International Journal of Engineering and Technology Innovation, vol. 5, no. 2, 2015, pp. 108-120

An Efficient Method for Automatic Generation of Linearly Independent Paths in White-box Testing

Xinyang Wang[1], Yaqiu Jiang[2], Wenhong Tian[3*]

School of Software and Engineering, University of Electronic Science and Technology of China, Chengdu, China.

Received 08 April 2014; received in revised form on 07 April 2015; accepted on 08 April 2015

Abstract

Testing is the one of most significant quality assurance measures for software. It has been shown that the software testing is one of the most critical and important phases in the life cycle of software engineering. In general, software testing takes around 40-60% of the effort, time and cost. Structure-oriented test methods define test cases on the basis of the internal program structures and are widely used. Path-based test is one of the important Structure-oriented test methods during software development. However, there is still lack of automatic and highly efficient tool for generating basic paths in white-box testing. In view of this, an automatic and efficient method for generating basic paths is proposed in this paper. This method firstly transforms the source-code program into corresponding control flow graph (CFG). By modifying the original CFG to a strongly connected graph, a new algorithm (ABPC) is designed to automatically construct all basic paths. The ABPC algorithm has computational complexity linear to the number of total edges and nodes in the CFG. Through performance evaluation of many examples, it is shown that the proposed method is correct and scalable to very large test cases. The proposed method can be applied to basis path testing easily.

Keywords: White box test; Path-Based Test; Control flow graph; Complexity Measure; Elementary circuit.

1. Introduction

It has been shown that the software testing is one of the most critical and important phases in the life-cycle of software engineering. In general, software testing takes around 40-60% of the effort, time and cost [16]. Structure-oriented test methods define test cases on the basis of the internal program structures and are widely used. Path-based test is one of the important Structure-oriented test methods during software development. One critical question facing path testing in software engineering is: how to generate linear independent paths automatically and efficiently [18].

1.1 Related Work

There are already many research on path-based test. Chen et al. [4] proposes an algorithm for the whole control flow paths of the program with the help of the LCC compiler in software coverage testing.

Lin et al. [12] discusses one generation method for test case based on genetic algorithms. Wegener et al. [23] introduces an evolutionary test method for automatic structural testing. Yan et al. [26] discusses one method to generate feasible paths for basis path testing, especially to identify possible infeasible paths in basis paths. Du et al. [5] proposes a method to simplify control flow graph to avoid infeasible path. McCabe introduces details of structured testing by using cyclomatic complexity measure, Prasanan et al [18] provides a survey on automatic test case generation. White et al. [22] designs a tool for simple loop patterns in path testing with loops, In [12], [11], [27], and [29], algorithms related to Genetic Algorithm and Particle Swarm Optimization (PSO) are proposed for test case generation. Papadakis et al. [30] [31] transform the control flow graph to an enhanced one in order to select paths for mutation testing. Offutt et al. [32] discuss automatic method to detect equivalent mutants and infeasible paths. Papadakis et al. [33] propose an automated symbolic execution tool for elimination of infeasible paths. Ngo et al. [34] introduce a heuristics-based approach to infeasible path detection for dynamic test data generation.

However, automatic and highly efficient generation of linearly independent paths from source codes in basis path testing is still lack. To solve the problem, this paper proposes a new and efficient method of path-based tests after analyzing open literature in this area. Through adding an edge to the program Control Flow Graph (CFG), basic path testing problem is transformed to find all strongly connected components and circuits in graph theory. We propose Automatic Basis Path Construction (ABPC) algorithm by reducing fruitless searching to improve time complexity and efficiency of basic path generating. Also scalability is showed in proposed method.

1.2 Preliminaries

1.2.1 Control Flow Graph

A Control Flow Graph (CFG) is a visual representation of the various paths that the source code of a computer program can take, and at the same time it is also a glimpse of the original structure of the source code.

A CFG is composed of a series of symbols, such as nodes and edges. Each node represents a specific line or lines of original programming code.

1.2.2 Cyclomatic complexity

Cyclomatic Complexity is a software metric that provides a quantitative measure of the logical complexity of a program. It was developed by McCabe in 1976 [13] and is used to show the complexity of a program. It can be computed in the following ways:

Lemma 1 Cyclomatic complexity V(G) for a flow graph G is defined as:

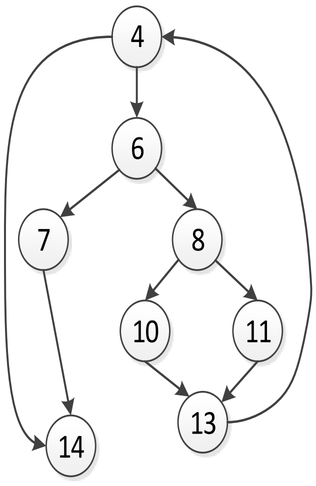
$$V(G) = E - N + 2P \tag{1}$$

where E is number of edges, N is the number of nodes and P is the number of disconnected parts of the graph.

Further, the cyclomatic complexity of any program which is structured with only one entry node and one exit node is consist with the number of decision points (such as 'if' statements or ‘for’ loops) contained in that source code of the program plus one [2][16]:

```
void Sort(int iRecordNum,int iType)
1  {
2      int x=0;
3      int y=0;
4      while(iRecordNum-->0)
5      {
6         if(0==iType)
7         {x=y+2;break;}
8         else
           9          if(1==iType)
10            x=y+10;
11          else
12            x=y+20;
13     }
14}
```

(a) A simple source code



(b) The CFG of (a)

Fig.1 Example

Lemma 2. Cyclomatic complexity V(G) of a control flow graph G with only binary decisions, is defined as:

$$V(G) = b + 1 \tag{2}$$

where b is number of binary decision.

This was proved in [13]. The example in Fig.1 is used to show how cyclomatic complexity is computed.

In Fig.1 (b), using equation (1), we have V(G) = E – N + 2P=10-8+2=4; using equation (2) we obtain V(G) = b + 1=3+1=4, so the results of the two methods are the same.

