Test Data Generation Based on Test Path Discovery Using Intelligent Water Drop

Praveen Ranjan Srivastava, Birla Institute of Technology and Science, Pilani, India
Amitkumar Patel, Birla Institute of Technology and Science, Pilani, India
Kunal Patel, Birla Institute of Technology and Science, Pilani, India
Prateek Vijaywargiya, Birla Institute of Technology and Science, Pilani, India

ABSTRACT

Automatic test data generation is required to generate test cases dynamically for a specific software program. Manual generation of test data is too tedious and a time consuming task. This paper proposes a technique using Intelligent Water Drop (IWD) for automatic generation of test data. Correctly generated test data helps in reducing the effort while testing the software. This paper discusses different algorithms based on IWD to generate test data and path coverage over Control Flow Graph. Test data is generated keeping in mind all of the programming constraints like “if,” “while,” “do while,” etc., available in the program.

Keywords: Control Flow Graph (CFG), Intelligent Water Drop (IWD), Software Testing, Test Data Generation, Test Path Discovery

INTRODUCTION

In software engineering area, software testing is a crucial phase (Kewen, Zilu, & Wenyong, 2009) to assess the quality of the software. Software testing contains almost 50% of total cost of the software development (Edvardsson, 1999). Aim of the software testing is to uncover errors and faults present in the program, so that customer requirement can be properly fulfilled. Testing phase includes in the review of specification, analysis, design, and implementation part of the Software Development Life Cycle (SDLC). Manual generation of test data for testing the program, results in low reliability and high cost (Li & Lam, 2005; Pressman, 2010). Due to the lack of cost and reliability, automation of testing process is necessary, so that the cost of testing can be reduced.

Software engineering is a knowledge-intensive activity, which requires access and manipulation of large quantity of information about the project domain (Naaz Raza, 2009). By managing these facts manually, cost becomes very high. Artificial Intelligence (AI) based techniques can help in removing this situation.
AI based technique helps in solving the problem by using fast and proper judgments rather than using step by step deduction (Naaz Raza, 2009). From the last decade various techniques like genetic algorithm, ant colony optimization and other metaheuristic techniques have been introduced for generating test path and test data (Korel, 1990; Michael et al., 1997; Michael & McGraw, 1998; McMinn, 2004; Kewen et al., 2009; Srivastava, Baby, & Raghurama, 2009). The main issues of various metaheuristic techniques (McMinn, 2004) are optimal test data generation and complete software coverage. Another new optimization technique, Intelligent Water Drop (IWD) (Shah-Hosseini, 2009), an algorithm which is based on swarm optimization techniques, can be used for generation of test data. IWD algorithm works similar to the natural water drop of a river bed.

This paper applies IWD algorithm to generate test data and also discover existing paths in the program. This paper is structured as follows; first, we describe the background work of software testing. Then, we present the basic IWD optimization approach. The next section applies IWD approach to generate test path and sequence generation. We then describe the case study of the suggested approach for test data generation, while the next section discusses the analysis part. Finally we conclude the paper along with future scope of the applied approach.

BACKGROUND

Various techniques (Korel, 1990; Pedrycz & Peters, 1998; Briand, 2002; McMinn, 2004; Harman, 2007; Kewen et al., 2009; Srivastava et al., 2009; Srivastava & Baby, 2010) have been proposed for automated testing to reduce efforts to a remarkable extent. Swarm optimization techniques are widely used for test data generation. Ant Colony Optimization (ACO) is one of them (Li & Lam, 2005; Kewen et al., 2009; Srivastava et al., 2009). Automatic test data generation based on ACO (Kewen et al., 2009) has introduced a model to generate test data using the branch function technique. It has solved the problem of local optimization, but this approach is applicable only for numeric data types and also the model is not suitable for object oriented programs and other types of input behavior. Software test data generation using ACO (Li & Lam, 2005) has also proposed a solution for state-based software testing but complete coverage is not possible by the proposed approach.

