Universal Journal of Computer Science and Engineering Technology
1 (1), 64-72, Oct. 2010.

A New Software Data-Flow Testing Approach via Ant Colony Algorithms

Ahmed S. Ghiduk
Department of Computer Science
College of Computers and Information Systems
Taif University, Taif, Saudi Arabia
asaghiduk@tu.edu.sa

***Abstract*—Search-based optimization techniques (e.g., hill climbing, simulated annealing, and genetic algorithms) have been applied to a wide variety of software engineering activities including cost estimation, next release problem, and test generation. Several search based test generation techniques have been developed. These techniques had focused on finding suites of test data to satisfy a number of control-flow or data-flow testing criteria. Genetic algorithms have been the most widely employed search-based optimization technique in software testing issues. Recently, there are many novel search-based optimization techniques have been developed such as Ant Colony Optimization (*ACO*), Particle Swarm Optimization (*PSO*), Artificial Immune System (*AIS*), and Bees Colony Optimization. *ACO* and *AIS* have been employed only in the area of control-flow testing of the programs. This paper aims at employing the *ACO* algorithms in the issue of software data-flow testing. The paper presents an ant colony optimization based approach for generating set of optimal paths to cover all definition-use associations (*du*-pairs) in the program under test. Then, this approach uses the ant colony optimization to generate suite of test-data for satisfying the generated set of paths. In addition, the paper introduces a case study to illustrate our approach.**

Keywords- data-flow testing; path-cover generation, test-data generation; ant colony optimization algorithms

I. INTRODUCTION

There are many critical activities associated with software testing such as 1) finding path-cover to cover a certain testing criterion 2) test-data generation to satisfy the path cover, 3) test execution by using the test data and the software under test and 4) evaluation of test results. A number of test-data generation techniques have been developed.

Random test-data generation techniques select inputs at random until useful inputs are found [1, 2]. This technique may fail to find test data to satisfy the requirements because information about the test requirements is not incorporated into the generation process.

Symbolic test-data generation techniques assign symbolic values to variables to create algebraic expressions for the constraints in the program, and use a constraints solver to find a solution for these expressions that satisfies a test requirement [3, 4]. Symbolic execution cannot determine which symbolic value of the potential values will be used for array as B[c] or pointer. Furthermore, symbolic execution cannot find floating point inputs because the current constraint solvers cannot solve floating point constraints.

Dynamic test-data generation techniques collect information during the execution of the program to determine which test cases come closest to satisfying the requirement. Then, test inputs are incrementally modified until one of them satisfies the requirement [5, 6]. Dynamic techniques can stall when they encounter local minima because they depend on local search techniques such as gradient descent.

Search-based optimization techniques (e.g., hill climbing, simulated annealing, and genetic algorithms) have been applied to a wide variety of software engineering activities including cost estimation, next release problem, and test-data generation [7].

Several search based test-data generation techniques have been developed [8, 9, 10, 11, 12, 13]. Some of these techniques had focused on finding test data to satisfy a wide range of control-flow testing criteria (e.g., [8, 10, 11]) and the other techniques had concentrated on generating test-data for covering a number of data-flow testing criteria [12, 13, 9]. Genetic algorithms have been the most widely employed search-based optimization technique in software testing area [7].

Recently, there are some novel search-based optimizations techniques have been developed such as Ant Colony Optimization (*ACO*) [14, 15], Particle Swarm Optimization (*PSO*) [16], Bees Colony Optimization [17], and Artificial Immune System (*AIS*) [18]. There are few efforts for applying some of these novel search-based optimization techniques in the area of software testing [18, 19, 20, 21, 22, 23, 24, 25, 26].

Ant Colony Optimization (*ACO*) has been applied in the area of software testing in 2003 [19, 20]. Boerner and Gutjahr [19] described an approach involving *ACO* and a Markov Software Usage model for deriving a set of test paths for a software system, and McMinn and Holcombe [20] reported on the application of *ACO* as a supplementary optimization stage for finding sequences of transitional statements in generating test data for evolutionary testing. H. Li and C. P. Lam [21, 22] proposed an Ant Colony Optimization approach to test data generation for the state-based software testing. Bouchachia [18] incorporated immune operators in genetic algorithm to generate software test data for condition coverage. Ayari et al.

Corresponding Author: Ahmed S. Ghiduk, Department of Computer Science, Taif University, Saudi Arabia

[23] proposed an approach based on ant colony to reduce the cost of test data generation in the context of mutation testing. Srivastava and Rai [24] proposed an ant colony optimization based approach to test sequence generation for control-flow based software testing. K. Li et al. [25] presents a model of generating test data based on an improved ant colony optimization and path coverage criteria. P. R. Srivastava et al. [26] presents a simple and novel algorithm with the help of an ant colony optimization for the optimal path identification by using the basic property and behavior of the ants.

However, data-flow testing is important because it augments control-flow testing criteria and concentrates on how a variable is defined and used in the program, which could lead to more efficient and targeted test suites. The results of using ant colony optimization algorithms in software testing which obtained so far are preliminary and none of the reported results directly addresses the problem of test-data generation or path-cover finding for data-flow based software testing.

This paper aims at employing the Ant Colony Optimization algorithms in the issue of software data-flow testing. To our knowledge, this paper is the first work using *ACO* in the issue of data-flow testing. The paper presents an ant colony optimization based technique for generating set of optimal paths to cover all definition-use associations (def-use or *du*-pairs) in the program under test. Then, this technique uses also the ant colony optimization algorithms to generate suite of test-data for satisfying the generated set of paths. In addition, the paper introduces a case study to illustrate our approach.