1.2.3. Basis Paths Generation

Definition 1: A path of CFG: Each path P of CFG G can be represented as a vector V=<n1, n2, … , nk>, where ni is the number of times that edge ei appears in path P. The operations on the paths can be defined as follows [26]. Let P1=<a1, a2, …, ak>, P2=<b1, b2,…, bk>, then

$$P_1 + P_2 = < a_1 + b_1, a_2 + b_2, \ldots, a_k + b_k > \tag{3}$$

$$\beta P_1 = < \beta a_1, \beta a_2, \ldots, \beta a_k > \tag{4}$$

Definition 2: linear combination. A path P2 is said to be the linear combination of path set P1 = {a1, a2, ..., an} (or we say P2 can be linearly represented by P1) if and only if there exist coefficients β1, β2, …, βn such that

$$P_2 = \beta a_1 + \beta a_2 + \ldots + \beta a_n \tag{5}$$

Definition 3: linearly independent paths: A set of paths S is said to be linearly independent if each path in it cannot be linearly represented by other paths in the set.

Based on these two concepts, the definition of basis path set, which is the test set of basis path testing [26], can be given as follows:

Definition 4: Basis path set. A linearly independent path set S of a program is said to be a basis path set if any path Pi of the program's CFG cannot be linearly represented by other paths in S.

The process of basic-path test method is described below:

(1) Derive control flow diagram G of from explicit design or source code. If flow chart is used to describe control structure, it can be mapped to a corresponding CFG. In a CFG, each node is called the node of the flow diagram and represents one or some statements. An arrow in CFG is called edge or connection, represents the control flow and is similar to the arrow in flow chart.

(2) Compute the complexity of CFG G. Let V(G) denote the cyclomatic complexity of CFG G.

(3) Find linearly independent elementary path set. Based on V(G) obtained in (2), there are total V(G) linearly independent paths in G.

(4) Generate test paths, and use them to cover basic path test set.

Currently, there is still lack of tools to automatically and highly efficiently generate basis paths from give source codes or CFG.

1.2.4 *An elementary circuit*

The following definition of elementary circuit is given by Johnson [9].

Definition 5: An elementary circuit: A circuit is a path in which the first and last vertices are identical. A path is elementary if no vertex appears twice. A circuit is elementary if no vertex but the first and last appears twice. Two elementary circuits are distinct if one is not a cyclic permutation of the other.

This excludes loops (edges of the form (v, v)) and multiple edges between the same vertices. We will use an elementary circuit in the later section.

1.3 *Our Major Contributions*

In brief, the major contributions of our work are:

- Proposing an approach for constructing CFG from a given source code (currently in Java), and modifying original CFG to a strongly connected graph (G);
- Proposing an automatic method to generate all basic paths;
- The proposed method can generate all basic paths highly efficiently with computational complexity linear to the number of total edges and nodes in the CFG.

2. Our Proposed Method

In our proposed method, we transform the problem of finding all linear-independent paths of the source code into finding all the elementary circuits of CFG. Before the transforming, we should do a change to the original CFG: we add an edge between the entry and exit node of CFG to make the original CFG (G) a connected graph (G'), the direction of the edge is from the terminal/exit node back to the entry node. Now the graph G' becomes a graph with circuits (or called cycles or connected components). Fig.2 is a sample example of the process.

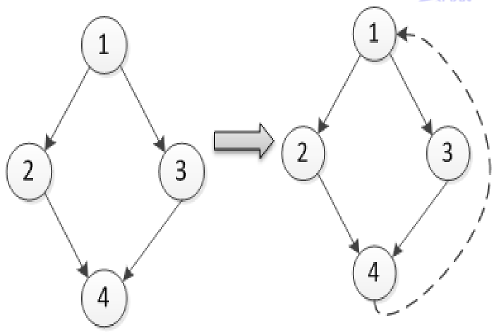


Fig. 2 adding an edge in original CFG

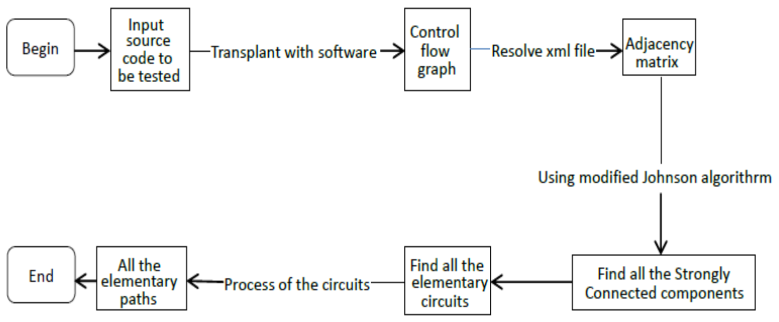
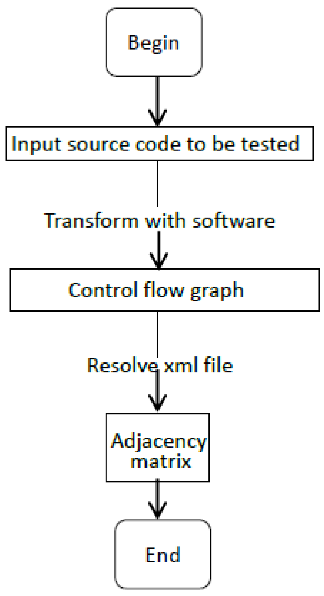

Begin
Input source code to be tested
Transplant with software
Control flow graph
Resolve xml file
Adjacency matrix
Using modified Johnson algorithrm
Find all the Strongly Connected components
Find all the elementary circuits
Process of the circuits
All the elementary paths
End

Fig. 3 Major procedure of proposed method

First of all, we input the source codes to be tested, and transform the source code into control flow graph (CFG G) using the Open Source Plug-in. The CFG is stored in a XML file, and it is going to be resolved by the Java program. So the data in the XML file will be transplant into a two-dimensional array that represents the adjacency matrix of the CFG. Until now we have finished all the jobs for the front-end subsystem. The back-end subsystem firstly adds an edge between the entry and exit node of the directed graph (control flow graph). Then we use modified Johnson algorithm [9] to find all strongly connected components and all the elementary circuits of directed graphs. Finally, we transform the elementary circuits into paths with the method of finding the circuits which begins and ends with start and exit node of the CFG G in an iterative way, and remove the edge we added in G'. Then we can output all the elementary paths as the set of basic paths.

