Untch , R.J. and Chu, C., Prioritizing Test Cases for Regression Testing, Proceedings of IEEE Transaction on Software Engineering ,Vol. 27, No:10, pp:929-948, 2001.

[14] Dorigo, M. and Socha, K., An Introduction to Ant Colony Optimization, CRC Press, 2007.

[15] URL:http://sir.unl.edu/portal/index.php.

Abbreviations

ACO	Ant Colony Optimization
DFC	Data Flow Coverage
BVA	Boundary Value Analysis
APBC	Average Percentage Block Coverage
APDC	Average Percentage Decision Coverage
APCS	Average Percentage Coverage Statement
WBT	White Box Testing
BBT	Black Box Testing
COP'S	Combinatorial Optimization Problems
RTS	Regression Test Cases Selection
SIR	Software Artifact Infrastructure Repository
LOC	Line of Code