The rest of the paper is organized as follows. Section 2 gives some basic concepts and definitions. Section 3 introduces two ant colony algorithms for using with data-flow testing. One algorithm generates set of paths for covering all def-use pairs in the software under test (*SUT*) and the other algorithm finds set of test data to satisfy this set of paths. Section 4 presents a technique for implementing the two algorithms in data-flow testing. Section 5 presents a case study to illustrate our approach. Section 6 introduces conclusion and future work.

II. BACKGROUND

This section gives set of basic concepts and definitions which will help in understanding this work.

A. Ant Colony Optimization

Ant Colony Optimization (*ACO*) is a population-based, general search technique for the solution of difficult combinatorial problems, which is inspired by the pheromone trail laying behavior of real ant colonies. The first *ACO* technique is known as Ant System [14] and it was applied to the traveling salesman problem. Since then, many variants of this technique have been produced. Dorigo and Blum in [27] surveyed the theory of ant colony optimization. In *ACO*, a set of software agents called artificial ants search for good solutions to a given optimization problem. To apply *ACO*, the optimization problem is transformed into the problem of finding the best path on a weighted graph. The artificial ants (hereafter ants) incrementally build solutions by moving on the graph. The solution construction process is stochastic and is biased by a pheromone model, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants. Figure 1 shows a generic ant colony algorithm.

Step 1: Initialization
– Initialize the pheromone trail
Step 2: Iteration
– For each Ant Repeat
– Solution construction using the current pheromone trail
– Evaluate the solution constructed
– Update the pheromone trail
– Until stopping criteria

Figure 1. A generic ant colony algorithm

The procedure to solve any optimization problem using *ACO* is:

1) Represent the problem in the form of sets of components and transitions or by means of a weighted graph that is traveled by the ants to build solutions.

2) Appropriately define the meaning of the pheromone trail, i.e., the type of decision they bias. This is a crucial step in the implementation of an *ACO* algorithm. A good definition of the pheromone trails is not a trivial task and it typically requires insight into the problem being solved.

3) Appropriately define the heuristic preference to each decision that an ant has to take while constructing a solution, i.e., define the heuristic information associated to each component or transition. Notice that heuristic information is crucial for good performance if local search algorithms are not available or cannot be applied.

4) If possible, implement an efficient local search algorithm for the problem under consideration, because the results of many *ACO* applications to NP-hard combinatorial optimization problems show that the best performance is achieved when coupling *ACO* with local optimizers.

5) Choose a specific *ACO* algorithm and apply it to the problem being solved, taking the previous aspects into consideration.

6) Tune the parameters of the *ACO* algorithm. A good starting point for parameter tuning is to use parameter settings that were found to be good when applying the *ACO* algorithm to similar problems or to a variety of other problems.

It should be clear that the above steps can only give a very rough guide to the implementation of *ACO* algorithms. In addition, the implementation is often an iterative process, where with some further insight into the problem and the behavior of the algorithm; some initially taken choices need to be revised. Finally, we want to insist on the fact that probably the most important of these steps are the first four, because a poor choice at this stage typically can not be made up with pure parameter fine-tuning.

An *ACO* algorithm iteratively performs a loop containing the following two basic procedures:

1) A procedure for specifying how the ants construct/modify solutions of the problem to be solved;

2) A procedure to update the pheromone trails.

The construction/modification of a solution is performed in a probabilistic way. The probability of adding a new item to

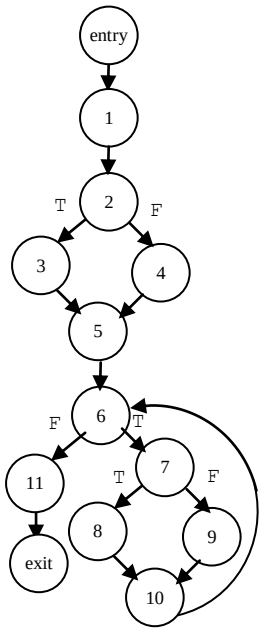
the current partial solution is given by a function that depends on a problem-dependent heuristic and on the amount of pheromone deposited by ants on the trail in the past. The updates in the pheromone trail are implemented as a function that depends on the rate of pheromone evaporation and on the quality of the produced solution.

B. Data-flow analysis and testing

Typically, in structural testing strategies a program's structure is analyzed on the program flow-graph, i.e., an annotated directed graph which represents graphically the information needed to select the test cases.

A control-flow graph (CFG) is a directed graph *G=(V,E),* with two distinguished nodes— a unique entry n_0 and a unique exit n_k. *V* is a set of nodes, where each node represents a statement, and *E* is a set of directed edges, where a directed edge *e = (n,m)* is an ordered pair of adjacent nodes, called tail and head of *e*, respectively. Figure 2(a) gives an example program Program1 and figure 2(b) gives its control-flow graph.

```
   #include <iostream.h>
   void main()
   {
      int a, b, c, n;
1     cin >> a >> b;
2     if(a < 6)
      {
3       c = a;
      }
      else
      {
4       c = b;
      }
5     n = c;
6     while(n < 8)
      {
7       if(b > c)
        {
8         c = 2;
        }
        else
        {
9         n = n + c + 7;
        }
10      n = n + 1;
      }
11    cout << a << b << n;
   }
```



(a) (b)

Figure 2. An example program (a), and its control-flow graph (b)

A path p in a *CFG* is a finite sequence of nodes connected by edges e.g., 1→2→3→5 and 2→4.

The key question addressed in software testing is how to select test cases with the aim of uncovering as many defects as possible.