2.1 *Design of the front-end subsystem*

The function of the front-end is mainly transforming source codes into corresponding CFG, and then transform the CFG into the proper data structure (as in this case we choose adjacency matrix) so we can resolve and store it in our system, as it is shown in the Fig.4. There are two parts in the design for the front-end of system: (1) using open source plugin to transform the source codes into CFG as a XML file;(2) Resolve the XML file and store the result in a adjacency matrix, waiting for further process.



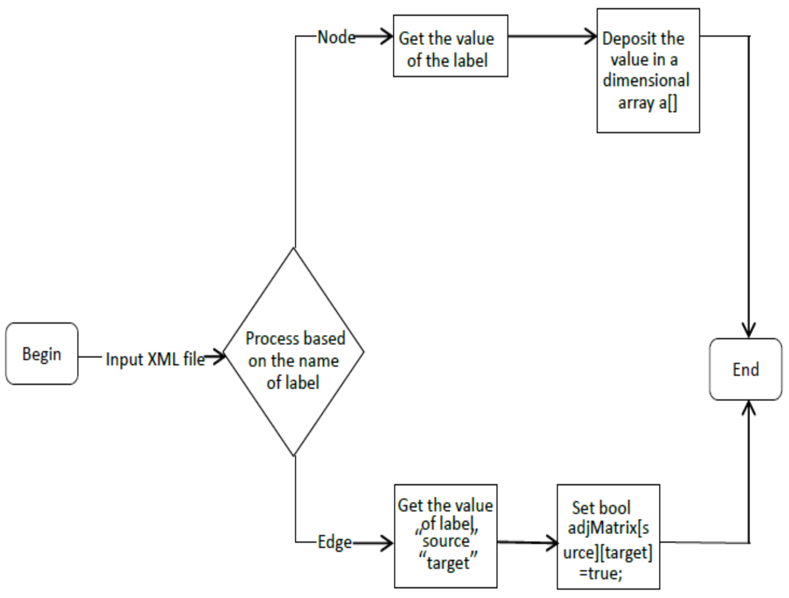

Begin
Input XML file
Process based on the name of label
Node
Get the value of the label
Deposit the value in a dimensional array a[]
Edge
Get the value of label, "source" "target"
Set bool adjMatrix[source][target] =true;
End

Fig. 4 Major procedure of the front-end subsystem

Fig. 5 Flowchart of resolving XML file

2.2 *Transform source codes into control flow graph*

The back-end subsystem needs intermediate results of the front-end subsystem, as in this case is CFG, to resolve. Generation from the source code to the corresponding CFG is similar to the compiler front-end processing: firstly we do some grammar and semantic analysis to the source code in units of lines, and process the result we get from the analysis, draw a CFG according to the actual process of the statement. Since the main aspect of this system is to achieve an efficient path generation method and feasible algorithm in the white box test, we choose an open source third-party software to achieve the function at the front-end.

Through the search and comparison of some third-party open-source tools, we choose open source plugin Dr. Garbage Plugins [28] to implement generation of the CFG of the source code because it is open source and easy to operate. It is installed as a plugin at configured eclipse environment, and then it can automatically generate target CFG of the source code.

2.3 *Resolving the XML file of the CFG*

Dr. Garbage Plugins [28] is an Open source plugin; it has a variety of output formats. In case of convenience of resolving and store, we choose XML file as the form of graph file. Through the resolving of XML file we get some proper data of CFG to do further process. In our system we use a Boolean type two-dimensional array to store the resolving of the XML file. Fig.5 represents the detail design of the resolving process.

XML file has two kinds of labels: Label "node" and Label "edge". Label "node" represents the nodes in the CFG, attribute value "name" represents the values of the nodes, and attribute value "label" represents the line number of the source code to be tested. Label "edge" represents edges of the CFG, each statement corresponds to an edge of the CFG, attribute value "source" and "target", respectively, represents the start and end of the edge. Attribute value "label" represents the several results of branch statements, such as “if” statement has two branches for true and false.

Firstly, the system reads in each line of the XML file when resolving the file, gets the attribute value of label and determines it is a “node” or “label”. If the value of label is “node”, we use a one-dimensional array a[] to save attribute value of the “name”. If the value of label is “edge”, we get the attribute value of “source” and “target” respectively, and set previously defined two-dimensional array with the type of Boolean adjMatrix[source][target] “true”. At the end of the process we should get value of the node according to the line number stored in one dimensional array a[], then we can output all the elementary paths in the path test.

2.4 Design of the back-end subsystem

The back-end subsystem of the project firstly transforms the problem in the white box into the problem of finding all the elementary circuits and strongly connected components to a connected graph in the graph theory.

Firstly, through the front-end subsystem, we have already transformed source code into the CFG, and stored in an adjacency matrix. Now we add an edge between the start and exit node of the directed graph (CFG), the direction of the edge from the exit node to the start node. In this way we transform the graph into a digraph with strongly connected components and elementary circuits. So we can apply our modified Johnson Algorithm (MJA) to find all strongly connected components and all the elementary circuits. The basic circuit processing will be able to get all the elementary path of the source code under test.

