Code coverage using intelligent water drop (IWD)

Article *in* International Journal of Bio-Inspired Computation · January 2012
DOI: 10.1504/IJBIC.2012.051396

3 authors, including:

Dr. Praveen Ranjan Srivastava
Indian Institute of Management, Rohtak
118 PUBLICATIONS **877** CITATIONS

SEE PROFILE

Code coverage using intelligent water drop (IWD)

Komal Agarwal, Manish Goyal and Praveen Ranjan Srivastava*

Department of Computer Science and Information Systems,
Birla Institute of Technology and Science (BITS),
Pilani, Pilani Campus, Rajasthan-333031, India
Fax: 01596-244183
E-mail: komalagarwal19@gmail.com
E-mail: goyalmanish76@gmail.com
E-mail: praveenrsrivastava@gmail.com
*Corresponding author

Abstract: Software testing is one of the most challenging and arduous phase of software development life cycle (SDLC) which helps in determining the software quality. Code coverage is a widely used testing paradigm, which describes the degree to which the code has been tested. Aim of the current paper is to propose an optimised code coverage algorithm with the help of an emerging technique, i.e., intelligent water drop (IWD). This approach uses dynamic parameters for finding all the optimal paths using basic properties of natural water drops. It proposes how test cases can be considered as an IWD moving on the edges of the control dependency graph for finding the optimal paths. The algorithm guarantees complete code coverage by generating automated test sequences.

Keywords: code coverage; intelligent water drop; IWD; control dependence graph; CDG; test case.

Reference to this paper should be made as follows: Agarwal, K., Goyal, M. and Srivastava, P.R. (2012) 'Code coverage using intelligent water drop (IWD)', *Int. J. Bio-Inspired Computation*, Vol. 4, No. 6, pp.392–402.

Biographical notes: Komal Agarwal is a graduate student presently doing her ME (Software Systems) in the Computer Science and Information Systems Department at Birla Institute of Technology and Science, Pilani, India. Her areas of interest lie in software testing, soft computing, and databases.

Manish Goyal is a graduate student presently doing his ME in Software Systems at Birla Institute of Technology and Science, Pilani, India. His areas of interest lie in software engineering, testing, database and computer networks and soft computing.

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, India. He is currently doing research in the area of software testing and software engineering using metaheuristic techniques. His research areas are software testing, quality assurance, quality attributes ranking, testing effort, software release, test data generation, agent oriented software testing, and advanced soft computing techniques. He has published more than 90 research papers in various leading international journals and conferences in the area of software testing. He has been actively involved in reviewing various research papers submitted in his field to different leading journals and various international and national level conferences.

1 Introduction

Software engineering deals with the various phases of software development right from requirement analysis to its maintenance. Software testing being the most important aspect of software quality assurance, it is the most labour intensive and expensive task (Myers et al., 2004). Software testing accounts for about 50% of the total software lifecycle cost and time (Bertolino, 2007).

The major challenges in the field of software testing are its efficiency, thoroughness and resource management. Coverage-based testing which includes both code and requirement coverage is one way to improve software efficiency (Beizer, 1990). Code coverage is a technique of prioritising the entire source code for testing, which promises to achieve maximum testing coverage with the least cost. These days artificial intelligence (AI) techniques are changing the process of software testing with the use of

heuristics. Many heuristic algorithms such as genetic algorithm (GA) (Srivastava et al., 2008), Tabu search (Glover and Laguna, 1989) and ant colony optimisation (ACO) (Srivastava and Baby, 2010) have been applied in the field of software testing. These problems suffer from ineffective and incomplete code coverage problem, and a large number of test cases are required for complete code coverage.

Recently, a new problem solving metaheuristic algorithm 'intelligent water drop (IWD)' (Shah-Hosseini, 2007, 2009) has been introduced. This algorithm tries to overcome some of the weaknesses of the earlier implemented techniques. This paper proposes an algorithm, which uses 'IWD' optimisation technique to generate automatic test sequences, to provide a solution to the code coverage problem. IWD is a very effective population-based optimisation technique, which is based on the process that happens in river systems involving natural water drops. Every water drop finds an optimal path by traversing the search space of the given problem and then modifies the environment in which it is flowing. Thus, it uses a constructive approach to build the optimal solution for a given problem. Using the properties of natural water drops, artificial water drops are created. Various research has already been done using IWD (Shah-Hosseini, 2007, 2009) in problems like travelling salesman problem (TSP), multidimensional Knapsack problem (MKP), n-queen puzzle and robot path planning. Effective results obtained in the above mentioned problems, inspired us to work on the behaviour of IWD for finding the optimal solution to the desired problem.