There are many activities normally associated with software testing such as 1) path-cover finding to cover a certain testing criterion 2) test data generation to satisfy the path cover, 3) test execution involving the use of test data and the software under test (*SUT*) and 4) evaluation of test results.

Coverage criteria require that a set of entities of the program control-flow graph to be covered when the tests are executed. A set of complete paths (path cover) satisfy a criterion if it covers the set of entities associated with that criterion. Depending on the criterion selected, the entities to be covered may be derived from the program control flow or form the program data flow. Frankl and Weyuker in [28, 29] defined a family of popular control flow and data flow test coverage criteria.

Data-flow testing considers the possible interactions between definitions and uses of variables.

The occurrences of a variable in a program can be associated with the following events:

- A statement storing a value in a memory location of a variable creates a definition (*def*) of the variable.
- A statement drawing a value from the memory location of a variable is a use of the currently active definition of the variable. In particular, when the variable appears on the right-hand side of an assignment statement it is called a computational use (*c-use*), when the variable appears in the predicate of the conditional statement it is called a predicate use (*p-use*) [29].
- A statement kills the currently active definition of a variable when its value becomes unbound.

A path is *def-clear path* with respect to a variable if it contains no new definition of that variable.

Data flow analysis determines the *defs* of every variable in the program and the uses that might be affected by these *defs* (i.e. the *du-pairs*). Such data flow relationships can be represented by the following two sets:

- *dcu(i),* the set of all variable *defs* for which there are def-clear paths to their *cuses* at node i; and
- *dpu(i, j),* the set of all variable *defs* for which there are def-clear paths to their *p-uses* at edge (i,j) [30].

Using information concerning the location of variable *defs* and *uses*, together with the 'basic static reach algorithm' [31], the sets *dcu(i)* and *dpu(i, j)* can be determined [30]. Tables 1 and 2 show samples of the *du-pairs* of Program1.

TABLE V. LIST OF DCU-PAIRS FOR PROGRAM1.

dcu	variable	def-node	use-node	killing nodes
1	a	1	3	None
2	c	8	9	3, 4

TABLE VI. LIST OF DPU-PAIRS OF PROGRAM1.

dpu	variable	def-node	use-edge	killing nodes
1	a	1	(2,3)	None
2	n	5	(6,7)	10

III. APPLYING ACO TO DATA-FLOW BASED TESTING

In order to apply *ACO* for generating test data or path cover or any software testing activity, the following number of issues need to be addressed:

1) Problem representation: transformation of the testing problem into a searching model (e.g., control-flow graph);

2) A heuristic measure for measuring the "goodness" of paths through the graph (e.g., how far is it from covering the target);

3) A mechanism for creating possible solutions efficiently and a suitable criterion to stop solution generation;

4) A suitable method for updating the pheromone; and

5) A transition rule for determining the probability of an ant traversing from one node in the graph to the next.

In the following subsections, we introduce two ant colony algorithms for using with data-flow testing. The first algorithm generates set of paths for covering all def-use pairs in the *SUT* and the second algorithm finds set of test data to satisfy this set of paths.

A. Path-Cover generation

The first aim of the paper is driving a path-cover for covering all def-use pairs in the *SUT* using an ant colony optimization algorithm. In this section we will modify and adapt the ant colony optimization algorithm which was suggested by Srivastava et al. in [26] to be correct and appropriate for data-flow testing.

1) Problem Representation

The purpose of the ant colony optimization algorithm is finding for each feasible def-use pair at least one def-clear path in CFG graph of the software under test. Therefore, we will use the control-flow graph as the searching model. In addition, ants will start at the def node and travel to the use node to find the def-clear path from the def node to the use node. Then, the algorithm will randomly select path from the start node to the def node and another path from the use node to the end node to construct a complete path.

For example, the control-flow graph in Figure 2(b) is the searching model for example program in Figure 2(a). In addition for the def-use (c, 8, 9), ants will start their search at node 8 and travel to the destination node 9.

2) Path Selection

Path selection depends upon the probability of this path. The path with high probability has high chances to be selected by the ant. The probability value of path depends upon:

a) Feasibility of path (F_{ij}), which shows that there is direct connection between the nodes and there is no killing nodes on this path;

b) Pheromone trail value (τ_{ij}), which helps other ants to make decision in the future (i.,e, guides the ants to the good path), and

c) Heuristic information (η_{ij}) of the path, which indicates the visibility of a path for an ant at the current node.

In some cases there are more then one feasible path has the asme probability vale then by the following policies the algorithm selects one of these feasible paths.

P.1) An ant will select the next position according to the value of visited status parameter (*Vs*). If current node v_1 is direct connected to the node say v_2 and v_2 not visited yet by the ant and is not killing node, then ant will select v_2 as the next position that means the path ($v_1 \rightarrow v_2$) is traversed.

P.2) If current node v_1 is direct connected to more than one node say v_2 and v_3 and both of them are not visited yet by the ant and are not killing node, then ant will select the nearest one to the use node as the next position that means if v_3 is closer than v_2 from the use node then path ($v_1 \rightarrow v_3$) is traversed.

P.3) If there are many nodes have the same properties then the ant will select any feasible path randomly.

P.4) The algorithm will stop if selection is not possible that means the current def-use pair is infeasible.

P.5) For loop the node will select two times at maximum.

P.6) An ant selects use node as the next node, means ant will select path from current node to use node.

P.7) The algorithm will randomly select path from the start node to the def node and another path from the use node to the end node to construct a complete path.

3) Information Updating

In the proposed algorithm ant has ability to collect the knowledge of all feasible paths from its current position. An approach for feasibility check of the paths from current node is used. This approach is defined in feasibility set of path (F_{ij}). The ant also has four other facts about path:

a) Pheromone level on path (τ_{ij}),

b) Heuristic information for the paths (η_{ij}),

c) Visited nodes with the help of visited status (Vs), and

d) Probability level L.