Genetic algorithm has been applied for dynamic test data generation (Michael et al., 1997) and the results are pretty impressive over random or exhaustive test data generation, but this model was not applicable for boolean and string type of variables.

Another approach is suggested for generation of test data using genetic algorithm and hamming distance concept (Srivastava et al., 2010) but optimal test data are uncertain, and stuck in the local optima and explore more repetitive paths. An approach for software testing has been recommended (Srivastava et al., 2009) in which ACO procedure is used to discover effective paths which are promising for full path coverage.

Genetic algorithm and tabu search have been applied on control-dependence graph (Rathore et al., 2011). This approach uses only the number of common predicates with the target node for finding objective value, but few important factors such as distance from initial and final node, criticality, etc., that it fails to take into account.

Many papers (Korel, 1990; McMinn, 2004; Srivastava et al., 2009) have suggested techniques for full path coverage and other have suggested approaches for optimal test data generation (Li & Lam, 2005; Srivastava et al., 2008; Kewen et al., 2009; Srivastava et al., 2010), but none of them have suggested combined approaches for tackle both the problems together though they are very closely related. This paper discusses the newly emerging approach for combined solutions to both of these problems together.

IWD algorithm (Shah-Hosseini, 2007; Shah-Hosseini, 2009) is a swarm-based optimization algorithm, simulated from observing
natural water drops in river. IWD has been applied to various problems like Travelling Salesman Problem (TSP), N-queen puzzle, Multidimensional Knapsack Problem (MKP), etc. These results have proved the significance of IWD algorithm over other swarm optimization algorithms. Another solution for TSP using IWD algorithm (Shah-Hosseini, 2007) is introduced where proposed algorithm converges very fast to the optimum solution.

The improved IWD algorithm (Duan, Liu, & Lei, 2008) has been applied to solve the air robot path planning in dynamic environments and results are quite impressive over genetic algorithm and ACO algorithm.

Since IWD has not yet been applied to the area of software testing and the effective results have been produced for various problems, this paper tries to derive a solution model for software testing using IWD in the hope that expected results will be more significant than the current solutions available for test data generation.

Before moving to the proposed solution of IWD, general introduction is provided which describes its strategy along with available metrics in it.

INTELLIGENT WATER DROP

IWD algorithm is a new swarm-based optimization algorithm inspired from natural rivers. In a natural river, water drops move towards centre of the earth, due to some gravitational force acting on it. Due to this the water drop follows the straight and the shortest path to its destination (Shah-Hosseini, 2009). Pictorial representation of basic IWD is shown in Figure 1. In ideal conditions it is observed that the optimal path will be obtained. Water drop flowing in the river has some velocity which is affected by another actor, i.e., soil.

Some changes that occurred while transition of water drop from one point to another point are:

1. Velocity of water drop is increased.
2. Soil content in the water drop is also increased.
3. Amount of soil in the riverbed from source to destination get decreased.

Water drop in the river picks up some soil in it when its velocity gets high and it releases the soil content when its velocity is less (Shah-Hosseini, 2007)

Some of the prominent properties of the natural water drop are taken, based on which IWD is suggested. IWD has the two following important properties,
The amount of soil the water drop carries, which is represented by Soil (IWD) or \( \text{soil}^{\text{IWD}} \).

2. The velocity of water drop with which it is moving now, denoted by Velocity (IWD) or \( \text{vel}^{\text{IWD}} \).

Value of both the properties may change during the transition. Environment contains lots of paths from source to destination (Duan et al., 2008) which may be known or unknown. When the destination is known, IWD follows the best path to reach the destination (best is in terms of cost and any other desired measure). When destination is unknown it finds the optimal destination.

From the current location to the next location Velocity (IWD) is increased by an amount, which is nonlinearly proportional to the inverse of the amount of soil between the two locations, referred to as the change in velocity. The Soil (IWD), is also increased by extracting some soil of the path between two locations. The amount of soil added to the IWD is inversely (and nonlinearly) proportional to the time needed for the IWD to pass from its current location to next location. IWD chooses the path with less soil content.

