Automatic Test Data Generation-Achieving Optimality Using Ant-Behaviour

Sumesh Agarwal, Shubham Gupta, and Nitish Sabharwal

***Abstract*—Software Testing is an important element in the Software Development Life Cycle which helps in the validation of the software. It should be made sure that before release, the software is free of errors and produces the desired outputs. This paper proposes a method for Behavioral Testing of a software under test (SUT) using the concept of Ant Colony Optimization for test data generation.**

***Index Terms*—Ant colony optimization, software testing, test data, test sequence, graph coverage, behavioral testing.**

I. Introduction

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.It remains the primary technique used to gain consumers' confidence in the software and the process of testing any software system is an enormous task which is time consuming and costly [1]. Software Development Life Cycle is the process of developing information systems through investigation, analysis, design, implementation and maintenance and it includes the processing steps for development of any software product [2]. The inputs for a specific stage are the outputs of the preceding step. In SDLC, software testing is more complex and an important issue [1].

Software Testing can be defined as a process of assessing the functionality and correctness of a program through execution or analytical methods. Thus, software testing could ensure that a program meets its intended specifications and requirements. Karen *et al*. [3] have noted that it is impossible to completely test an application because: 1) input range of the software may be very complex and large, 2) verity of input paths are exists in software, and 3) modelling and designing structure and its issues related to its specifications are difficult to test [2]. They point out the shortcomings of any Software Testing method deployed. But nevertheless, in spite of all these limitations, a proper software testing process definitely improves the quality of the software under consideration.

Software Testing is broadly of two types, black box testing and white box testing [1], [2]. While white box testing focuses on the structure (program) of the software under test (SUT), black box testing focuses on the functionality of the system. This paper proposes a new method for optimized test-sequence generation for black box testing of an SUT to derive sets of input conditions that will fully exercise all functional requirements of that SUT.

Manuscript received August 10, 2014; revised October 18, 2014.

The authors are with the Birla Institute of Science and Technology, Pilani, India (e-mail: agarwal.sumesh@gmail.com, shubham2490@gmail.com, nitishsab@gmail.com).

The algorithm to achieve optimality in the test data generation [4] is inspired by the behaviour of ants living in colonies. Li and Lam [5] have proposed an Ant Colony Optimization (ACO) [6]-[8] approach to test data generation for state-based software testing [1]. Ghiduk has also employed Ant Colony Optimization (ACO) algorithms for the case of software data-flow testing [9]. This paper proposes a method of optimal test-sequence generation using ACO on control flow graph (CFG) [1], [2] of the SUT to construct a test suite for that SUT.

The paper is organized as follows: Section II details previously done work in the area of software testing and introduces the ACO technique, Section III gives an overview of ACO using it to generate test data. In Section IV, the proposed algorithm for test data generation is explained in detail which is further analyzed in Section V with the help of case studies, in which comparisons are drawn between the proposed algorithm and other approaches of test data generation by other researchers, with the help of illustrations. Conclusion and ideas for future work are discussed in Section VI.

II. Background

Software testing is necessary for the delivery of reliable and quality oriented software product, more satisfied users, lower maintenance cost, and more accurate and reliable results. Hence it is an important activity of SDLC. The importance of testing can be understood by the fact that "around 35% of the elapsed time and over 50% of the total cost are expending in testing programs" [10]. To optimize resources in the area of software testing, quality and reliability, academicians and researchers are using Artificial Intelligence [11] (AI) approaches for better accuracy. The application of AI techniques in Software Engineering (SE) is an emerging area of research. ACO is one of the AI techniques that is used extensively by researchers in software testing. For example, Karl Doerner and Walter J. Gutjahr [12] have tried to investigate methods for deriving a suitable set of test paths for a software system. They try to resolve the trade-off between coverage and testing costs. Their results have not been verified using more extensive experiments. Reasonable solutions are obtained after less number of iterations as compared with standard ACO but still the number is not small enough.

