Generation of Optimal Testing Paths using Anti –Ant Colony Algorithm

Abdullah H. Ahmed
Software Engineering Department
College of Computer Science and Mathematics
University of Mosul, Iraq

Dujan B. Taha
Software Engineering Department
College of Computer Science and Mathematics
University of Mosul, Iraq

ABSTRACT
The most important issue for software industry is to ensure software quality. The software that delivered to the end user must have high quality that meets user requirements. Software must be tested to ensure its quality. In the context of the path-based testing it is important to find the optimal paths from all possible code execution paths to reduce testing process cost and time. The optimal paths are the less number of the paths that cover all source code statements.

The proposed algorithm accepts java source code as an input and generates the control flow graph(CFG) corresponding to that code. Using anti-ant colony optimization, the optimal paths corresponding to the source code will be generated. The proposed algorithm determines the minimum set of paths that cover all source code statements efficiently and accurately.

General Terms
Software testing.

Keywords
Software Testing; Path Testing; Anti-Ant Colony Optimization; Cyclomatic Complexity; Control Flow Graph(CFG)

1. INTRODUCTION
Software testing is an important step in software development process. The primary goal of software testing is to ensure that the developed software have high quality that meets end user requirements.

Software testing is the most costly step of software development life cycle that costs about 40-60% of the effort, time and cost of all software development process [1][2]. Software testing strategies can be divided into black-box (functional) testing and white-box (structural) testing.

White box testing (glass box testing), is a method for test case design that depends on the control structure of the procedural design to derive test cases [3]. Black box testing (functional testing) is a testing type in which a software tested without having the knowledge of the internal structure of the code [4]. Functional testing is based on functional requirements whereas structural testing is done on code itself. Gray box testing is another testing type that merge the white box testing and black box testing. The software testing process must cover high percentage of the software source code.

Basis path testing is one of the most important white-box testing techniques. The most important aspect in white box testing is selecting the test paths that can increase the probability of finding defects in the software [5]. Software testing can be done either manually or automatically using testing tools. The automatic software testing is better than manual testing because the former reduces the cost and time of testing. However, there is still lack of automatic and highly efficient tool for generating basic paths in white-box testing [1].

In this work we generated the CFG for java code by parsing and analyzing the source code. The CFG was represented as a graph matrix that represents the control flow states connections. Then the optimal bases paths from the graph matrix have been generated using anti-ant colony algorithm. The optimal bases paths generated by the proposed algorithm represent the less number of paths that cover all source code statements.

The rest of the paper is organized as follows. Section 2 reviews background and some terminologies that will be used in this paper. Section 3 presents ant colony optimization. Section 4 discusses the related works. Section 5 demonstrates the proposed algorithm. Section 6 presents a case study. Finally, Section 7 concludes the paper.

2. BACKGROUND
One of the most important issue of software testing is that the testing process must cover the code under testing as much as possible. In software testing it is very important to use techniques and methods that find maximum faults in minimum time [6]. Modern software became complex and contain large amount of source code so it is impossible to test all code statements manually, therefore there are many works and tools now for automatic software testing to reduce testing time and cost.

White box testing focus on procedural details of the code. The source code logical paths are tested by providing test cases that exercise specific paths to cover specific conditions and/or loops that contained in the code [4].

There are many types of testing coverage metrics like statements coverage, conditional coverage, branch coverage, decision coverage, and function coverage [6].

Path testing or basis path testing is one of the most important techniques in white box testing [5]. It is infeasible to test all the paths in the CFG because it is a time and cost consuming process. So, it is important to find a set of optimal paths that cover all code statements.

CFG that represents the control flow of code are widely used in software analysis and testing. CFG depicts the logical structure of the source code under test [5][7]. The CFG contains a set of nodes and edges. Every node in the CFG represents one or more code statements while each edge represents the relationship between nodes or flow of control between nodes [1][4]. In the CFG, the area that curved by nodes and edges is called region of CFG [6].

Cyclomatic complexity is one of the most used metrics that provides a measure of the logical complexity of the program

[1]. It is always used to find a number of linearly independent paths in the CFG. The independent path is the path that has at least one new node that not contained in previous paths [5].

The value of cyclomatic complexity V(G) can be computed in the following ways [1]:

1. $V(G) = e - n + 2$ ……. (1)

Where e represents the edges number in the CFG and n is the nodes number.