IWD is a graph-based algorithm and the nature of any software can be represented in the form of control flow graph (CFG), hence IWD can be used to model any such problem. Using the basic model of IWD all the possible test sequences are generated which are used to calculate the percentage of code covered. All the possible paths which the test cases can traverse with their corresponding weights are computed using the proposed algorithm. Weight corresponds to the complexity of paths, i.e., path with higher weight is more error prone with more decision statements, loops, and functions. The proposed algorithm which is applied on control dependency graph (CDG) (Pargas et al., 1999) designed from the source code primarily focuses on software analysis for code coverage.

The paper is organised as follows. In Section 2, an overview of software testing is discussed. Sections 3 and 4 describe the IWD algorithm and the proposed methodology for test sequence generation. Section 5 discussed demonstration of the algorithm with an illustration, while Section 6 analysis of the algorithm and comparison with other metaheuristic techniques is shown. Finally, Section 7 presents the conclusions drawn.

2 Background work

Software analysis has always been helpful in determining the quality of any software. Correctness and completeness of software is essential for its quality assurance. One of the major problems associated with the field of software testing is to determine the optimal test sequences and test cases. As exhaustive testing is not feasible in most of the cases, so automated generation of test data needs to be done in minimum cost and time.

Several AI techniques (Srivastava et al., 2008; Glover and Laguna, 1989; Srivastava and Baby, 2010) provide a solution to the above mentioned software testing issue. The size of test data is of great concern, due to which some heuristics need to be applied in this field. A GA-based technique has been proposed to generate test data from UML state diagram, in which test data is generated before coding. As only one best solution is selected, it leads to poor performance in test data generation.

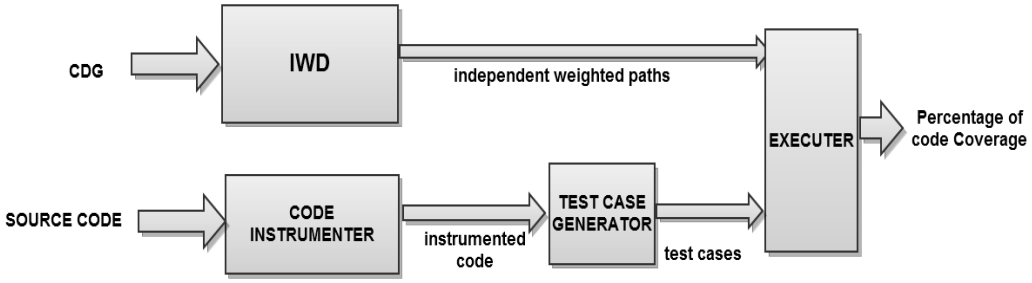
Various work (Srivastava et al., 2008; Lin and Yeh, 2000; Nirpal and Kale, 2011) has been done on test data generation using GA. This work focused on test data generation and corresponding software coverage. ACO (Srivastava and Baby, 2010; Li and Lam, 2005) is being recently used in software testing for generating test data. ACO proves to be an effective technique for finding good paths through graphs. ACO is also used for state-based testing (http://www.cs.helsinki.fi/u/paakki/software-testing-s05-Ch611.pdf), which considers only states, not the complete transactions.

One of the graphs-based algorithm that has been proposed in Mala and Mohan (2008) works on finding the shortest path between the starting node and its sub nodes in an execution state sequence graph (ESSG). This algorithm is not able to cover all transitions leading to weak level of coverage for software testing. Hence, there is a lot of scope for improvement. Traditional method of code prioritisation uses dominator analysis to assign code priorities for testing to achieve best coverage.

Dominator analysis (Agrawal, 1994) is a method to find related basic blocks in the code, of which if one is covered the remaining are guaranteed to be covered. One limitation with dominator analysis is that it considers only the node relationship within a class and method, while the dependencies among classes and methods are not taken into account. The other limitation is the guaranteed relationship among basic block nodes is sometimes not necessary, and its calculation algorithm is a bit complex.