Li and Lam [5] have proposed an Ant Colony Optimization approach to test data generation for state-based software testing. Here the test sequence is generated automatically which they claim is always feasible, non-redundant and achieves all state coverage criterion. The main problem in their work is of the complete software coverage and moreover they do not factor in the criticality of states. Ghiduk [9] has employed Ant Colony Optimization algorithms for the case of software data-flow testing. The approach generated all the optimal paths to cover all du-pairs for the program under test. The major demerit in his work is that of the redundant paths obtained in the test data. As can be seen in the case study taken by him, there are redundant def-clear paths which are associated with a set of def-use pairs.

A Genetic Algorithm [13] based test data generation technique has also been proposed by Lefticaru [13]. In order to generate test data before coding, it uses UML state diagram for test data generation. The major drawback of this approach is that in this case, coverage of all transitions is not possible due to fitness function and their chromosome length. So it is not suitable of strong level software coverage. Srivastava [14], [15] has proposed a technique to generate optimized test sequences from a Markov chain based usage model. Major merits of his work are that this technique has proved to be capable of deriving test cases giving priority to the most critical states and transitions. Also, even with strict cost limitations it was able to cover the most critical states but in process of covering most critical states his method produces a lot of redundant paths. From these works and the approaches followed by them, it can be observed that the problems of redundant test cases and incomplete state coverage have not been completely tackled. In this paper, an algorithm based on the concept of ACO has been proposed which tries to solve both of these problems and aims to provide an efficient tool for software testing.

Ant Colony Optimization (or ACO) [6]-[8] is a technique which can help differentiate the "good paths" (or required paths) from the "bad paths" using heuristics. The technique is inspired from the foraging behaviour of ant species. Real ants are capable of finding shortest path to food source through stigmergic communication and without following visual cues [16]. Foraging ants deposit chemicals (pheromones) while walking in search of food source, thus increasing the probability of other ants following the same path. This communication is a pheromone mediated indirect communication, and the ants exploit the pheromone information to help them to find a shortest path in the search of food. So in a graphical problem, the behaviour of ants can be used to derive optimal paths [1], [2].

III. Test Data Generation Using ACO

Testing involves three main steps: generation of a set of test inputs, execution of those inputs on the program under tests, and then checking whether the test executions reveal faults [17], [18]. A Control Flow graph (CFG) or program graph represents the control flow of a program and is widely used in the analysis of software. The nodes of a control flow graph are statements of the program and the edges represent the control flow between the statements. CFGs have been widely studied for many years (Jalote, [19]; Kosaraju, [20]; McCabe, [21]; Paige, [22]; Rapps and Weyuker, [23]; White, [24]; Zhu *et al.* [25]) and they have been used for problem representation in our algorithm. This paper skips the part of modelling the SUT in the form of a CFG and focuses on how the ants can traverse the graph and generate an optimal test suite for that SUT.

For a given CFG, an ant is placed at a specific node which travels the rest of the graph until it reaches a "dead end" based on the idea that it travels from a node to another node only along that outgoing edge which has zero pheromone level. The reason for this choice is to cover maximum edges in the graph. The presence of pheromone at an edge signals the ant that edge has been traversed by some other ant, so it takes the one for which pheromone is not yet updated thereby increasing the graph coverage. Also, unlike the common ant colony behaviour, the evaporation of pheromone with time has not been taken into account in the proposed algorithm, as the pheromone levels are just an indicator to an ant as to which edges have been traversed, and therefore they need not be decreased with time. Any value of pheromone other than 0 indicates that the edge has been traversed and thus no ant should take up that edge.

The pheromone level for every feasible transition in the graph is initially set to zero. The transition from a node i to node j leads to an increment in the pheromone level of that edge by $\Delta\tau$, the pheromone update constant. An ant chooses path with the least pheromone level avoiding the case when it may re-traverse a path. Following sets are referred by an ant at a node n traversing the graph in the proposed algorithm. A number of ants can be used depending on the structure of the root set, which can get modified as the program progresses, to cover the CFG of the SUT.