2. $V(G) = P + 1$ ……. (2)

Where P is the number of predicate nodes in the CFG.

3. $V(G) =$ number of CFG regions ……. (3)

The outside of the CFG is also represents a region.

The computed value of the cyclomatic complexity represents the upper bound of linearly independent paths in the CFG [3]. Thus in some times the number of independent paths may be less than the cyclomatic complexity value.

3. ANT COLONY OPTIMIZATION

Ant colony optimization is one of the artificial intelligence metaheuristic techniques that inspired by real ant colonies [6][8]. The ants that seek for food coordinate with each other by dropping and sensing the paths pheromone level. The ants depend on stochastic or probability theory to select their path [2][9][10]. The ACO was originally applied to solve the classical travelling salesman problem and it finds a good solutions[11]. During last few years Ant Colony Optimization approach has been used to solve the complex computational problems and software testing is one of these problems [6].

In our work we use anti-ant Colony Optimization to find the optimal testing paths. The ant moves from start node to other nodes until it reaches the exit node. During its movement from node to another it updates the pheromone value between the two nodes so that the other ants use the updated value to determine their paths. The difference between ant colony algorithm and anti-ant colony algorithm is that in ant colony algorithm the ant selects the path that has maximum value of pheromone while in anti-ant colony algorithm the ant selects the path that has the minimum pheromone value.

4. RELATED WORKS

There are many works on different types of software testing. Some of them focus on path-based test. Most of path-based test works use the artificial intelligence and optimization algorithms.

Lin et al. [12] use genetic algorithms for test case generation. Chen et al. [13] suggest an algorithm for the whole control flow paths of the source code with the help of the LCC compiler in software coverage testing. Sharma et al. [2] propose an approach that tries to find out the test sequence and effective paths by applying ant colony optimization principle and some set of rules and try to reach maximum software coverage with minimal redundancy. Saurabh et al. [5] define a method to optimize time and complexity of software testing by using ant colony optimization algorithm to prioritize the feasible paths. Papadakis et al. [14] define an approach that convert the CFG to an enhanced one in order to use it for paths selection for mutation testing. Bhuvnesh [5] define a method to optimize time and complexity of software testing using ant colony optimization algorithm to prioritize the feasible paths. Mukesh Mann et al. [2] suggest a method for generating and prioritizing optimal paths using ant colony optimization. Xinyang Wang et al. [1] introduce a method for transforming the source code to corresponding CFG and suggest an algorithm to find out all basic paths automatically.

5. THE PROPOSED METHOD

The proposed method accepts the source code written in java as an input and then processes this code through multiple steps to generate the CFG corresponding the source code. The CFG was represented as a state transition matrix, and the anti-ant colony algorithm was applied on this matrix to generate the optimal paths for the source code. Figure 1 shows the architecture of the proposed method.

First, the algorithm reads the java source code as a text file. Code pre-preprocessing includes removing the comment sentences from the source code and make every line contains only one code statement.

In the second step, the algorithm parses the source code to identify the type of each code statements by tokenization.

Generally, there are three types of code statements, they are [4]:

1. simple statements: this type of statements contains initialization, variable declaration, assignment, input and output statements.