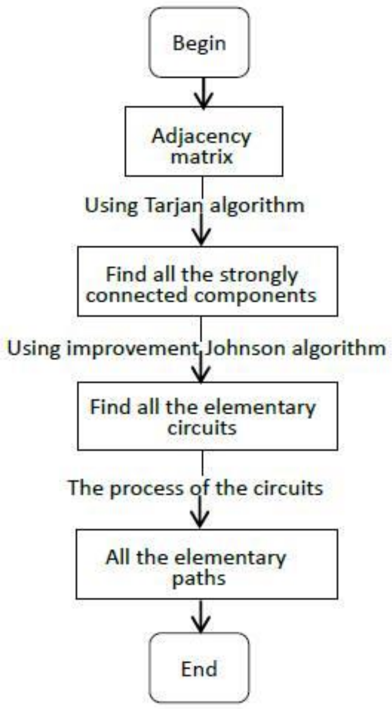

Begin
Adjacency matrix
Using Tarjan algorithm
Find all the strongly connected components
Using improvement Johnson algorithm
Find all the elementary circuits
The process of the circuits
All the elementary paths
End

Fig. 6 procedure of the back-end subsystem

2.4.1 Johnson Algorithm

According to [9], Johnson’s algorithm has two modifications to Tarjan’s algorithm [20]: To avoid duplicating circuits, set a vertex v blocked only under the circumstance of it is added to some elementary path with the beginning of s. It stays blocked as long as every path from v to s intersects the current elementary path at a vertex other than s. Otherwise, only set a vertex as a root vertex when it is the least vertex in at least one elementary circuit for constructing elementary paths. These two characteristics avoid plenty of meaningless searching in [6][19][24].

2.5 Automatic Basic Paths Construction (ABPC) Algorithm

After transforming the CFG of the source code into a connected directed graph by adding an edge to the graph, and applying Johnson algorithm, we can obtain all the elementary circuits.

Note that a basic path starts at the entry node and ends at the exit node of the original CFG. However, direct results from Johnson algorithm may not be able to obtain all basis paths but circuits. Therefore, we need modifications to Johnson algorithm.

2.5.1 Finding elementary circuits

After applying Johnson's algorithm, there are four cases for all the elementary circuits: (1) They are all starting with the entry node, and ending with the exit node at the same time; (2) Some of them are not starting with the entry node while ending with the exit node;(3) Some of them are not ending with the exit node while starting with the entry node;(4) They satisfy (2) and/or (3).

(1) For case (1), we do not have to do anything but just output all the elementary circuits.

```
if(edgeHead!=2&&edgeTail==nail)
// the edge doesn't start with the entry node of the original CFG
{
  adjMatrix[edgeHead][2] = true;
  //add an temporary edge to the CFG
  ElementaryCyclesSearch ecsNew = new ElementaryCyclesSearch(adjMatrix, nodes);
  List cyclesNew = ecsNew.getElementaryCycles();
  for (int i = 0; i < cyclesNew.size(); i++)
  {
    List cycle2 = (List) cyclesNew.get(i);
    int edgeHead1 = Integer.parseInt((String)cycle2.get(0))
    int edgeTail2 = Integer.parseInt((String)cycle2.get(cycle2.size()-1));
    if(edgeHead1!=2|| edgeTail2!=edgeHead) continue;
    //we only focus on the circuits starting with the entry node
          ……
          output all the basic paths
              ……
  }
  adjMatrix[edgeHead][2] = false;
  //recovery to original CFG
          continue;
}
```

Fig. 7 Pseudo Code for Case (2)

(2) Since case (4) is the combination of (2) and (3), so we just focus on the process (2) and (3). If there is an elementary circuit which belongs to (2), we can add an edge to the CFG which starts from the outset of the circuit and points to the entry node of the CFG, then we try to find all the circuits which begin at the entry node of the CFG and end with outset of the specific circuit we mentioned before. Pseudo code is in Fig.7.

(3) Similarly, we can also use this method for case (3). At this time we add an edge to the CFG that starts from the exit of the CFG and points to the exit of the circuit.

(4) Finally, we combine the circuits we have just found from (2) and (3). Then we can output them in non-decreasing order of the number of vertices in the paths.

Then we obtain all the elementary paths starting at the entry node and ending with the exit node of the CFG.

Now we use an example to show our algorithm. As depicted in Fig. 8, we have already found partial path of the CFG, 2->3->5 and 2->4->5 using Johnson algorithm. So we need to add an edge from node 2 to 1, i.e., 2->1, to find paths from the outset of CFG and to the outset of the circuits (in this case, from 1 to 2). After that we can use the same method to find paths from the exit of circuits and to the exit node of the CFG (in this case, from 5 to 6). Finally we combine the partial paths from the former two steps, then we get all the elementary paths: 1-2-3-5-6 and 1-2-4-5-6.

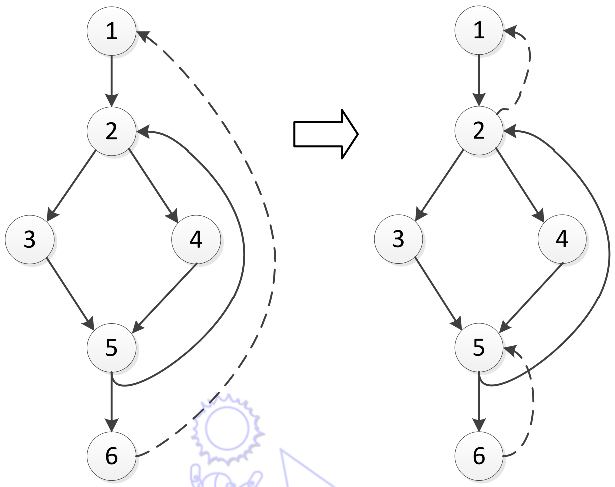


Fig.8 A case of process for all the elementary circuits

In this way we finish the process of finding all the elementary circuits. We observe that there may be redundant paths in the elementary circuits we just obtained, therefore we design a method to remove all redundant paths as follows.

2.5.2 *Removing redundant paths and infeasible paths*

The APBC algorithm may generate redundant paths, so we figure out a way to solve the problem. We use a two-dimensional array to represent matrix of CFG and value "1" represents corresponding edge. When value of the array equals "1" which means the edge of the original CFG is not visited. According to McCabe's theory [13, 15], a path is a new path if and only if there's a new edge of the path that is not present in the existing paths. So when we output each path, we go through every edge of the path to check whether there's a new edge in the path. If a path is not a new one, we drop it instead of output it. Obviously the first path must be a new path, so we find each edge of the path and set the corresponding value in the array as "0". Next, from the second path, we add all the value in the array of corresponding edges in the path to check whether there's a new edge in the path. If the result is "0", that means no new edge is in the path, and we drop the path. Until all the paths have been checked, we get the all the basis path as McCabe' theory suggests.