IV. The Proposed Algorithm

Our approach uses as input a CFG for an SUT and gives the optimal test sequences to minimize the redundancy and maximize graph coverage. We define the following sets that are used in our algorithm:

- **Feasibility Set** = { F_n } contains those nodes for which the corresponding transitions from n to those nodes have a pheromone level of zero.
- **Root set =** { R_i } contains the nodes which specify the starts nodes for the ants in the I^{th} iteration.
- **Pheromone level matrix** = { *pLevel* } is a double-array of nodes for storing the pheromone level corresponding to every edge in the CFG.

The following section lists the conditions used in the algorithm for any ant *K* at a node *N* in the i^{th} iteration:

- **Node visibility condition**: An ant can see only those nodes from a given node *N* which are in the feasibility set (described in the subsequent part of this section). The condition prevents the ant from covering edges that have been previously covered thus avoiding re-traversal of paths and getting stuck on loops in the graph.
- **Node selection condition**: An ant chooses one of the visible nodes, say *D*, from the feasibility set of *N* as its destination node based on the following ordered preference:

  If *D*'s feasibility set is empty i.e. *D* is a dead end.

  If *D*'s out-degree is maximum.

The out-degree of any node is the adjacency of a node i.e. how many nodes are directly reachable from that node via feasible transitions. An ant first check for the first preference and if it is unsuccessful in doing so, then it looks for a node with higher out degree. In case there are more than one node satisfying the criterion then the choice is arbitrary amongst the colliding nodes. The first preference for a destination node makes the algorithm look for shorter sequences of states in order to minimize state redundancy. The second preference is based on the fact that a path with more number of outgoing edges is expected to have a higher number of uncovered paths. These selection conditions target the specific segments of routes in the graph that have been left uncovered until then with minimum re-traversal of edges.

- **Stopping condition**: An ant K continues to move over the graph until it reaches a node whose feasibility set is empty. This condition forces the ant to stop traversal when non-redundant paths are not available.
- **Root formation condition**: if N has at least one element left in feasibility set when K has travelled from N to some other node, then N is added to the root set R_{i+1}. Thus, the edges uncovered are traversed by some ants from their root itself, ensuring complete coverage of the graph.
- **Coverage condition**: This condition is satisfied when all the edges of the graph are covered.

The algorithm stops looking for new test sequences once this condition is satisfied.

In every iteration i, an ant is initially placed on the first node specified by the root set. It then uses its *node visibility* and *node selection condition* to travel further in the graph until it meets its *stopping condition*. During each edge transition pheromone is updated for that transition by a constant amount $\Delta\tau$ and thus the pheromone value is updated for the edge. Also, the *feasibility set* is updated for the source node because it will not have that edge in its *feasibility set* any longer since it is covered by the ant. This is repeated for every node in the root set R_i in the i^{th} iteration. During the i^{th} iteration, construction of R_{i+1} also takes place based on the *root formation condition*. In the next iteration $(i+1)$, same procedure is followed for the elements in R_{i+1}. The whole process continues until the *coverage condition* is met. The flowchart of the proposed algorithm for an ant K has been given in Fig. 1.

V. Case Study

To bring out the practicality of the algorithm a tool called Optimal Test Sequence generator (OTSG) was developed to implement the proposed algorithm. OTSG takes a CFG as an input and gives some sequences of nodes as output which form a part of the test suite for the SUT. For the practical work used in this paper, OTSG was implemented in C programming language in accordance with the proposed algorithm. We use as case studies the CFGs from the works of Ghiduk [9] and Srivastava *et al*. [15] to demonstrate and compare the performance of our proposed algorithm with their algorithms. Ghiduk [9] has proposed in his paper an algorithm for calculating the test-case sequences and Table I shows the test sequences achieved by that algorithm for the CFG shown in Fig. 2. Table II shows the test-sequences generated using the proposed algorithm. The value of the pheromone update constant $\Delta\tau$ was taken as 1 for our proposed algorithm.