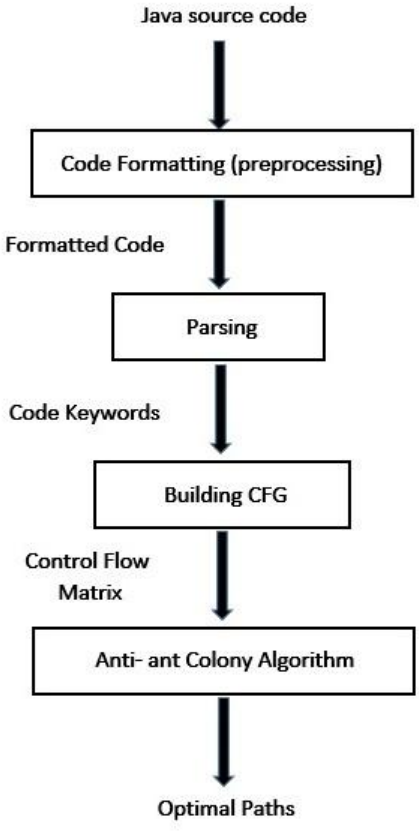

Figure 1. The proposed method architecture

N=0;
While (N < Cyclomatic Complexity)
1. Add first node to the current path
While (current node! = end node)
2. Select the next node according to the following rules:
a. If current node leads to only one node, then select this node as next node
b. If current node leads to multiple nodes, then select the node that has less connection value as a next node, if there are multiple node that has the same minimum connection value, then select the first node (minimum number node) as next node.
4. increase the value of edge connection between current node and next node by one.
5. Set next node as current node.
6. Add current node to the current path.
End while
7. Add current path to the paths list
8. Reset the current path to be empty
9. N=N+1
End while
10. If there is a repeated path in the paths list then remove it.

Figure 2. The proposed anti-ant colony algorithm

2. Conditional Statements: contains if, if – else and switch statements.

3. Loop Statements: this type contains for, while, do – while statements.

Conditional and loop statements affect the flow of program execution, so those statements are searched. If the code statement doesn't contain a loop or conditional keywords, then the statement is a simple statement. The output of this step is a list of keywords that represent the structure of each code statement. The generated keywords list is used to generate the CFG corresponding to the source code. The state number for every keyword and the states connections with each other according to the flow of code execution was produced. The CFG was represented by a graph matrix of size (n x n) where n represents the number of nodes in the CFG. If there is a connection between (n node) and (m node) then the value of graph matrix at index (n, m) is set to one, otherwise the value is set to 0. After the generation of the graph matrix, the cyclomatic complexity is calculated using one of the three methods mentioned in section 2.

The last step of the algorithm finds the optimal paths for the source code by applying the anti-ant colony optimization algorithm on the values of the CFG matrix. Figure 2 describes the proposed anti-ant colony optimization algorithm for finding the optimal paths set.

The ant starts from start node and it moves node by node until it reaches the exit node. In every node the ant decides the next step movement according to the connection value in the graph matrix. If the ant finds that the current node has only one edge, it moves directly to the next node and the connection value in matrix is increased by 1. If there are multiple edges that connect the current node with the next nodes, then the ant must decide to which node it moves. In this case the ant decision depends on the connection value in the graph matrix. The ant chooses the edge that has the least connection value. If the edge connections have the same value, then the ant moves to the first (least number) node. In every step the node is added to the current path and the connection value between current node and next node is increased by one. When the ant reaches the exit node, the current path is added to the paths list. In every step new ant starts its search and finds a new independent path. The number of ants we needed in the proposed method are equal to the cyclomatic complexity value. Sometimes paths may be repeated. So, in the final step all repeated paths are removed from the independent paths list.

6. CASE STUDY

In the case study we introduce an example for a triangle classification. The same example was used by the work of Mukesh Mann et al [2]. The difference is that our code was written in java while their code was written in C. Figure 3 represents the case study source code.

The CFG produced by the algorithm corresponding to the source code is as shown in figure 4. The CFG contain 18 nodes, the number 1 node represents the start node while the node number 18 represents exit (end) node. The CFG represented as a graph matrix, the matrix values represent the connections between CFG nodes. For example, in our case study the node number 2 has a connection to node 3 and node 7 so in the graph matrix the connection is represented by the value 1 in the (2,3) and (2,7) positions of the matrix. Figure 5 represents the graph matrix for the CFG.

After the generation of CFG we calculate the value of cyclomatic complexity for the source code by using one of the three ways mentioned previously in section 2.

Since the number of CFG edge = 23 and the number of CFG nodes= 18, then, using the first formula for computing cyclomatic complexity, the cyclomatic complexity =7

Since there are 6 predicate nodes in the CFG, the second formula produces the value 7 for the cyclomatic complexity.

using the third formula, the cyclomatic complexity is computed by calculating the number of CFG regions. Since the CFG has 7 regions, then the cyclomatic complexity is 7.

```
{    double a1, a2, a3; int valid = 0;
   Scanner in = new Scanner(System.in);
   if (a > 0 && a < 100 && b > 0 && b <= 100 && c > 0 && c < 100) {
      if ((a + b) > c && (b + c) > a && (c + a) > b) {
         valid = 1;
      } else {
         valid = -1;
      } }
   if (valid == 1) {
      a1 = (a * a + b * b) / (c * c);   a2 = (b * b + c * c) / (a * a);  a3 = (c * c + a * a) / (b * b);
      if (a1 <= 1 || a2 <= 1 || a3 <= 1) {
         System.out.println("obtuse angled triangle");
      } else if (a1 == 1 || a2 == 1 || a3 == 1) {
         System.out.println("right angled triangle");
      } else {
         System.out.println("acute angled triangle");
      }
   } else if (valid == -1) {
      System.out.println("invalid triangle");
   } else {
      System.out.println("input values are out of range");
   }  }
```