In the proposed approach, IWD is applied over the Control Flow Graph (CFG) to obtain the number of paths available in the program. The CFG depicts the logical control flow of the program (Pressman, 2010). All linearly independent paths could be obtained by CFG. Independent path is the path in the program that determines at least one new set of processing statement. In other words it introduces at least one new edge in the graph. Number of available paths can be obtained by finding the cyclomatic complexity of the graph (Pressman, 2010).

**PROPOSED SOLUTION**

Software testing is a mandatory phase of software life cycle model but it increases the time duration as well as cost of the development. It should avoid the exhaustive search for generating the test cases. This paper tries to implement the solution model which is useful for generating test data for program dynamically. The proposed algorithm using IWD is shown in Figure 2.

The proposed model for test data generation consists of two major phases. In the first phase Lex and Yacc (Brown, Levine, & Mason, 1995) parsing tool is used to generate output file which contains the details about control flow graph. It also generates the predicate node along with its existing conditions. Second phase takes output file of the first phase as an input which contains the control flow graph.

Second phase works in three steps.

STEP 1: Find out cyclomatic complexity (CC) to each node of CFG.

STEP2: Extract all possible paths for traversing the CFG from start node to end node using proposed IWD algorithm.

STEP 3: Generate test data for all independent paths.

For processing the noted steps, three algorithms, namely, Algorithm 1, Algorithm 2 and Algorithm 3, are suggested. Algorithm 1 describes the working of Step-1 which assigns cyclomatic complexity to each node of the CFG. Cyclomatic complexity of a node is obtained from Algorithm 1 which shows the number of paths available from the current node to reach the end node of CFG. This information is fed into Algorithm 2, which extracts all the independent paths of the CFG using IWD algorithm.

From the algorithm, all available independent paths are obtained. Now, in Algorithm 3, test data is generated for each of the independent path.

In Algorithm 3, all individual paths are collected and for each particular path, test data is generated. Evaluation of the proposed algorithm is discussed below. Depth analysis of proposed approach, few case studies have been wisely chosen such that comparison studies can be done properly. For experimental study a tool has been developed for which snapshots are available in the Appendix (Figures 9 through 11).
EXPERIMENTAL STUDY

This section shows the result of the execution of algorithm, in order to determine the effectiveness and feasibility of the algorithm. Algorithms demonstrate that it is a reasonably accurate technique in terms of the paths covered using IWD and test data generation, as compared to the exhaustive methods. The proposed IWD approach is implemented using Java programming environment on an Intel Pentium 4 system running Windows XP Service Pack-2. In experimental study, IWD is applied on two programs, one is ‘triangle classifier problem’ (Classical

Algorithm 1. Assign CC to each node of CFG

Initialization:
Global array visited [N]
/* empty array of size N, where N = number of nodes in the graph */
Input: root node of Graph(N,E)
/* Graph(N,E) = Graph containing N nodes and E edges */
Output: cyclomatic_complexity of each node
Step 1. If node is already visited OR node is an end node then return 1;
Step 2. Add node to visited list;
Step 3. Loop for all branches
Step 3.1 Increment result by cyclomatic complexity of noderesult += cyclomatic_complexity(node);
    /* cyclomatic complexity is calculated for the given node */
    Goto Step-1
Step 3.2 End loop
Step 4. return result;
Algorithm 2. Extract all possible paths using IWD

Initialization:
Max_Iteration - Maximum time Iteration to be performed
Init_Soil - Initial soil on the path
soil(i,j) - Available amount of soil between node (i) and node (j)
Visited_Path - Contains information about node that has been visited
paths(i) - Number of path from node (i) yet to be explored
p(i,j) - Probability of choosing path between node (i) and node (j)

Input: Graph(N,E), cyclomatic_complexity of each node
Output: Path_List = Contains all extracted paths (initially empty list)