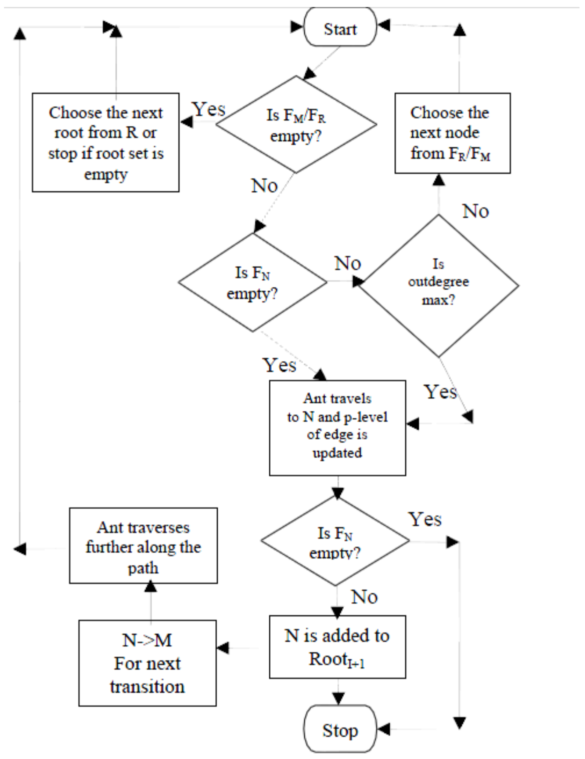
Start
Choose the next root from R or stop if root set is empty
Yes
Is F_M/F_R empty?
Choose the next node from F_R/F_M
No
Is F_N empty?
No
Is outdegree max?
No
Yes
Ant travels to N and p-level of edge is updated
Yes
Ant traverses further along the path
Is F_N empty?
Yes
No
N->M For next transition
N is added to Root_I+1
Stop

Fig. 1. Flowchart of the proposed algorithm for any iteration i.

TABLE I: Test-Sequence Generated Using Ghiduk's [9] Algorithm

Paths	Nodes
Path 1	0-1-2-3-5-6-7-8-10-6-7-9-10-6-11-12
Path 2	0-1-2-4-5-6-7-8-10-6-7-9-10-6-11-12
Path 3	0-1-2-3-5-6-11-12
Path 4	0-1-2-4-5-6-7-9-10-6-11-12

TABLE II: Test-Sequences Using the Proposed Algorithm

Paths	Nodes
Path 1	0-1-2-3-5-6-7-8-10-6-11-12
Path 2	2-4-5
Path 3	7-9-10

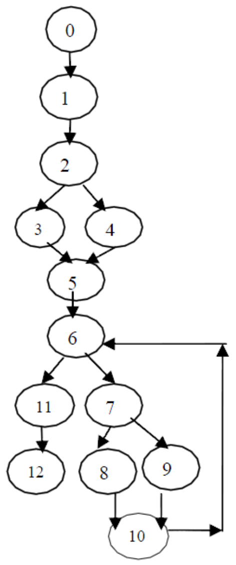

Fig. 2. CFG used for comparison taken from the paper of Ghiduk [9].

As it can be seen, the solution proposed by Ghiduk [9] has a lot of redundancy/repetition of paths. The proposed solution removes the redundant paths, thereby giving a more efficient and optimal solution. Table III lists the test sequences achieved by the algorithm proposed in Srivastava's work [15] for the CFG shown in Fig. 4. Table IV shows the test-sequences generated using the proposed algorithm on the same CFG, from which it can be observed that the redundancy has been reduced. The graph shown in Fig. 3 illustrates the results obtained for the effect of increasing the number of nodes on redundancy in test cases using Ghiduk's [9], Srivatava's [15] and the proposed approach. Thus, for identical conditions, the proposed algorithm gives optimal solution with a zero redundancy. Though with a large number of nodes the behaviour is uncertain, but it can be approximated to giving close to zero-redundancy solution.

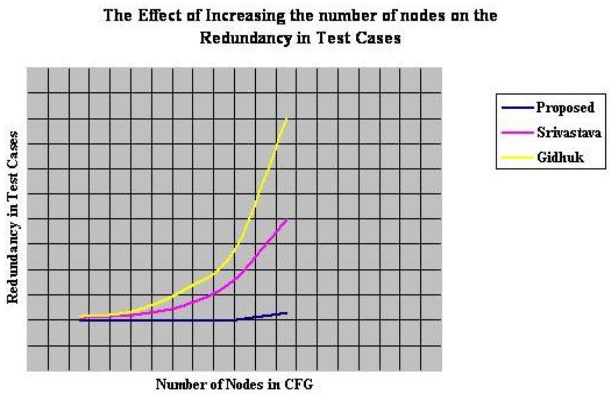

Fig. 3. Graph showing the variation in redundancy in the test paths with no. of nodes in CFG for the approaches used by Gidhuk [9], Srivastava [15] and the one proposed in this paper.