GAs solves (Pargas et al., 1999; Dahal et al., 2007) the problem of exhaustive testing, but they suffer from several disadvantages. As they rely on Black box testing, i.e., state-based testing, this has a complex structure and slow execution speed. So in order to make the software testing cost effective there should be some technique, which gives desired test sequences.

Figure 1 System block diagram



Automatic test sequence generation and path coverage using metaheuristic approaches was proposed. Application of ACO for software coverage has proposed better results as compared to the GA algorithms. However, ACO techniques comes up with some disadvantages like constant pheromones deposition on the edges, irrelevance of the velocity of ants in further iterations, inability of the ants to change the amount of pheromones on the edges. All these major problems faced in ACO can be removed using IWD algorithm due to its dynamic behaviour. In contrast to ACO, IWD changes the amount of soil on the edges, which ensures its dynamic behaviour. In addition, IWD can both remove and add soil to an edge, whereas these changes could not be done in the ACO algorithm. In contrast to ACO where each ant deposits a constant amount of pheromone on the edges, the changes made in the soil of an edge are not constant in the IWD algorithm. Besides, while flowing down a path the IWDs carry some soil along with them from the path, this leads to a change in the velocity of the IWDs, which is constant in ACO. This shows the dynamic behaviour of IWD and ensures greater efficiency than various other techniques specified during the course of this section. The next section explains IWD.

3 Introduction to IWD and its behaviour

IWD (Shah-Hosseini, 2007, 2009) algorithm has been applied for the below mentioned problem which is idealised upon natural water drops in rivers. Natural water drops have two properties that are soil and velocity. Every water drop flows with some velocity and carries an amount of soil with itself. The velocity of the water drop helps in removing some soil from the riverbeds, and this soil is added to the soil of the water drop.

During transition, the water drops gains speed; due to which a water drop with more velocity can erode more soil. One more interesting feature of the water drop is that it chooses an easy path when there are many paths available at the same time. Here, easy path means the path with lesser soil. In this way constructively, the water drop chooses its optimal path. Using these remarkable properties IWDs are created. These drops possess two important properties of natural water drops that are:

1 the soil it carries, denoted by soil (IWD)

2 the velocity that it possesses, denoted by velocity (IWD).

Both these values can change when an IWD flows from one location to another in its environment (given problem statement). The IWDs' velocity is non-linearly proportional to the inverse of the soil on the path between the two locations. The IWDs' soil is increased by removing some soil from the path, and the amount of soil added is non-linearly proportional to the inverse of the time taken to move between the locations. This time is proportional to the velocity and inversely proportional to the distance between the two locations. In addition, the path choosing property is implemented by calculating a uniform random distribution function, such that the probability of choosing a path is inversely proportional to the soil of the available paths.

This paper briefs about IWDs, which is used to find optimised test sequence(s) in the design phase of the software development life cycle (SDLC). The following section explains how the properties of water drops are applied to the desired problem and the process of achieving the optimised test sequences.

4 Proposed methodology

The block diagram for the implementation and operation of the proposed algorithm is shown in Figure 1.

CFG (Mathur, 2007) represents the graphical representation of the programme, which is then converted into a control dependence graph (CDG). The nodes and edges in a CDG represent statements and control dependencies among them respectively. The nodes which have the minimum height, and are not traversed yet are considered as the target node. In case there is more than one node with the minimum height then the target is randomly selected. Since CDG is in tree format loops can easily be handled, and the problem of getting stuck in local optima is avoided. CDG also simplifies the coverage problem, and it directly describes the dominance of one node over the other node. That is why, CDG was preferred over CFG.

CDG is supplied manually to the system, and all possible independent paths are found out by the application of the proposed IWD algorithm. The source code under test is instrumented in order to get the coverage data. The instrumented code is generated by the code instrumenter by adding few lines to the source code (for completion of a programme). This generated instrumented code is then supplied to the test case generator in order to find the test cases for the respective source code. Now, the percentage of

code coverage is calculated with the help of the executer on the basis of paths covered by the test cases.

Each block of the block diagram in Figure 1 is explained below:

- *Code instrumenter* (http://en.wikipedia.org/wiki/Instrumentation_%28computer_programming%29)

  Instrumentation of the source code is done by inserting some statements called probes at the beginning of every block of code without altering the programme logic. This is done in order to monitor the paths followed by the generated test cases. The source code is given as input, and the instrumented code is the output generated.