After selection of a particular path ant will update the pheromone level as well as heuristic value. Pheromone level is increased according to last pheromone level and heuristic information but heuristic information is updated only on the basis of previous heuristic information.

Suppose that an ant *t* at node *'i'* and another node *'j'* which is directly connected to *'i'*, it means there is a path between the nodes *'i'* and *'j'* (i.e., $i \rightarrow j$). In the graph this path associated with five values $F_{ij}(t)$, $\tau_{ij}(t)$, $\eta_{ij}(t)$, $Vs(t)$ and $L_{ij}(t)$ where *t* shows that values associate with ant *t*. The description of these attribute is given below [12]:

1) Feasible path set: $F = \{F_{ij}(t)\}$ represents the direct connection with the current node *'i'* to the neighboring node *'j'*. Direct connection shows that the nodes which are adjacent to the current node *'i'*, i.e. a direct edge exist in between the current node *'i'* and the chosen node *'j'*.

- $F_{ij} = 1$ means that path between the node *'i'* and *'j'* is feasible and node *'j'* is not a killing node.
- $F_{ij} = 0$ means the path between the node *'i'* and node *'j'* is not feasible or node *'j'* is a killing node for the current def-use.

2) Pheromone trace set: $\tau = \tau_{ij}(t)$ represents the pheromone level on the feasible path $(i \rightarrow j)$ from current node *'i'* to next node *'j'*. The pheromone level is updated after the

particular path traversed. This pheromone helps other ants to make decision in future.

3) Heuristic set: $\eta = \eta_{ij}(t)$ indicates the visibility of a path for an ant at current node *'i'* to node *'j'*.

4) Visited status set: Vs shows information about all the nodes which are already traversed by the ant *t*. For any node *'i'*:

- Whereas *Vs (i) =1* indicates that node *'i'* is already visited by the ant *t*.
- *Vs (i) =0* shows that node *'i'* is not visited yet by the ant *t*.

5) Probability set: Selection of path depends upon probabilistic value of path, because it is inspired by the ant behavior. Probability value of the path depends upon the feasibility of path $F_{ij}(t)$, pheromone value $\tau_{ij}(t)$ and heuristic information $\eta_{ij}(t)$ of path for ant *t*. There are two more parameter α and β which used to calculate the probability of a path. These parameters α and β control the desirability versus visibility. α and β are associated with pheromone and heuristic value of the paths respectively.

The proposed ant colony algorithm helps to get not only knowledge of present node but also all feasible paths from current node to next node and historical knowledge of already traversed paths and nodes by the ant.

B. Test-data generation

The second aim of this paper is generating a set of test data to cover all def-use pairs of the *SUT*. In this section we will introduce an adaptation for the ant colony optimization algorithm which was suggested by K. Li et al. in [25] to be suitable to data-flow testing.

1) Problem Representation

The first problem is how to represent the problem in a model which is traveled by the ants to build solutions. The problem can represent in ordered and circular graph [23] or in hierarchical model [25]. In this paper, we augment the hierarchical model with a start node and we use it to represent the problem. The hierarchical model is created by using the input domain of program. Suppose that the input set of program *Prog* is $A=\{x_1, x_2, x_3 \ldots\ldots x_k\}$. Assume that x_i has an input domain D_i, $i \in \{1, 2, 3 \ldots k\}$. Each input domain D_i is divided into sub-domains $D_{i1}, D_{i2} \ldots D_{in}$. Finally, a hierarchical model is built like Figure 3.

The links between layer and layer are complete in this model. By searching the model, we could find the combination between set *n* in layer *i* and set *m* in layer *j*. The data generated from the sets *n* and *m* will have a higher possibility to satisfy the selected path. According to the analysis of these combinations of layers, it is not difficult to obtain the distribution of the data that satisfies the selected path.

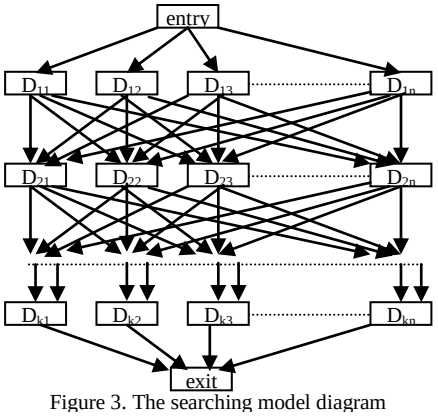


Figure 3. The searching model diagram

2) Path Selection

After constructing the representation graph, this part will introduce the main process for selecting the test data. At first putting a certain number of ants at the start node of the model, then the ant selects a branch to move until getting to the end node. According to the number of each node record in each layer, we can use the data generation functions to get the corresponding data in the corresponding interval. Then we use these data to drive the tested program to run, calculate the executed path and compare it with the def-clear path which will influence the release of pheromone. The pheromone can be updated according to the updating rules.

3) Information Updating

a) The Rules of Pheromone Update

In this approach, modified ant density model is used to update pheromone. The original ant density model is as follows:

$$\Delta\tau_{ij}^{k}(t,t+1) = \begin{cases} Q & \text{if ant passes ij} \\ 0 & \text{otherwise} \end{cases}$$

In the initial density model for any ant *k*, *Q* is a constant, that is, the increment of pheromone is a fixed value. The new *Q* defined in this paper is the number of common nodes between the executed path and the def-clear path of the current def-use pair.