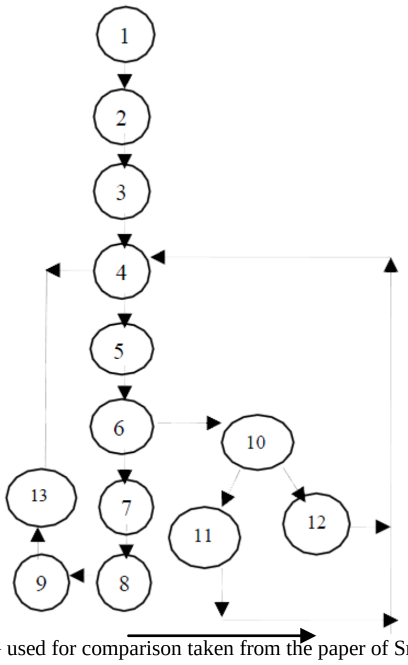

Fig. 4. CFG used for comparison taken from the paper of Srivastava [15].

It is worth mentioning that the algorithm used in this paper not only produces optimal paths but produces them in such a way that those paths are discovered first which end early because of the condition imposed on the ant that it should first look for the end nodes. Secondly, during the journey the ant is made to go through the denser paths with the help of the condition imposed on the ant that it should look for the out-degree of the node and proceed towards the one having more out-degree. These conditions make the ants look for longer paths with uncovered nodes or terminate on shorter paths if it is able to find a node which doesn't have any more uncovered outgoing edges. For example from table 4 it can be seen that path1 has covered almost all the nodes of the CFG of Fig, 4.

TABLE III: TEST-SEQUENCE GENERATED USING SRIVASTAVA'S ALGORITHM [15]

Paths	Nodes
Path 1	0-1-2-3-4-13
Path 2	0-1-2-4-5-6-7-8-9-13
Path 3	0-1-2-3-4-5-6-10-11-4-5-6-7-8-9-13
Path 4	0-1-2-3-4-5-6-10-12-4-5-6-7-8-9-13

TABLE IV: TEST-SEQUENCE USING THE PROPOSED ALGORITHM

Paths	Nodes
Path 1	0-1-2-3-4-5-6-10-12-4-13
Path 2	6-7-8-9-13
Path 3	10-11-4

VI. CONCLUSION AND FUTURE WORK

Optimality is achieved in the generated test suite for an SUT by removing the redundancy in the test–sequences, completely, without losing the coverage and efficiency. The proposed method makes sure that only paths which are not covered in the previous test-sequences are taken into account when generating new test sequences. The algorithm is logical and follows certain rules to achieve maximum coverage. The high running time for large inputs can be said to be one of the drawbacks of the algorithm. As for the future work in this regard, the algorithm could be made to trade off the running time with the coverage of the graph for large inputs. In a limited running time, the priorities of the test sequences generated will be a subject of matter which could be handled by introducing modifications in the algorithm according to the tester's will.