- *Test case generation*

  In this module, the test cases for the required source code are generated randomly using simple machine learning algorithm called Gibbs Sampling (Jandaghi, 1994). The number of inputs, its range and the type should be specified by the tester himself. These generated test cases are further fed to the executer.

- *Executer*

  The executer has the generated test cases, instrumented code and the independent paths with their weights and priority as its inputs. The paths executed by the test cases are recorded and are compared with the independent paths generated from the application of IWD on the CDG. The paths which are left uncovered are recognised as untraceable paths, and are expected to have some errors. Hence the code coverage percentage can be calculated.

  In the next portion, the working of IWD on the generated CDG to get the independent paths is explained:

- *Working of IWDs on CDG*

  Initially the important factors affecting the working of the algorithm are explained followed by a stepwise explanation of the algorithm.

  1 *Fitness function*

  A fitness function (Chen et al., 2009) has been designed for the calculation of soil on all the links of the graph. The basic idea behind using the fitness function is to give importance to more deep and complex paths. A well-constructed fitness function may increase the chances of finding a solution and reaching higher coverage, which is always desirable.

  For the proposed algorithm, the Fitness function used is:

$$\text{soil}(i,j) = a * \text{subgraph}(j) + b * \text{condition}(j),$$

$$\text{soil}(i,j) = 1\{\text{if subgraph}(j) \;\&\&\; \text{condition}(j) == 0\}$$

  where soil (i,j) is the soil on the edge between the two nodes i and j.

  Condition (j) of a node j can be either 0 or 1, depending on whether the particular node has a decision element or not. If it has a decision statement then it is assigned a value of 1 else 0.

  Subgraph (j) is the subgraph value, i.e., number of nodes below that particular node (j) in the given graph. The IWD starts from the initial node of the graph and after that the probability of each path is calculated using the above mentioned fitness function. The path with the highest probability is considered as the desired path for the water drop. The configuration parameter 'a' and 'b' is assigned a constant value of 2 and 1 respectively. The reason behind assigning these values is that, more weight should be assigned to nodes with a larger subgraph value rather than nodes with smaller subgraph value. Subgraph value has been given greater importance in the designing of the fitness function, because by traversing the part of the code with greater depth the probability of achieving complete code coverage is high.

  Value of soil(i,j) is one, whenever the value of subgraph(i,j) and condition(i,j) is zero. The reason behind this is the value soil(i,j) is later used for calculating the probability of corresponding node, and probability can never be zero in our context.

  This fitness function was so chosen, because during code coverage there are chances of errors in either decision nodes, or more dense and complex paths. In order to assign weight to links based on their complexity, this fitness function is chosen.