We observed that most of CFGs generated by Dr. Garbage Plugins [28] could void infeasible paths (paths which may not be testable). We also can apply methods introduced in [5, 26,32,33] to remove infeasible paths.

2.6 *Computational Complexity of ABPC algorithm*

Theorem 1 the computational complexity of ABPC algorithm is O((n + e)(c + 1)) and space bounded by O((n + e)(c + 1)).

Proof: Detailed proof is explained in [9], here we provide a sketch. Johnson algorithm finds all the elementary circuits of a directed graph in time bounded by O((n + e)(c + 1)) and space bounded by O(n + e), where there are n vertices, e edges and c elementary circuits in the graph. In our ABPC algorithm, we use JOHNSON's algorithm to calculate and output all the elementary circuits, and adding post-process to find all basic paths which does not increase time complexity, so the computational complexity of ABPC algorithm is also O((n + e)(c + 1)) and space bounded by O((n + e)(c + 1)).

3. Performance Evaluation

In this section, performance evaluation of some typical examples is provided for our proposed methods.

Example 1: The following example program is to calculate the effective number of the students' scores, the total score and average score for up to 50 input numbers ("-1" as the end symbol of input).

3.1 *The expected results*

Step 1: Obtaining the corresponding CFG by the program flow graph (Fig.9.).

Step 2: Determining the cyclomatic complexity V(G)：

(1) V(G)= 6 (by areas)

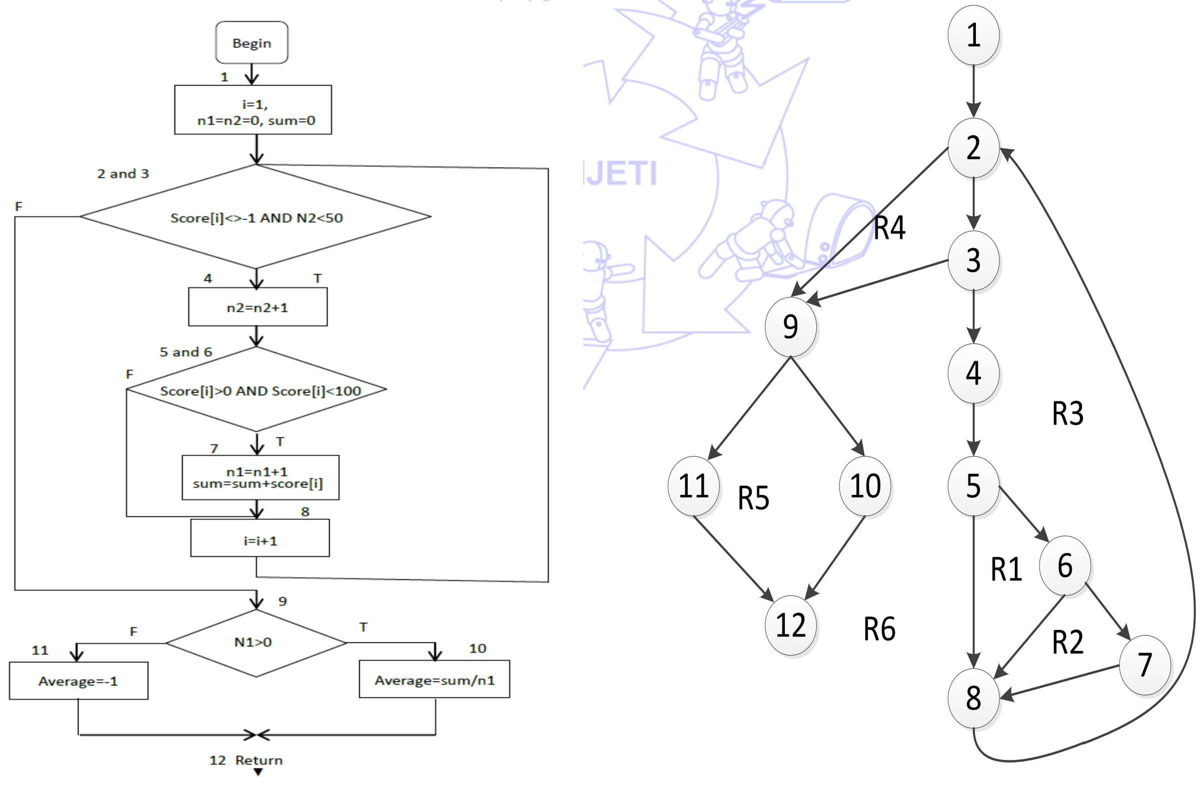
(2) V(G)=E-N+2=16-12+2=6（"E" represents number of edges and "N" represents number of nodes)

(3) V(G)=P+1=5+1=6 （P represents number of binary decision, in this case, 2、3、5、6、9 are binary decision points).

Step 3: Determining the elementary path set. (There are 6 independent paths in this case)

Path 1：1-2-9-10-12; Path 2：1-2-9-11-12; Path 3：1-2-3-9-10-12;

Path 4：1-2-3-4-5-8-2…Path 5：1-2-3-4-5-6-8-2…Path 6：1-2-3-4-5-6-7-8-2…



(a) (b)

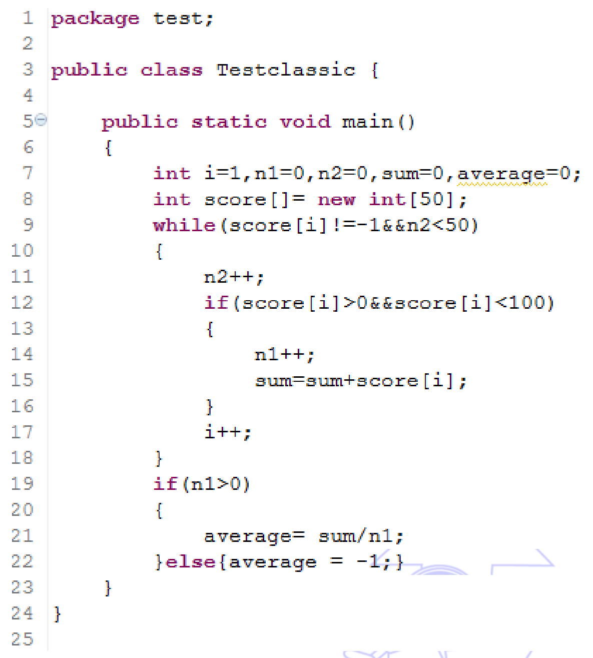
Fig. 9 Transform from Flow graph (a) to the CFG (b) of the average score program