Step 1. Initialize all static parameters (from Step-1.1 to Step-1.8)
Step 1.2 Max_Iteration = CC which we have found using Algorithm 1
Step 1.3 Velocity updating parameters
   a_v = 1, b_v = 0.01, c_v = 1 and α = 1
   Here, the a_v, b_v, c_v and α are user-selected positive parameters.
Step 1.4 Soil updating parameters
   a_s = 1, b_s = 0.01 and c_s = 1
   Here, the a_s, b_s and c_s are user-selected positive parameters.
Step 1.5 Initial soil on each edge of graph
   Init_Soil = 10000, (It is assigned to each edge, i.e., soil(i,j) = Init_Soil)
Step 1.6 For all IWD, Soil(IWD) = 0
Step 1.7 Initial velocity of each IWD
   Init_Vel = 200 (It is assigned to each drop)
Step 1.8 Visited_Path = empty list
Step 2. Put IWD on root node of the graph(CFG)
Step 3. Calculate probability for choosing next node (j) from available path
   of node (i).
Step 3.1 Probability can be find using formulae
   \[ p_{i,j} = \frac{\text{paths}(i)}{\text{paths}(j)} \times \frac{\text{soil}(i,j)}{\text{Init}_\text{Soil}} - \frac{\sum_{k \neq i} \text{soil}(i,k)}{\text{No.of k}} \]
   where, paths (i) = number of path from node (i) yet to be explored (CC(i)),
   \[ \sum_{k \neq i} \text{soil}(i,k) = \text{Sum of the soil of every path i to k, i\neq k}. \]
   Probability formulae is used for finding probability of a path when there are two nodes are avail-
   able for moving forward from the current node (i). This function can also
tackle the blocked path situation. Along with that for handling the situation
of paths having same CC, the concept of soil has also been introduced in the
fitness function which is likely to be varied for the different paths
Step 3.2 Using the formulae as mentioned in Step-3.1, find probability for all
   outgoing paths from the current node (i).
Step 4. Choose next path which is having greatest probability because it is
   the optimal path where many other paths are yet to be explored, e.g.,
   \[ p(1,2) > p(1,3), \]
   then choose path(1,2) and add it to the visited path
   list.
Step 5. Update the Velocity(IWD) (denoted by vel\textsuperscript{IWD}) moving from node (i) to
   node (j).
   \[ \text{vel}_{\text{IWD}}(t+1) = \text{vel}_{\text{IWD}}(t) + \frac{a_v}{b_v + (c_v \times \text{soil}\textsuperscript{IWD}(i,j))} \]
   where, vel\textsuperscript{IWD} (t+1) is the updated velocity of IWD
Step 6. Update time parameter for IWD.
   \[ \text{time}(i,j,\text{vel}_{\text{IWD}}(t+1)) = \frac{\text{HUD}(j)}{\text{vel}_{\text{IWD}}} \]
continued on the following page
problem used in many research work) (Kewen et al., 2009; Edvardsson, 1999) and another is ‘maximum of three inputs.’

In the study-1, IWD is applied on the triangle program (Program 1). This program takes 3 sides of triangle as input, and classifies the triangle as equilateral, isosceles, scalene and not a triangle. Program-1 shows triangle problem and the corresponding CFG which contains the nodes and flows between them is shown in Figure 3.

In the CFG, individual cyclomatic complexity for each node is assigned by the Algorithm 1 as given in Table 1.

As explained in Algorithm 1, number of branches is calculated recursively. It means that for finding the number of branches for Node-1,
the branch value is calculated first for each child node.

For example, the branch value for Node-8 is 4. Now for calculating this value, branch values for the nodes 9, 10, 11, 12, 13, 14 and 15 needs to be calculated. Then sum up only the branch values of immediate children of Node-8, which gives the branch value for Node-8. Immediate children for Node-8 are Node-9 and Node-12. Branch values for Node-9 and Node-12 are 2 and 2, respectively. The branch value for Node-8 is 4 (2+2).