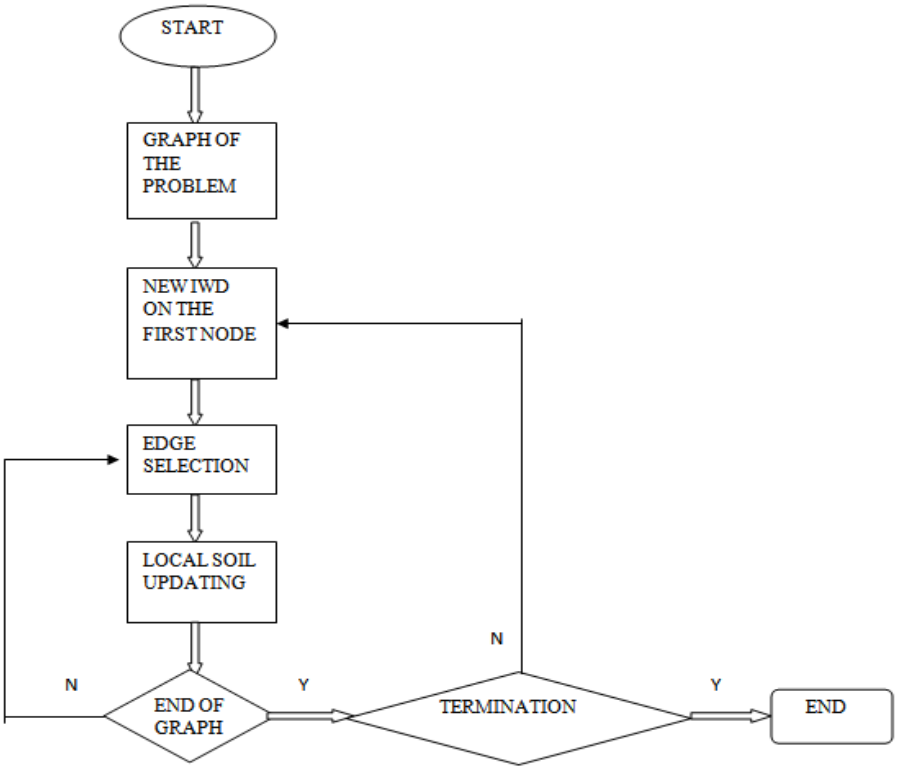
  2 *List of visited paths*

  A list of visited paths known as 'Vc' is maintained to record the paths traversed by the IWD. In order to keep track of the nodes visited, a check is performed on the node, and if all its children are visited then it is deleted. This operation is performed recursively. This is done in order to avoid the IWD from traversing the path which has been already traversed.

  3 *Constant values*

  Various configuration parameters (a_s, b_s, c_s), (a_v, b_v, c_v), ρ (Shah-Hosseini, 2007, 2009) are used in the calculation and updating of soil and velocity respectively during implementation of the IWD algorithm. Figure 2 shows the various steps involved in applying the IWD algorithm on the CDG. The relevance of the different parameters used in algorithm has also been depicted. The stepwise implementation of the algorithm follows the sequence as shown in Figure 2.

Figure 2 Flow chart for the working of IWD



Algorithm 1

Stepwise implementation of the algorithm:

Step 1: Initialisation of parameters

All the parameters which are used are:

- Probability (i, j) is the probability of the IWD to move from node i to node j.
- Time (i, j) is the time taken by the IWD to move from node i to node j.
- ΔSoil (i, j, iwd) is the soil eroded by the IWD on the link between node i and node j.
- Vel (i, j, iwd) is the velocity of the IWD when it moves from node i to node j.

a Velocity updating parameters are a_v, b_v, c_v. Here, for the given problem values are taken as: $a_v = 1$, $b_v = 0.1$, $c_v = 1$.

These are user defined configuration parameters whose values are decided as per the requirement of the problem, i.e., suitable for calculation of updation of parameters (soil and velocity).

b Soil updating parameters are a_s, b_s, c_s. Here, for the given problem values are taken as: $a_s = 1$, $b_s = 0.1$, $c_s = 1$.

c The local soil updating parameter is ρ. Here ρ = 0.1 is chosen for better experimental results. Value of ρ can range from 0 to 1, depending upon the amount of soil on the link (nodes).

d The initial soil on each edge of the graph is computed using the fitness function and the configuration parameters

e The initial velocity of each IWD is set to 100.

f Every IWD has a visited path list Vc (IWD) which is initially empty, i.e., Vc (IWD) = {}.

Step 2: Updation of parameters values

2.1 While (CDG is not empty)

2.2 Calculation of probability:

For the IWD receding at node i, the next node j (which is not in visited node list Vc(IWD) is chosen on the basis of probability $p_i^{IWD}(j)$. The probability for IWD to move from the current node i to node j is calculated using the given formula:

Probability (i, j) = Soil (i, j) / Σ k ≠ vc Soil (i, k)

where Soil (i, j) is the soil between the two nodes, Σ k ≠ vc Soil (i, k) denotes the summation of the soils of all the paths which can be traversed from the current node i. In the original algorithm, the IWD moves to the location where the soil is less, but in our case we traverse the IWD to the path with more soil. Since, we need to test the more complex parts in order to catch errors earlier.

2.3 Calculation of time taken:

After calculating the probability, IWD chooses the next node which it has to move on. After selecting the respective node, firstly the time taken to move from node i to node j is calculated as:

Time (i.j) = (subgraph (i)–subgraph (j)) / vel (iwd)

where subgraph(i) – sugraph(j) correspond to the distance between the nodes and vel(iwd) is the original velocity of the IWD.