3.2 *The results by our proposed method*

First step, input the source code to be tested into a new eclipse project, the result is shown in the Fig.10.After that, we will generate the CFG of source code with Dr. Garbage Plugins. Firstly, we select the project just input, and then select the "Create Graphs for selected Class" Option after right click the project, choose "Create Source code Graphs" of the menu. In the pop-up dialog window we first select the folder where we putted the class file to be tested, then we choose the "Graph XML" as the format of the control flow graph.

Fig.11 shows the CFG in the format of ".graph", while Fig.12 shows the CFG in XML format. Since now we have already finished all the task of the front-end of the system.In Fig.14, there are more than one "true" or "false" for a node, or sometimes there exists "false" and "true" at the same time for a node. It means in one branch statement there are multiple judgment statements, so the "true" and "false" are listed in order by each judgment statement.

Finally, we run the back-end subsystem to resolve the XML file and get all the elementary circuits of the CFG, and finally we obtain all six basic paths in Fig. 13 after post-processing. These results are same as in expected results.



```
 1 package test;
 2
 3 public class Testclassic {
 4
 5     public static void main()
 6     {
 7         int i=1,n1=0,n2=0,sum=0,average=0;
 8         int score[]= new int[50];
 9         while(score[i]!=-1&&n2<50)
10         {
11             n2++;
12             if(score[i]>0&&score[i]<100)
13             {
14                 n1++;
15                 sum=sum+score[i];
16             }
17             i++;
18         }
19         if(n1>0)
20         {
21             average= sum/n1;
22         }else{average = -1;}
23     }
24 }
25
```

Fig. 10 Java source code of average score

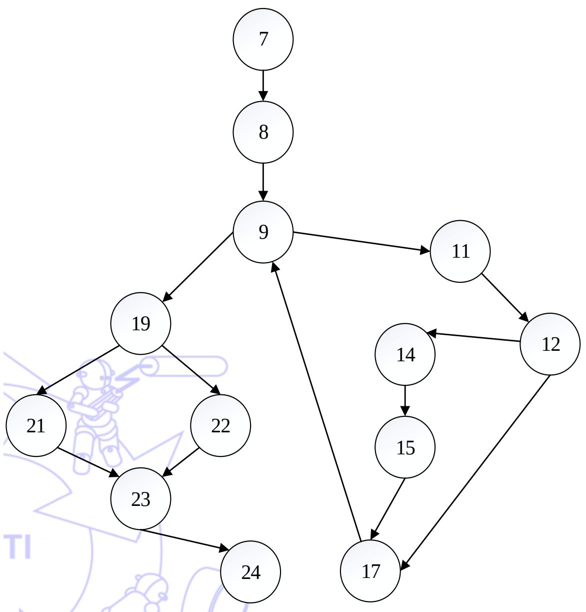
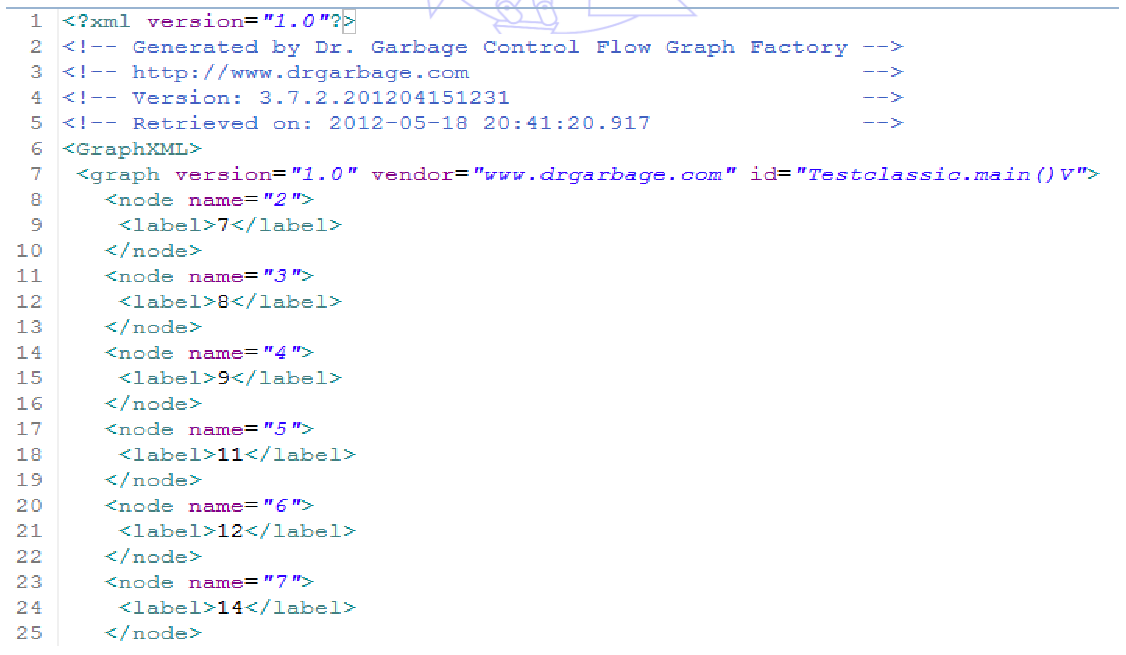
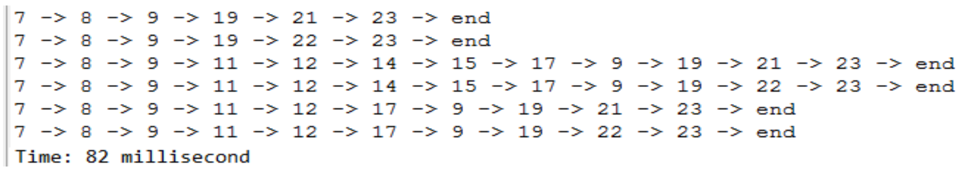