Figure 3. The case study source code

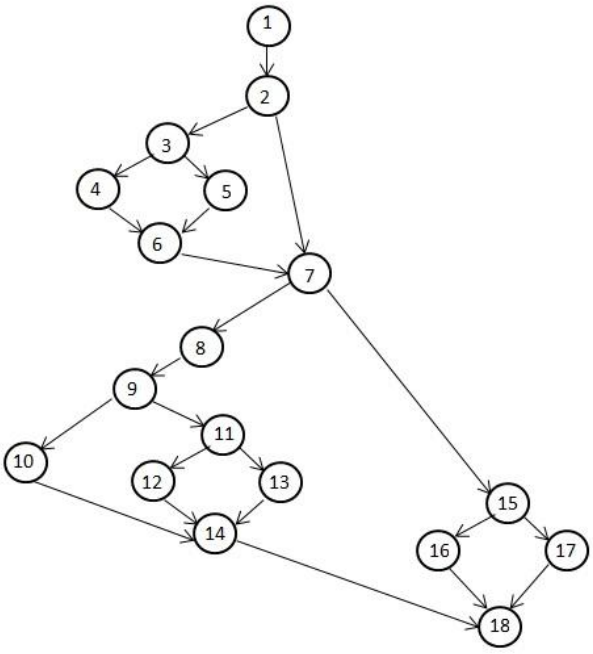

Figure 4. CFG for the case study source code

Figure 5. Graph matrix for the CFG

Figure 6. The updated graph matrix after finding the first path

After computing the cyclomatic complexity value, the anti-ant colony algorithm was used to find the optimal set of independent paths that cover all the CFG nodes.

The first ant finds the first path which is: 1, 2, 3, 4, 6, 7, 8, 9, 10, 14, 18. Figure 6 shows the updated graph matrix after the first ant finds the first path and updates the connection values. The updated matrix shows that the connection values that the ant pass from them were updated by increasing the value by one. For example, the ant moved from node number 2 to node number 3, so the value of the matrix at the position (2,3) was increased by one and the connection value became 2. The anti-ant colony algorithm continues to find the optimal set of independent paths. In our example, at the end, the algorithm finds five independent paths that cover all the nodes of the CFG. These paths are:

Path 1: 1, 2, 3, 4, 6, 7, 8, 9, 10, 14, 18

Path 2: 1, 2, 7, 15, 16, 18

Path 3: 1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 14, 18

Path 4: 1, 2, 7, 15, 17, 18

Path 5: 1, 2, 3, 5, 6, 7, 8, 9, 11, 13, 14, 18

As result shows, the set of optimal basis paths contains only five independent paths from fifteen available paths, while the cyclomatic complexity value is seven. In this example we found the least number of paths that covers all CFG nodes and that means that these five paths cover all source code statements. Table 1 shows the comparison between Mukesh Mann et al. [2] algorithm results and the proposed algorithm results.

7. CONCLUTION

In the proposed algorithm, we use the anti-ant colony optimization to find the set of optimal paths for software testing. The results show that the proposed algorithm is very appropriate to find the less number of independent paths from CFG.

The optimal paths are the least number of source code paths that cover all the source code statements. In some times the number of optimal paths is equal to the cyclomatic complexity value, but in other times the number of optimal paths is less than the cyclomatic complexity value.

In comparison with the result of the work presented by Mukesh Mann et al [2]. They found seven independent paths, whereas we found 5 paths in our proposed algorithm. So, we found the optimal number of paths. That is the paths that cover all source code statements.