REFERENCES

[1] R. Pressman, *Software Engineering – A Practitioner's Approach*, 5th edition, New York, NY: McGraw Hill, 2001.
[2] I. Sommerville, *Software Engineering*, 8th Edition, Pearson Education, 2009.
[3] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel, "WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation," in *Proc. the 22nd International Conference on Software Engineering*, ACM Press, 2000, pp. 230–239.
[4] W. H. Deason, "Rule-based software test data generation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, issue 1, pp. 108-117, Mar. 1991.
[5] H. Li and C. P. Lam, "Software test data generation using ant colony optimization," in *Proc. International Conference of Computational Intelligence*, Istanbul, Turkey, pp. 1-4, 2004.
[6] M. Dorigo and T. Stutzle, *Ant Colony Optimization*, Phi publishers, 2005.
[7] M. Dorigoa and T. Stutzle, "Ant colony optimization," *The Knowledge Engineering Review*, New York, USA: Cambridge University Press, vol. 20, pp. 92 – 93, 2005.
[8] M. Dorigo, "The ant colony optimization metaheuristic: Algorithms, applications, and advances," *International Series in Operations Research & Management Science*, vol. 57, Springer, New York, pp. 250-285, 2003.
[9] A. S. Ghiduk, "A new software data-flow testing approach via ant colony algorithms," *Universal Journal of Computer Science and Engineering Technology*, pp. 64-72, 2010.
[10] D. Ślęzak, T.-H. Kim, A. Kiumi, T. Jiang, and J. Verner "Advances in software engineering," in *Proc. International Conference on Advanced Software Engineering and Its Applications*," Jeju Island, Korea, December 10-12, pp. 20-35, 2009.

[11] M. Last, A. Kandel, and H. Bunke, *Artificial Intelligence in Software Testing, Quality and Reliability*, World Scientific Publishers, 2004.

[12] K. Doerner and W. J. Gutjahr, "Extracting test sequences from a markov software usage model by ACO," *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 2465-2476, 2003.

[13] R. Lefticaru and F. Ipate, "Automatic State-based test generation Using genetic algorithms," in *Proc. Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, 2007, pp.188-195.

[14] P. R. Srivastava, N. Jose, S. Barade, D. Ghosh, "Test sequence generation from usage models using ant colony optimization," *International Journal of Software Engineering & Applications (IJSEA 2010)*, vol.1, no.2, 2010.

[15] P. R. Srivastava, K. Baby, and G Raghurama, "An approach of optimal path generation using ant colony optimization," in *Proc. TENCON 2009*, IEEE Press, 2009, pp. 1-6.

[16] B. Holldobler and E. O. Wilson, *The Ants*, Berlin: Springer-Verlag, 1990.

[17] A. Roya and L. Shahriar, "The new approach for software testing using a genetic algorithm based on clustering initial test instances," in *Proc. International Conference on Computer and Software Modeling*, IACSIT Press, Singapore, 2011.

[18] W. Zheng, M. R. Lyu, and T. Xie, *Test Selection for Result Inspection via Mining Predicate Rules*, 2008, pp. 1-4.

[19] P. Jalote, *An Integrated Approach to Software Engineering*, Springer, New York, 2005.

[20] S. Kosaraju, "Analysis of structured programs," in *Proc. the 5th Annual ACM Symposium on Theory of Computing*, Austin, TX, USA, 1973, pp. 240–252.

[21] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, issue 4, 1976, pp. 308–320.

[22] M. Paige, "On partitioning program graphs," *IEEE Transactions on Software Engineering*, vol. 3, issue 6, 1977, pp. 386–393.

[23] S. Rapps and E. Weyuker, "Data flow analysis techniques for test data selection," in *Proc. the 6th International Conference on Software Engineering*, Tokyo, Japan, 1982, pp. 272–278.

[24] L. White, "Basic mathematical definitions and results in testing," in B. Chandrasekaran and S. Radicchi eds., *Computer Program Testing*, North-Holland, New York, 1981.

[25] H. Zhu, P. Hall, and J. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

Sumesh Agarwal is a web developer. Currently he is working at Hyderabad. He obtained his masters of science & tech degree in information systems from Birla Institue of Science and Technology, Pilani. His research interests include software development models, data mining, web development, information security and computational applications.

Shubham Gupta is a software developer. Currently he is working at Bangalore. He obtained his masters of science & tech degree in information systems from Birla Institute of Science and Technology, Pilani and is currently working at providing innovative web solutions in IT industry. His research interests include study and analysis of algorithms, web development, mobile development and machine learning.

Nitish Sabharwal is a software developer. Currently he is working in Hyderabad. He obtained his bachelor of engineering degree in computer science (hons.) from Birla Institute of Technology and Science, Pilani. His research interests include artificial intelligence, genetic algorithms, computation and analysis, web and mobile development, machine learning and computational neuropsychology.