Fig. 11 The CFG of Fig. 10



```
<?xml version="1.0"?>
<!-- Generated by Dr. Garbage Control Flow Graph Factory -->
<!-- http://www.drgarbage.com                          -->
<!-- Version: 3.7.2.201204151231                       -->
<!-- Retrieved on: 2012-05-18 20:41:20.917             -->
<GraphXML>
 <graph version="1.0" vendor="www.drgarbage.com" id="Testclassic.main()V">
   <node name="2">
    <label>7</label>
   </node>
   <node name="3">
    <label>8</label>
   </node>
   <node name="4">
    <label>9</label>
   </node>
   <node name="5">
    <label>11</label>
   </node>
   <node name="6">
    <label>12</label>
   </node>
   <node name="7">
    <label>14</label>
   </node>
```

Fig. 12 XML file represents CFG for average score program



```
7 -> 8 -> 9 -> 19 -> 21 -> 23 -> end
7 -> 8 -> 9 -> 19 -> 22 -> 23 -> end
7 -> 8 -> 9 -> 11 -> 12 -> 14 -> 15 -> 17 -> 9 -> 19 -> 21 -> 23 -> end
7 -> 8 -> 9 -> 11 -> 12 -> 14 -> 15 -> 17 -> 9 -> 19 -> 22 -> 23 -> end
7 -> 8 -> 9 -> 11 -> 12 -> 17 -> 9 -> 19 -> 21 -> 23 -> end
7 -> 8 -> 9 -> 11 -> 12 -> 17 -> 9 -> 19 -> 22 -> 23 -> end
Time: 82 millisecond
```

Fig. 13 Output for all the elementary paths in average score program

3.3 *Scalability Test Examples*

In order to verify the accuracy and efficiency of our method, we have conducted following scalability tests.

3.3.1 *A complete directed graph for scalability test*

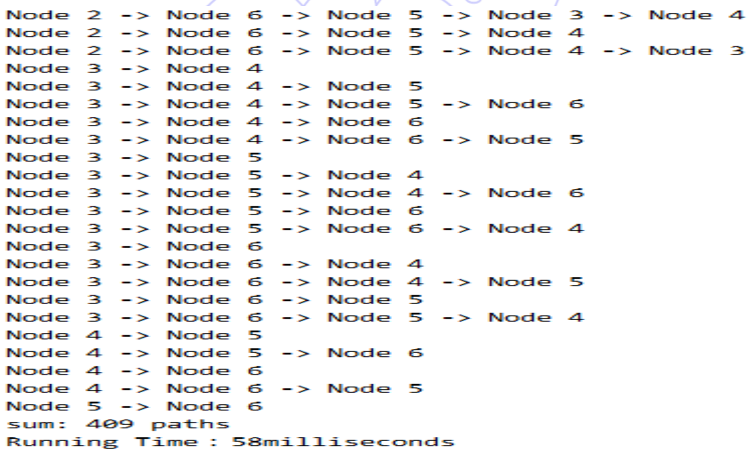
We use complete directed graph for scalability test since the number of elementary circuits (basis paths) are known.

Definition 6. ***A directed graph G=(V,E) consists of a nonempty and finite set of vertices V and a set E of ordered pairs of distinct vertices called edges. There are N vertices and E edges in G.***

For a complete directed graph with N nodes, the number of basis paths is:

$$V(G)=\sum_{i=1}^{N-1}\binom{N}{N-i+1}(N-i)! \qquad (6)$$

The number of elementary circuits in a directed graph can grow faster with N than the exponential 2^N.We firstly test on complete directed graphs with the vertices of 6, and record the running time and number of paths.



```
Node 2 -> Node 6 -> Node 5 -> Node 3 -> Node 4
Node 2 -> Node 6 -> Node 5 -> Node 4
Node 2 -> Node 6 -> Node 5 -> Node 4 -> Node 3
Node 3 -> Node 4
Node 3 -> Node 4 -> Node 5
Node 3 -> Node 4 -> Node 5 -> Node 6
Node 3 -> Node 4 -> Node 6
Node 3 -> Node 4 -> Node 6 -> Node 5
Node 3 -> Node 5
Node 3 -> Node 5 -> Node 4
Node 3 -> Node 5 -> Node 4 -> Node 6
Node 3 -> Node 5 -> Node 6
Node 3 -> Node 5 -> Node 6 -> Node 4
Node 3 -> Node 6
Node 3 -> Node 6 -> Node 4
Node 3 -> Node 6 -> Node 4 -> Node 5
Node 3 -> Node 6 -> Node 5
Node 3 -> Node 6 -> Node 5 -> Node 4
Node 4 -> Node 5
Node 4 -> Node 5 -> Node 6
Node 4 -> Node 6
Node 4 -> Node 6 -> Node 5
Node 5 -> Node 6
sum: 409 paths
Running Time : 58milliseconds
```

Fig. 14 Output of the test case with all the elementary circuits

As we can see in Fig.14, we output part of total 409 paths with the running time of 58 milliseconds. To provide accuracy and efficiency of different number of vertices in complete graph, we then test larger number of nodes and summarize results in Table 1, all results are obtained by using a Pentium PC with 2Ghz CPU and 2G memory.

Table 1 Number of basis paths in directed graph (for scalability test)

Number of Nodes	Number of Paths	Running Time
		(micro seconds)
4	20	9
5	84	21
6	408	58
7	2365	149
8	16064	974
9	125664	15065

4. Conclusions

In this paper, we propose an efficient method to find basis paths in white-box testing automatically, with time complexity linear to the number of nodes and edges in the control flow graph. Through performance evaluation of many examples, it is shown that the proposed method is correct and scalable to very large test cases. The proposed method can be applied to basis path testing easily. We also provide a preliminary website (http://121.49.110.25:8088/NewPathBasedTest/) for this tool. There are several research directions for future research:

(1) Extending to other programming languages. Currently, we are able to generate CFG and basis paths for Java programs from source codes. For other programming language such as C, C++ etc., similar approach should be developed.