Table 1. the comparison between Mukesh Mann et al. [2] algorithm and the proposed algorithm

	Mukesh Mann et al.[2] algorithm	**Proposed algorithm**
No. of paths	7	5

paths	Path 1: 1, 2, 7, 15, 17, 18 Path2: 1, 2, 7, 15, 16, 18 Path3: 1, 2, 7, 8, 9, 11, 13, 14, 18 Path4: 1, 2, 7, 8, 9, 11, 12, 14, 18 Path5: 1, 2, 7, 8, 9, 10, 14, 18 Path6: 1, 2, 3, 5, 6, 7, 8, 9, 10, 14, 18 Path7: 1, 2, 3, 4, 6, 7, 8, 9, 10, 14, 18	Path 1: 1, 2, 3, 4, 6, 7, 8, 9, 10, 14, 18 Path 2: 1, 2, 7, 15, 16, 18 Path 3: 1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 14, 18 Path 4: 1, 2, 7, 15, 17, 18 Path 5: 1, 2, 3, 5, 6, 7, 8, 9, 11, 13, 14, 18
Algorithm complexity	High	Low

8. REFERENCES

[1] Xinyang Wang , Yaqiu Jiang, and Wenhong Tian, "An Efficient Method for Automatic Generation of Linearly Independent Paths in White-box Testing," International Journal of Engineering and Technology Innovation, vol. 5, no. 2, pp. 108-120, 2015.

[2] Mukesh Mann and Om Prakash Sangwan , "Generating and prioritizing optimal paths using ant colony optimization, " Computational Ecology and Software, 5(1): 1-15, 2015.

[3] Roger S. Pressman, Software Engineering A practitioner's approach 7th Edition. 2010.

[4] Hina Sattar , Imran Sarwar Bajwa , and Umar Farooq Shafi , "Automated DD-path Testing and its Significance in SDLC Phases," Journal of Digital Information Management, Volume 13, Number 5, 2015.

[5] Saurabh Srivastava, Sarvesh Kumar, and Ajeet Kumar Verma, "Optimal Path Sequencing in Basis Path Testing," International Journal of Advanced Computational Engineering and Networking, ISSN (p): 2320-2106, Volume – 1, Issue – 1, 2013.

[6] Robert Gold, "Control Flow Graphs And Code Coverage," Int. J. Appl. Math. Comput. Sci., Vol. 20, No.4, 739–749, 2010.

[7] Bhuvnesh Sharma, Isha Girdhar, Monika Taneja, Pooja Basia, Sangeetha Vadla, and Praveen Ranjan Srivastava, "Software Coverage : A Testing Approach through Ant Colony Optimization," B.K. Panigrahi et al. (Eds.): SEMCCO 2011, Part I, LNCS 7076, pp. 618–625, 2011. © Springer-Verlag Berlin Heidelberg 2011.

[8] Wang, Y.; Xie, J. Y. Ant colony optimization for multicast routing in Circuits and Systems. IEEE: The 2000 IEEE Asia-Pacific Conference; 2000; pp 54-57.

[9] M. Dorigo, C. Blum, Ant colony optimization theory: a survey, Theor. Comput. Sci. 344 (2005) 243–278.

[10] Silva, A. R. M.; Ramalho, G. L. Ant system for the set covering problem Systems.2001 IEEE International Conference on Man and Cybernetics;2001; Vol. 5, pp 3129-3133.

[11] M. Dorigo, V. Maniezzo, and A. Colorni. Ant Systems: Optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26: 1996, 29-41

[12] J. Lin, P. Yeh., "Using Genetic Algorithms for Test Case Generation in Path Testing," Proceedings of the Ninth Asian on Test Symposium, Asian, pp. 241-246, 2000.

[13] Y. Chen, Z. Li, H. Jin, J. He, "Control Flow Paths Subset of Tested Program Generation Algorithm Based on LCC," Computer Engineering, vol. 35, no. 7, pp. 39-41, 2009.

[14] M. Papadakis, N. Malevris, "Mutation based test case generation via a path selection strategy," Information & Software Technology, vol. 54, no. 9, pp. 915-932, 2012.

AUTHORS

Dr. Dujan B. Taha (Assistant Prof.) is currently a lecturer at Mosul University, College of Computer Science and Mathematics / Software Engineering Department. She received B.Sc. degree in Computer Science / University of Mosul in 1991, M.Sc. degree / University of Mosul in 1996 and Ph.D. degree / University of Mosul in 2005. Her research interests are in information and network security, software engineering, image processing and pattern recognition.

Abdullah H. Ahmed is currently an M.Sc. student in Software Engineering Department / Collage of Computer Science and Mathematics / University of Mosul.