Algorithm 2 finds out the entire individual paths available in the program. Experimental result for the exploration of Path-1 is given in

<table>
<thead>
<tr>
<th>Node</th>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. Cyclomatic complexity of each node for the triangle problem

Program 1. Triangle Problem

```c
int TriType(int i, int j, int k)
{
    1.     if (i <= 0  || 2.     j <= 0  || 3.     k <= 0  || 4.     i+j-k <= 0  || 5.     i+k-j <= 0  || 6.     j+k-i <= 0)
          7.         type = Not a Triangle 8. else if (i == j){
                  9.             if (j == k)
                          10.          type = Equilateral 11.        else          type = Isosceles 12. else{ if (j == k)
                            13.              type = Isosceles 14.            else              type = Scalene}
                  15. return type;
}
```
Figure 3. CFG for triangle problem

Table 2. Experimental result of triangle problem for path exploration

<table>
<thead>
<tr>
<th>Path</th>
<th>Probability</th>
<th>Chosen Path</th>
<th>Velocity of IWD</th>
<th>Time of IWD</th>
<th>Change In Soil</th>
<th>Soil of IWD</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2)</td>
<td>0.9</td>
<td>(1,2)</td>
<td>200.01</td>
<td>0.449</td>
<td>1.468</td>
<td>1.468</td>
</tr>
<tr>
<td>(1,7)</td>
<td>0.1</td>
<td>(1,2)</td>
<td>200.01</td>
<td>0.039</td>
<td>1.556</td>
<td>3.0254</td>
</tr>
<tr>
<td>(2,3)</td>
<td>0.888</td>
<td>(2,3)</td>
<td>200.01</td>
<td>0.035</td>
<td>1.662</td>
<td>4.687</td>
</tr>
<tr>
<td>(2,7)</td>
<td>0.111</td>
<td>(2,3)</td>
<td>200.01</td>
<td>0.029</td>
<td>1.793</td>
<td>6.480</td>
</tr>
<tr>
<td>(3,4)</td>
<td>0.875</td>
<td>(3,4)</td>
<td>200.03</td>
<td>0.024</td>
<td>1.961</td>
<td>8.441</td>
</tr>
<tr>
<td>(3,7)</td>
<td>0.125</td>
<td>(3,4)</td>
<td>200.03</td>
<td>0.019</td>
<td>2.187</td>
<td>10.629</td>
</tr>
<tr>
<td>(4,5)</td>
<td>0.857</td>
<td>(4,5)</td>
<td>200.04</td>
<td>0.009</td>
<td>3.065</td>
<td>41.443</td>
</tr>
<tr>
<td>(4,7)</td>
<td>0.143</td>
<td>(4,5)</td>
<td>200.06</td>
<td>0.004</td>
<td>12.39</td>
<td>13.695</td>
</tr>
<tr>
<td>(5,6)</td>
<td>0.833</td>
<td>(5,6)</td>
<td>200.05</td>
<td>0.004</td>
<td>4.281</td>
<td>17.976</td>
</tr>
</tbody>
</table>
Path explored: 1 2 3 4 5 6 8 9 10 15
Table 2. Brief description for Path-1 is given below.

Put IWD on the top most node of CFG, i.e., Node-1, it can have two possible outgoing paths, either Node-2 or Node-7. As per formulae given in Step-3 of Algorithm 2, it calculates the probability for both nodes by using probability fitness function.