(2) Combining methods of removing infeasible paths from generated basis paths. It is observed by some researchers such as in [26, 29], that some paths in the basis path set may be not feasible for real testing. We observed that most of CFGs generated by Dr. Garbage Plugins [28] could void infeasible paths (paths which may not be testable). We also can apply methods introduced in [5, 26] to remove infeasible paths. We are conducting research for finding feasible paths efficiently and will combing this method into our main tool.

(3) Investigating automatic test data generation and test results generation for basis path testing. To make basis path testing complete, it is important to also generate test data and results automatically for the generated basis paths.

Combining all these issues and solutions, comprehensive methods for basis path testing and white-box testing can be developed.

References

[1] D. Berndt, J. Fisher, L. Joshson, “Breeding Software Test Cases with Genetic Algorithms,” Proceedings of the 36th Hawaii International Conference on System Sciences, Big Island, Hawaii, USA, 2003.

[2] D. J. Berndt, A. Watkins, “Investigating the Performance of Genetic Algorithms-Based Software Test Case Generation,” Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering ,Tampa Florida, USA, pp. 261-262, 2004.

[3] A. G. Holzman, A. Kentand, J. Belzer, Encyclopedia of Computer Science and Technology, CRC Press, pp. 36-367, 1992.

[4] 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, May 2009. (in Chinese)

[5] Q. F. Du, D. Xiao, “An improved algorithm for basis path testing,” the Proceedings of 2011 International Conference on Business Management and Electronic Information, vol. 3, pp. 175-178, 2011.

[6] A. Ehrenfeucht, L. Fosdick, L. Osterweil, An algorithm for finding the elementary circuits of a directed graph, Tech. Rep. CU-CS-024-23, Department of Computer Science, University of Colorado, Boulder, 1973.

[7] M. Evangelist, “An Analysis of Control Flow Complexity,” Eighth International Computer Software and Application Conference, pp. 328-396, 1984.

[8] F. Harary, E. Palmer, Graphical Enumeration, Academic Press, New York, 1973.

[9] D. B. Johnson, “Finding All the Elementary Circuits of A Directed Graph.SIAM Journal on Computing,” vol. 4, no.1, pp. 77-84, Mar. 1975.

[10] B. F. Jones, H. H Sthamer, D. E. Eyres, “Automatic Structural Testing Using Genetic Algorithms,” Software Engineering Journal, vol. 11, no. 5, pp. 299-306, 1996.

[11] A. G. Li, “Automatic Generating All-Path Test Data of a Program Based on PSO,” WRI World Congress on Software Engineering (WCSE '09), vol. 4, May 2009.

[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] T. McCabe, “A Complexity Measure,” IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

[14] G. Myers, “An Extension to the Cyclomatic Measure of Program Complexity,” SIGPLAN Notices, Oct. 1977.

[15] T. McCabe, “Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Measure”, NBS Special Publication, pp. 99-500, Dec. 1982.

[16] G. Myers, The Art of Software Testing. 2ed. John Wiley & Son. pp. 234, 2004

[17] J. Poole, “A Method to Determine a Basis Set of Paths to Perform Program Testing,” NISTIR 5737, Nov. 1995.

[18] M. Prasanna, S. N. Sivanandam, R. Venkatesan, R. Sundarrajan, “A Survey on Automatic Test Case Generation,” Academic Open Internet Journal, 2005.

[19] J. C. Tiernan, An efficient search algorithm to find the elementary circuits of a graph, Comm. ACM, pp. 722-726, 1970.

[20] R. Tarjan, “Depth-first search and linear graph algorithms,” SIAM Journal on Computing, vol. 1, no. 2, pp. 60-146, June 1972.

[21] R. Tarjan, “Enumeration of the elementary circuits of a directed graph, SIAM Journal on Computing,” vol. 2, no. 3, pp. 16-211, Sept. 1973.

[22] L. J. White, B. Wiszniewski, “Path Testing of Computer Programs with Loops using a Tool for Simple Loop Patterns,” Software - Practice and Experience, vol. 21, no. 10, pp. 102-1075, Oct. 1991.

[23] J. Wegener, A. Baresel, H. Sthamer, “Evolutionary Test Environment for Automatic Structural Testing,” Information and Software Technology, vol. 43, no. 14, pp. 841-854, 2001.

[24] H. Weinblax, “A new search algorithm for finding the simple cycles of a finite directed graph,” Journal of the Associaition for Computing Mechinery, vol. 19, no. 1, pp. 43-56, 1972.

[25] W. Xin, J. W. Cao, “Computing Strongly Connected Components Using Kosaraju algorithm,” vol. 31, no. 4, pp. 05-691, Feb. 2010.

[26] W. Xin, J. W. Cao, “Computer Engineering and Design,” vol. 31, no. 4, pp. 05-691, Oct. 2007.

[27] J. Yan, J. Zheng, “An efficient method to generate feasible paths for basis path testing,” Information Processing Letters, vol. 107, no. 3-4, pp. 87-92, Jul. 2008.

[28] G. Plugins, “Control Flow Graph Factory,” http://www.drgarbage.com/control-flow-graph-factory.html, 2008.

[29] S. Zhang, “Automatic Path Test Data Generation Based on GA-PSO,” In 2010 IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS), Oct. 2010.

[30] A. S. Ghiduk, O. Said, S. Aljahdali, Basis Test Paths Generation Using Genetic Algorithm, Proceedings of ICCIT, 2012.

[31] M. Papadakis, N. Malevris, “An Effective Path Selection Strategy for Mutation Testing,” APSEC 2009, pp. 422-429, 2009

[32] 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.

[33] M. Papadakis, N. Malevris, “A Symbolic Execution Tool Based on the Elimination of Infeasible Paths,” ICSEA 2010, pp. 435-440, 2010.

[34] M. N. Ngo, H. B. Tan, “Heuristics-based infeasible path detection for dynamic test data generation,” Information & Software Technology, vol. 50, no. 7-8, pp. 641-655, 2008