The formula of updating pheromone is:

$$\tau_{ij}(t+n) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}$$

b) The Rules of Next node Selection

Because of the lack of pheromone information in the initial search, ant colony algorithm might easily fall into local optimization. The paper proposes such a strategy, that is, at the early stage of searching, letting the ant choose the path that has the smallest pheromone and ignore the impact of pheromone. In short, we call it the "choose the poorest" strategy. After several iterations, the algorithm abandons this strategy, turning to determine the selection of path which has the most pheromone. The aim of this strategy is to allow ants to explore more paths at the early stage of searching in order to avoid searching partial paths and prevent the algorithm from falling into local optimization. In this way, the new rules for next node selection (i.e., state transition) are:

when m <= tempnum
next_node(i)=min(τ_{ij})
/ next_node(i) returns next node which connects with i*/*
/ min(τ_{ij}) return node j that connects with node i and path ij has the least pheromone */*
m++;
when tempnum < m < maxnum
next_node(i)=max(τ_{ij})
/ max(τ_{ij}) return node j that connects with node i and path ij has the most pheromone */*
m++;
*/*tempnum denotes the iterations times which uses the "choose the poorest" strategy. maxnum denotes the total iterations times of the algorithm. m denotes the loop counter.*/*

In the next section, we present an *ACO* approach using the above information to automatically generate path cover and test data from the control-flow graph for data-flow based software testing.

IV. Our Proposed Approach

In this section we describe our proposed approach for data-flow testing of C++ programs. This approach based upon the ant colony optimization algorithms in section III to solve the problem of deriving a path cover for the def-use associations of the program under test and generating a set of test data that satisfies this path cover. Figure 4 shows the overall diagram of our proposed technique.

Our proposed technique performs the following tasks:

1) Analysis and reformatting of source code.

2) Generating set of program entities to be covered (i.e., all def-use pairs).

3) Generating set of paths to cover the all def-use pairs using ant colony algorithm in section III (A).

4) Generating set of test data using ant colony algorithm in section III (B) to satisfy the set of paths.

The technique performs these tasks in three stages. We give a detailed description of these three stages of the technique in the following subsections.

A. Analysis Module

The analysis and reformatting module has been built to perform the following tasks:

1) Read the program under test, testing criterion and input domains of the variables.

2) Classify program statements and reformats some of them to facilitate the construction of the program control-flow graph.

3) Construct the control-flow graph of the reformatted version of the program.

4) Construct the test data searching model in Figure 3 by using the input domains of the input variables.

5) Produce the set of entities to be covered that satisfies the def-use associations criterion.

6) Instrument the program under test to trace and calculate the executed path.

7) Pass the searching model and the input domains of the variables for the test data generation module.

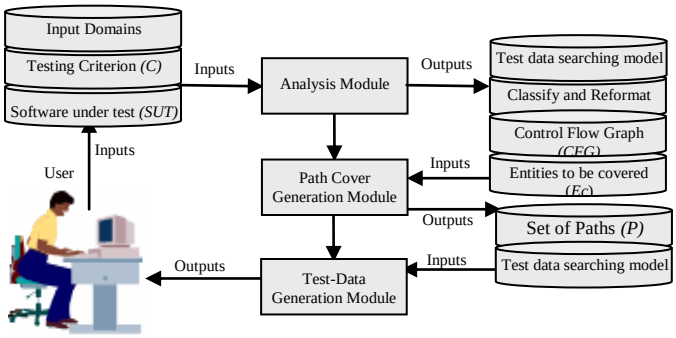


Figure 4: The block diagram of the proposed technique

B. Path-Cover Generation Module

The path-cover generation module uses the following algorithm to generate set of paths to cover all the def-use associations in the software under test. The algorithm easily traverses all the nodes and derives a set of paths which is required for all def-use coverage criterion.

Algorithm for ant t:

Step 0: for each def-use pairs do steps from 1 through 3

0.1 Select DU: select uncovered yet def-use pair to be covered.

0.2 Set start and end node: set the start node to be the def node and the end node to be the use node.

Step 1: Initialize all parameter

1.1 Set heuristic Value (η): for every branch (i.e., branch is a connection between two nodes) in the *CFG* initialize heuristic value $\eta = 2$.

1.2 Set pheromone level (τ): for every branch in the *CFG* initialize pheromone value $\tau = 1$.

1.3 Set visited status (Vs): for every node in the *CFG Vs=0* (initially no node is visited by the ant).

1.4 Set Probability level (L): for each branch in the *CFG* initialize probability *L=0*.

1.5 Set α=1, and β= 1, here α and β are the parameter which controls the desirability versus visibility i.e. desirability means if an ant wants to traverse any particular path on the basis of pheromone value and visibility means the solution which ant has on the basis of prior experience regarding the path. These parameters are associated with pheromone and heuristic values of the paths respectively.

1.6 Set count: count = cc cyclometic complexity describes the different possible paths in *CFG*. The technique automatically calculates the maximum number of possible paths depending upon the value of number of cc value.

1.7. Set key: key = end _node, it is a variable which store the value of end node.

Step 2: Repetition the following steps while count > 0

2. While (count>0)

Evaluation at node *'i'*

2.1. Initialize: *start=i , sum=0, visit=0.*

visit is a variable which used to discard a redundant path and sum used to calculate the value of strength of the path, which later used to prioritize the paths.

2.2. Update the track: Update the visited status for the current node *'i'*
i.e. *if (Vs[i] ==0) then Vs[i] =1 And visit =visit+1 /*increase the value of variable visit*/.*

2.3 Evaluate Feasible Set: Means to determine *F(t)* for the current node *'i'*, this procedure evaluate the entire possible path from the current node *'i'* to the all the neighboring nodes with the help of *CFG* diagram. If there is no feasible path then go to step 3.