Probability (1, 2) = 0.9 & Probability (1, 7) = 0.1

Probability value shows the probability of selecting a particular path among all available paths from the current node. It chooses the path with the highest probability. Probability (1, 2) = 0.9, it chooses path (1, 2) and ignores path (1, 7). Probability of the path is affected by all the metrics of IWD. Velocity (IWD) moving from Node-1 to Node-2 is updated as per the equation shown in Step-5 of Algorithm 2, i.e., Velocity (IWD) = 200.01. In Step-6 of Algorithm 2, update time parameter for IWD. It is calculated as time(1,2;IWD) = HUD(2) / Velocity(IWD) = 90/200.01 = 0.449. Where HUD (2) = 10 (CC of CFG) × 9 (CC at Node-2) = 90. In Step-7 of Algorithm 2, change of soil is calculated, i.e., Δsoil(1,2) = 1.468 as per Step-7 of Algorithm 2. Update the Soil (IWD) = 1.468 loaded from path between Node-1 and Node-2 as per Step-8 of Algorithm 2. In Step-9 of Algorithm 2, soil of path between Node-1 and Node-2 is updated, i.e., soil(1,2) = soil(1,2) - Δsoil(1,2) = 10,000 - 1.468 = 9,998.532.

By using the same approach, Algorithm 2 iterates for the exploration of the remaining paths. It finds all possible paths which count to cyclomatic complexity (i.e., 10 for the given triangle problem) of CFG. The available paths in the given triangle problem are as follows.

Path 1: 1 2 3 4 5 6 8 9 10 15
Path 2: 1 2 3 4 5 6 8 12 13 15
Path 3: 1 2 3 4 5 6 8 9 11 15
Path 4: 1 2 3 4 5 6 7 15
Path 5: 1 2 3 4 5 7 15
Path 6: 1 2 3 4 7 15
Path 7: 1 2 3 7 15
Path 8: 1 2 7 15
Path 9: 1 7 15
Path 10: 1 2 3 4 5 6 8 12 14 15

From each of the paths obtained, test data is generated based on the condition available

Table 3. Experimental result of triangle problem

<table>
<thead>
<tr>
<th>Path No.</th>
<th>Condition</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; (i==j) &amp; (j==k)</td>
<td>i 4  j 4  k 4</td>
</tr>
<tr>
<td>2</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 6  j 7  k 7</td>
</tr>
<tr>
<td>3</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 6  j 6  k 1</td>
</tr>
<tr>
<td>4</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 5  j 2  k 2</td>
</tr>
<tr>
<td>5</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 3  j 8  k 5</td>
</tr>
<tr>
<td>6</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 3  j 6  k 9</td>
</tr>
<tr>
<td>7</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 9  j 0  k 5</td>
</tr>
<tr>
<td>8</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 0  j 7  k 3</td>
</tr>
<tr>
<td>9</td>
<td>!(i&lt;=0) &amp; !(j&lt;=0) &amp; !(k&lt;=0) &amp; !(i+j-k&lt;=0) &amp; !(i+k-j&lt;=0) &amp; !(j+k-i&lt;=0) &amp; !(i==j) &amp; (j==k)</td>
<td>i 6  j 4  k 5</td>
</tr>
</tbody>
</table>
on that path. Table 3 shows the result of proposed algorithm for generating test data from the obtained path which contains the attributes, namely, ‘path number,’ ‘condition,’ and ‘values of the variable.’ For particular path, test data is generated based on the conditions available on that path.

**Detailed Calculation on Path-1: 1 2 3 4 5 6 8 9 10 15, is given.**

Condition on any path can be extracted by parsing the input program using YACC parsing tool (Brown et al., 1995), only for each node present in the condition; append all extracted condition using AND operation (shown as &&). Conditions present in Path-1 are as shown.

\[
!(i<=0) \&\& !(j<=0) \&\& !(k<=0) \&\& !(i+j-k<=0) \&\& !(i+k-j<=0) \&\& !(j+k-i<=0) \&\& (i==j) \&\& (j==k)
\]

Algorithm 3 tries to fire all different combinations of variables i, j and k. Execute generated combination on the given condition until condition evaluates to true.

For example, randomly generated values are

- Variable: i, value: 1
- Variable: j, value: 8
- Variable: k, value: 5

Given combinations are not satisfying the above condition, so it tries to generate different combinations.

At some point it generates values which satisfies the above condition for the variable i, j and k like,

- Variable: i, value: 4
- Variable: j, value: 4
- Variable: k, value: 4

These values are given as output as the test data for Path-1: 1 2 3 4 5 6 8 9 10 15.

In the same way it generates test data for the rest of the paths explored during Algorithm 2 and assigns unique combination to each one. Table 3 shows the results for each path as per condition that has been encountered.

Similarly, the proposed strategy is applied to the min-max problem and results observed are also very impressive.

In the study-2, proposed IWD approach is being applied on another problem, determining maximum of three inputs, as listed in Program 2. This program takes three values of i, j, k as an input, and finds the maximum amongst the three values.

The cyclomatic complexity for Program-2 is 4 which can be calculated after constructing the CFG for the program. Table 4 shows the result of the proposed algorithm by listing...
conditions extracted for respective path and generated test.

Table 4 contains attributes, namely, ‘Path Number,’ ‘Traversed Path,’ ‘Condition,’ and values of the ‘Variable.’ For particular path, test data is generated based on the condition available on that path.

The above two studies show that, it discovers all paths available in the CFG with less number of iterations and also generates the test data for the obtained paths. Comparison of these approaches with other approaches is detailed in the next section.

ANALYSIS

IWD approach is used for exploring the number of paths available in the program. As already shown, the algorithm is effectively generating test data based on the obtained paths. This section gives the comparison of the proposed approach with other approaches.

Figure 4 shows the comparison of the IWD with artificial bee colony algorithm (Dahiya, Chhabra, & Kumar, 2010) for triangle problem (which finds type of triangle from three sides of the triangle). Comparison criteria have been taken as the number of iterations needed to generate test data for each path. In artificial bee colony algorithm, the number of iterations for triangle problem implemented by code having CC=7 is 6197, whereas in the proposed IWD implementation it needs only 518 iterations for the same problem implemented by code having CC = 10.

Figure 5 shows the comparison graph for the generation of character type test data in addition to numeric test data where the proposed approach overcomes the conceptual limitations for character test data generation. The proposed approach can generate numeric as well as character data but the other two approaches, i.e., automatic test data generation based on ACO (Kewen et al., 2009) and application of genetic algorithm and Tabu search in software

<table>
<thead>
<tr>
<th>Path No.</th>
<th>Path Traversed</th>
<th>Condition</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3 8</td>
<td>(i&gt;j) &amp;&amp; (j&gt;k)</td>
<td>4 2 1</td>
</tr>
<tr>
<td>2</td>
<td>1 5 6 8</td>
<td>!(i&gt;j) &amp;&amp; (j&gt;k)</td>
<td>1 2 0</td>
</tr>
<tr>
<td>3</td>
<td>1 2 4 8</td>
<td>(i&gt;j) &amp;&amp; !(j&gt;k)</td>
<td>3 0 0</td>
</tr>
<tr>
<td>4</td>
<td>1 5 7 8</td>
<td>!(i&gt;j) &amp;&amp; !(j&gt;k)</td>
<td>0 1 4</td>
</tr>
</tbody>
</table>

Table 4. Experimental result for maximum problem
Program 2. Maximum of three input

```c
int findMax(int i, int j, int k)
{
    int max;
    1. if(i>j) then
    2.     if(i>k) then
    3.         max = i;
    4.     else max=k;
    5. elseif(j>k)
    6.     max = j;
    7. else max = k;
    8. return(max);
}
```
testing (Rathore et al., 2011) are applicable only for numeric test data.

Figure 6 shows the comparison results of the number of iterations during path generation using IWD and random process. The number of iterations required during path exploration using random process is too high as compared to the iterations required using IWD. The random process is independent of the number of paths but as graph of IWD is linearly monotonically increasing it means it is directly dependent on the number of the paths. The advantage of this algorithm is that, it requires exactly the same number of iterations as that of available paths.