2.4 Sense the trace: To sense the trace, evaluate the probability from the current node *'i'* to all non-zero connections in the *F(t)*, as discussed earlier ant's behavior is probabilistic. For every non-zero element belongs to feasible set *F(t)*, we calculate probability with the help of below formula.

$$L_{ij} = \frac{(\tau_{ij})^{\alpha} \times (\eta_{ij})^{-\beta}}{\sum_{1}^{k} ((\tau_{ik})^{\alpha} \times (\eta_{ik})^{-\beta})}$$

For every k belongs to feasible set *F(t)*.

2.5. Move to next node: Using the below rule move to next node

R1: Select paths ($i \rightarrow j$) with maximum probability (L_{ij}).

R2: If two or more paths (e.g., $i \rightarrow j$ and $i \rightarrow k$) have equal probability level like ($L_{ij} = P_{ik}$) then select path according to below rule:

R2.1. Compare each entry in the feasible set with the end_node
If (feasible set entry==end _node) then select end_ node as the next node otherwise follow R2.2.

R2.2. Select that path which have next node not visited yet (i.e., Visited status *Vs =0*). If two or more nodes have same visited status i.e. *Vs[j] =Vs[k]* then follow R2.3.

R2.3. if *Vs[j] =Vs[k]* then select randomly

2.6. Update the parameter:

2.6.1 Update Pheromone: Pheromone is updated for path ($i \rightarrow j$) according to the following rule

$$(\tau_{ij}) = (\tau_{ij})^{\alpha} + (\eta_{ij})^{\beta}$$

2.6.2 Update Heuristic: $\eta_{ij} = 2*(\eta_{ij})$

2.7. Calculate Strength: It shows the values associated with each path
sum = sum + τ_{ij}
strength [count] = sum.
start = next_node.

2.8. if (start! = end_node) then go to step 2.3 else if (visit==0) then discard the path it is the redundant path otherwise add new path.

2.9. Update count: decrement count by one each time.
count =count-1.

Step 3: Complete the generated path

3.1 Randomly select a path from the beginning of the control-flow graph to the def node.

3.2 Randomly select a path from the use node to end node of the control-flow graph.

3.3 Select another uncovered def-use pair and go to step 0.

End //end of algorithm

Variable *count* represents the cyclomatic complexity of a method, as count becomes zero; it shows all the decision nodes traversed. Algorithm will stop automatically in two condition, firstly if there is no feasible def-use pairs and secondly if the all feasible def-use pairs are covered at least once.

C. Test Data Generation Module

The test-data generation module uses the following algorithm to generate set of test data to satisfy the set of paths in the path cover. The algorithm easily traverses all the nodes and derives the required set of data.

Initializing Steps:

1. Build the searching model as in Figure 3.
2. Select one def-clear path from the path cover and mark it.
3. Put ants at start node of the searching model.

Moving Ants Steps:

4. Ant moves and records the number of node.
5. if (ant not get to the end node) goto step 4.
6. Record the path
7. Generate the corresponding data.
8. Execute the program under test using the generated data and record the execution path.
9. Compute the similarity between the execution path and the def-clear path.
10. Update pheromone.
11. if (execution path not cover the def-clear path) goto step 3.
12. record the test data
13. if (there is unmarked def-clear path in the path cover) goto step 2
14. Output the set of test data and the set of covered def-clear paths.
15. End // the algorithm

Algorithm will stop automatically if there are no unmarked def-clear paths in the path cover.

V. CASE STUDY

We have developed a prototype tool called PCTDACO using the proposed algorithms to automatically derive a path cover for all def-use pairs in the program under test and generate a set of test data for this path cover. The proposed prototype is implemented by using C++ based on the above algorithms. This tool is fully automatic because it takes only as inputs the program under test, input domains of the input variables of the program under test. Tool gives output analysis in file format. The tool also produces a file contains the def-use pairs, the path which covers it, and test data which satisfy this path. Tester can see the internal values generated by ant like heuristic, pheromone values, probability calculation and describe selection of best path according to algorithm.

PCTDACO tool automatically calculates the total number of nodes.

For generating the path cover, an ant must start from the def node and it can generate a def-clear path. Def-clear path

depends upon the feasibility of path from the current node to other nodes and accordingly it will take decision for further proceeding and in the end it gives the optimal test path in CFG diagram of software under test. Here optimal means all decision nodes traversed at least once.

Table 3 shows the different def-clear paths which are associated the set of def-use pairs of the example program in Figure 2.

TABLE III. A SET OF DEF-CLEAR PATHS

Def-use pairs	Def-clear path
(a,1,3)	1→2→3
(c,8,9)	8→10→6→7→9
(a,1,[2,3])	1→2→3
(n,5,[6,7])	5→6→7

Table 4 shows the different complete paths which are covered the of def-use pairs *(c,8,9).*

TABLE IV. A SET OF COMPLETE PATH COVER

Def-use pairs	Complete paths
(c,8,9)	entry→1→2→3→5→6→7→ **8→10→6→7→9**→10→6→11→exit
	entry→1→2→4→5→6→7→ **8→10→6→7→9**→10→6→11→exit

Table 5 shows a complete path cover which is covered the set of def-use pairs of the example program in Figure 2.