Figure 7 contains the graph that shows the number of iterations is closely related to the range of variable per execution, e.g., In triangle problem if range is set to be 0..5 then number of iterations required for the test data generation for each path is less than that with range 0..255. In addition to this, as depicted in the graph, number of iterations for generation of data increased drastically for worst case conditions like \((a==b)\) and \((b==c)\) has been given. So, probability of generating the same value for variable ‘b’ as well as ‘c’ again is \(1/(\text{max(Range)})^2\). In short, number of iterations can be reduced by choosing the range wisely for test data generation. The developed tool provides option to select the range of variables from user (developer/tester) because he/she is the only person who can take decision between trade-off of the selecting the wide range v/s number of iterations for generating test data.

Figure 8 shows the optimization techniques for generating values of the variables present in the path of predicate node. The technique is applied during test data generation. This saves unnecessary effort which would have been applied and wasted in randomly generating data for unnecessary variable in each path.

As shown in Figure 8, consider any program containing 200 variables and N number of paths. Randomly chosen strategy generates the values for each of the 200 variables during test data generation of every path, whereas the proposed approach generates test data for each path according to the variables that are present in the predicate node of the given path.
CONCLUSION

This paper presents strategies for automatic test-path exploration using IWD algorithm which is able to find optimal solutions. The efficiency and practicability of IWD is proved by the testing of several real problems, and the results were compared to that of random processes. From the results, it is proved that IWD is more efficient than random process, and it also avoids the problem of consuming a large number of iterations. The results of the experimental study support the claims that the proposed algorithm is superior to other related strategies.

This paper has provided solutions for the test data generation taking into consideration only the numeric and character types. In future, more works are needed to be carried out for extending the proposed solutions to the string data type.

REFERENCES


Praveen Ranjan Srivastava is working under the Software Engineering and Testing Research Group in the Computer Science and Information Systems Department at the Birla Institute of Technology and Science (BITS) Pilani at the Pilani Campus in Rajasthan, India. He is currently doing research in the area of software testing. His research areas are software testing, quality assurance, quality attributes ranking, testing effort, software release, test data generation, test effort estimation, agent oriented software testing, and metaheuristic approaches. He has published more than 80 research papers in various leading international journals and conferences in the area of software engineering and testing. He has been actively involved in reviewing various research papers that he has submitted in his field for different leading journals and various international and national level conferences. He has received various funds from different agencies like Microsoft, IBM, Google, DST, CSIR, etc.

Amitkumar Patel is pursuing Master of Engineering specializing in Software Systems from Computer Science and Information Systems Department at the Birla Institute of Technology and Science (BITS) Pilani, Pilani Campus, Rajasthan, India. He was working with one of the IT giant – Wipro Technologies before joining BITS. His areas of interests include research in field of software engineering & algorithms designs.

Kunal Patel is pursuing Master of Engineering specializing in Software Systems from Computer Science and Information Systems Department at the Birla Institute of Technology and Science (BITS) Pilani, Pilani Campus, Rajasthan, India. His areas of interests are software development, software testing and parallel computing. He is an open source enthusiast and has been actively contributing to the community.
Prateek Vijaywargiya is currently a first year Master’s (Masters of Engineering-Software System) student at Birla Institute of Technology & Science, Pilani and has just completed a Bachelors Degree in Computer Science and Engineering in the university of Rajiv Gandhi Technical University, Bhopal. He has most of his interest in the “Development Area” within the area of his specialization that is C/C++/Java. Apart from his regular curriculum, he was an active member of the institute for two years and achieved FIRST position in the Software Debugging. He has also achieved FIRST position in the TAEK-WON-DO at State and District Level.
APPENDIX

The application of the tool that has been developed for simulation of proposed solution has been described through following screen shots. Simulated results displayed in the output text area. Status messages shown in the respective section of the UI.

*Figure 9. The simple GUI for simulating the results of the proposed intelligent water drop algorithm*

*Figure 10. As the input CFG, all the independent paths have been extracted as explained in Algorithm 2 in above sections. Execution time is presented in the status.*
The output in the text area shows the generated test data according to each path and the conditions extracted in each path. In the status part, elapsed time is displayed along with the total number of iterations.