TABLE V. A COMPLETE PATH COVER

Def-use pairs	Complete paths
(a,1,3)	entry→**1→2→3**→5→6→7→ 8→10→6→7→9→10→6→11→exit
(c,8,9)	entry→1→2→4→5→6→7→ **8→10→6→7→9**→10→6→11→exit
(a,1,[2,3])	entry→**1→2→3**→5→6→11→exit
(n,5,[6,7])	entry→1→2→4→**5→6→7**→9→10→6→11→exit

Our approach arranges the set of complete paths for the same def-use pairs in a priority depending upon the strength of the path (i.e., according to the length of each path) such that the short path has a higher priority than the long one. For example, for the def-use *(a,1,[2,3])* the complete path *entry→1→2→3→5→6→11→exit* has a higher priority than the complete path entry→1→2→3→5→6→7→8→10 →6→7→9→10→6→11→exit.

The brief description about how the def-clear path *8→10→6→7→9* is generated for the def-use pairs (c,8,9) in the *CFG* of the example program Program1 is given in the below.

The tool selects the def-use pairs *(c,8,9)* and initializes all parameter according to step 1, as it is clear from the algorithm of path cover generation. The tool put an *'t'* ant at def node (node 8), for def node tool which generate the feasible set *F(def) = {10}* and ant move to next node 6 as there is no decision node from def node to node 6,ant keep on moving and update all values as per algorithm. At node 6 feasible set i.e. *F[6] ={7,11}* with equal probability and visited status *L(6-7) = L(6-11)* and *V[7] = V[11] = 0*, so according to R3 ant select a node randomly from nodes 7 and 11. Suppose the algorithm selects node 7 as the next node then update parameter along with calculation of Strength.

At node *7* feasible set i.e. *F[7] ={8,9}* with probability level *L(path7-9) > L(path 7-8)* so according to R2.1 ant select node *'9'* as the next node then update parameter along with calculation of Strength. The current ant traveled the path *8→10→6→7→9* and reached the end node which is the use node of the current def-use (node 9). Therefore, the tool will save the current def-use (i.e., *(c,8,9)*) and its def-clear path (i.e., *8→10→6→7→9*). Then, the tool randomly generates any path from the *entry* node of the *CFG* to the def-node (node *8*) and another path from the use node (node *9*) to the exit node of the CFG. The tool can generate the paths *entry→1→2→3→5→6→7* and *10→ 6→11→exit*. Then, the complete path which cover the def-use *(c,8,9)* is *entry→1→2→3→5→6→7→**8→10→6→7→9**→10→6→ 11→exit*.

The tool repeats the above policy with all def-use pairs to complete the path cover.

For generating the test data, an ant must start from the *entry* node of the searching model in Figure 3 and it can generate test datum.

In our case study, we set the range of each input variable of the example program Program 1 (variables *a* and *b*) is 1~100 and divide each range into four smaller ranges: 1~25, 26~50, 51~75, 76~100. We select the path of *entry→1→2→3→5→6→7→**8→10→6→7→9**→10→6→ 11→exit* as the target path. The model we built for this experiment is as follows:

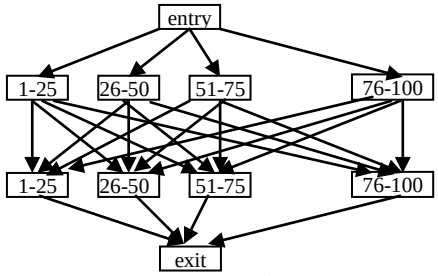


Figure 5. The searching model for example program

The tool starts the by putting *n* ants at the *entry* node of the searching model. Suppose ant *'t'* randomly selects the first node (domain 1 to 25) at the first layer. Then, the ant will select the second node (domain 26-50) in the second layer. Then the ant will get the exit node. Suppose the corresponding data are 6 and 30. Then the tool executes the program under test using the data and record the executed path. The executed path is *entry→1→2→3→5→6→7→ **8→10→6→7**→8→10→6→7→10→6→11→exit*. Then, the tool updates the pheromone and repeats the above strategy until getting the required test data which execute a path covers the selected path. The tool repeats the same strategy with each path in the path cover.

VI. CONCLUSION AND FUTURE WORK

To our knowledge, this paper is the first work using *ACO* in the issue of data-flow testing. This paper aims at employing the Ant Colony Optimization algorithms in the issue of software data-flow testing. The paper presented an ant colony optimization based approach for generating set of optimal paths to cover all definition-use associations (*du*-pairs) in the program under test. This approach uses also the ant colony

optimization algorithms to generate suite of test-data for satisfying the generated set of paths. The ant colony algorithms are adopted to search the *CFG* and a model built on the program input domain in order to get the path cover and the test data that satisfies the selected path.

Our future work will focus on estimates the efficiency of ant colony optimization algorithms against genetic algorithms in this area. In addition, we will concentrate on solving the problem of constructing the searching model for the program with input variable of boolean and character type. In addition, how to revise the model to be applied to object-oriented programs?

REFERENCES

[1] H. D. Mills, M. D. Dyer, and R. C. Linger, "Cleanroom software engineering," IEEE Software, vol. 4, pp. 19-25, 1987.

[2] J. M. Voas, L. Morell, and K. W. Miller, "Predicting where faults can hide from testing," IEEE, vol. 8, pp. 41-48, 1991.

[3] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Transactions on Software Engineering, vol. 3, no. 4, 266-278, 1977.

[4] T. E. Lindquist, and J. R. Jenkins, "Test-case generation with IOGen, IEEE Software," vol. 5, no. 1, pp. 72-79, 1988.

[5] R. Ferguson and B. Korel, "The chaining approach for software test data generation," ACM TOSEM, vol. 5, pp. 63-86, 1996.

[6] B. Korel, "Automated software test data generation," IEEE Trans. on Software Engineering, vol. 16, pp. 870-879, 1990.

[7] M. Harman, "The current state and future of search based software engineering," Proc. of the International Conference on Future of Software Engineering (FOSE'07), May 2007, pp. 342-357. IEEE Press.

[8] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test data generation using genetic algorithms, Journal of Software Testing," Verifications, and Reliability, vol. 9, pp. 263-282, 1999.

[9] A. S. Ghiduk, M. J. Harrold, M. R. Girgis, "Using genetic algorithms to aid test-data generation for data flow coverage," Proc. of 14th Asia-Pacific Software Engineering Conference (APSEC 07), Dec. 2007, pp. 41-48. IEEE Press.

[10] C. C. Michael, G. E. McGraw, M. A. Schatz, "Generating software test data by evolution," IEEE Transactions on Software Engineering, vol.27, no.12, pp. 1085-1110, 2001.

[11] J. Wegener, A. Baresel, H. Sthamer, "Evolutionary test environment for automatic structural testing," Journal of Information and Software Technology, vol. 43, pp. 841-854, 2001.

[12] L. Bottaci, "A genetic algorithm fitness function for mutation testing," Seminal: Software Engineering Using Metaheuristic Innovative Algorithms, 2001.

[13] M. R. Girgis, "Automatic test data generation for data flow testing using a genetic algorithm," Journal of Universal computer Science, vol. 11, no. 5, pp. 898-915, 2005.

[14] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant System: Optimization by a Colony of Cooperating Agents," IEEE Transactions on Systems, Man, and Cybernetics-Part B Cybernetics, vol. 26, no. 1, pp. 29-41, 1996.

[15] C. Blum, "Ant colony optimization: introduction and hybridizations" Proc. of 7th International Conference on Hybrid Intelligent Systems (HIS'07), Sept. 2007, pp. 24-29. IEEE Press.

[16] X. Zhang, H. Meng, and L. Jiao, "Intelligent particle swarm optimization in multiobjective optimization," Proc. of the 2005 IEEE Congress on Evolutionary Computation, Vo. 1, pp. 714-719. IEEE Press.

[17] D.T. Pham, A. Ghanbarzadeh, E. Koç, S. Otri, S. Rahim, and M. Zaidi "The bees algorithm – A novel tool for complex optimisation problems" Proc. of Innovative Production Machines and Systems Conference (IPROMS'06), 2006, pp.454-461.

[18] A. Bouchachia, "An immune genetic algorithm for software test data generation" Proc. of 7th International Conference on Hybrid Intelligent Systems (HIS'07), Sept. 2007, pp. 84-89. IEEE Press.

[19] Doerner, K., Gutjahr, W. J., "Extracting Test Sequences from a Markov Software Usage Model by ACO", LNCS, Vol. 2724, pp. 2465-2476, Springer Verlag, 2003.

[20] McMinn, P., Holcombe, M., "The State Problem for Evolutionary Testing", Proc. GECCO 2003, LNCS Vol. 2724, pp. 2488-2500, Springer Verlag, 2003.

[21] H. Li and C. P. Lam, "Software test data generation using ant colony optimization" World Academy of Science, Engineering and Technology vol.1, 2005, pp.1-4.

[22] H. Li and C. Peng LAM , "An Ant Colony Optimization Approach to Test Sequence Generation for State based Software Testing", Proceedings of the Fifth International Conference on Quality Software (QSIC'05), pp 255 – 264,2005.

[23] K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," Proc. of International Conference on Genetic and Evolutionary Computation Conference (GECCO'07), July 2007, pp 1074-1081. ACM Press.

[24] P. R. Srivastava, and V. K. Rai "An ant colony optimization approach to test sequence generation for control flow based software testing" Proc. of 3rd International Conference on Information Systems, Technology and Management (ICISTM'09), March 2009, pp. 345-346. Springer Berlin Heidelberg

[25] K. Li, Z. Zhang, and W. Liu, "Automatic Test Data Generation Based On Ant Colony Optimization," Proc. of Fifth International Conference on Natural Computationn 2009, pp. 216-219. IEEE Press.

[26] P. R. Srivastava, K. Baby, and G Raghurama, "An Approach of Optimal Path Generation using Ant Colony Optimization," Proc. of TENCON 2009, pp.1-6. IEEE Press.

[27] M. Dorigo and C. Blum "Ant colony optimization theory: A survey", Theoretical Computer Science, 344(2-3), pp. 243-278, 2005.

[28] P. G. Frankl, and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," IEEE Transactions on Software Engineering, vol. 14, 1988, no. 10, pp. 1483-1498.

[29] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information," IEEE Transactions on Software Engineering, vol.11, no. 4, pp. 367-375, 1985.

[30] M.R. Girgis and M.R. Woodward, "An integrated system for program testing using weak mutation and data flow analysis," Proceedings of Eighth International Conference on Software Engineering, IEEE Computer Society, pp. 313-319, 1985.

[31] F. E. Allen and J. Cocke, "A program data flow analysis procedure," Communication of the ACM, 19 (3), 137-147, 1976.

Ahmed S. Ghiduk is an assistant professor at Beni-Suef University, Egypt. He received the BSc degree from Cairo University, Egypt, in 1994, the MSc degree from Minia University, Egypt, in 2001, and a Ph.D. from Beni-Suef University, Egypt in joint with College of Computing, Georgia Institute of Technology, USA, in 2007. His research interests include software engineering especially search-based software testing, genetic algorithms, and ant colony. Currently, Ahmed S. Ghiduk is an assistant professor at College of Computers and Information Systems, Taif University, Saudi Arabia. One can connect Ahmed S. Ghiduk on asaghiduk@yahoo.com or gamil.com.