2.4 Calculation of Soil of the IWD:

Then the soil carried by IWD, and its velocity (i, j) is computed. Using the time we calculated, the soil carried by the IWD as:

$\Delta Soil(i, j, iwd) = a_s / (b_s + c_s * time(i, j))$

where a_s, b_s, c_s are the positive parameters as specified earlier

2.5 Calculation of Velocity of the IWD:

The velocity of the IWD after it has moved from node i to j is calculated as:

$vel(i, j, iwd) = vel(iwd) + (a_v + (b_v + c_v\ soil(i, j)))$

where soil (i, j) is the soil on the path before the IWD traversed the required path. a_v,b_v, c_v are the positive parameters as specified earlier and vel(IWD) is the previous velocity of the IWD.

2.6 Soil updation of the link:

Since, the IWD carries some amount of soil with it; the soil on the link is reduced. Therefore, the soil on the link is updated as:

$Soil(i, j) = (1 - \rho)\ soil(i, j) - \rho * \Delta soil(i, j, iwd)$,

where ρ is a positive parameter, Δsoil (i, j, iwd) is the soil carried by the IWD while moving from the node i and j.

2.7 Now, the current node is updated and the same procedure from Step 2.2 is repeated until it does not reach leaf node, or one complete path is not reached. After it reaches the leaf node then that node is deleted, and its parent is also checked. If all its leaves have been deleted, then it is also deleted.

2.8 After completing one possible path by the water drop, the visited path list is updated and the IWD again starts with a new iteration where all parameters are initialised again from Step 1 onwards. If the CDG becomes empty, that means all the paths have been traversed, and we have to exit the loop else all the steps from 2.1 are executed.

In this way, all the possible paths are found out with their corresponding weights and the complexity of the test sequence is directly proportional to its weight. The path with highest weight should be tested first. The next section shows the demonstration of the above mentioned algorithm on an illustrated problem.

5 Case study

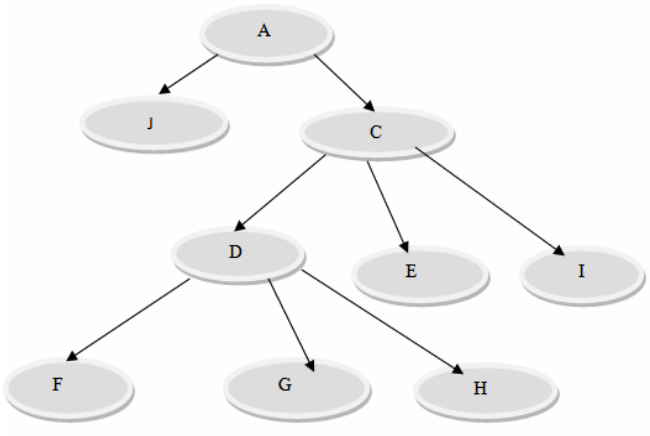
The proposed approach using IWD for code coverage has been tested on various real time problems [for example: greatest common divisor (GCD) problem, sorting problems like bubble sort, merge sort and many others]. Here, the demonstration of the algorithm will be shown using one of the data structure problems known as binary search problem (Cormen et al., 2001). It is one of the classical examples used in the field of software testing. The code for the binary search problem is shown in Figure 3, which is used for the implementation of the proposed IWD algorithm.

For the source code in Figure 3, the CDG (Pargas et al., 1999) is constructed which is shown in Figure 4.

Figure 3 Source code of binary search problem

```
int search(int key , int elemarray[] )
{
int bottom =0, top =8, mid , result=-1;        } Node A
while(bottom<=top)
{
mid=(top+bottom)/2;                            } Node C
if(elemarray[mid]==key)
{
result=mid;cout<<"4";                          } Node E
return result ;
}
else
{                                              } Node D
if(elemarray[mid]<key)
bottom=mid+1;                                  } Node F
else
top=mid-1;                                     } Node G
}                                              } Node H
}                                              } Node I
return result;
}                                              } Node J
```

Figure 4 CDG of the given problem of BST



The application of the IWD algorithm on the CDG of the above given binary search problem has been specified graphically in Figures 5 to 7.

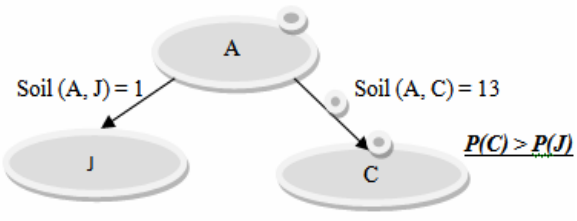
Subgraph of all the nodes is calculated first. Subgraph value of a node is the no of nodes below that particular node. Node A has all the nodes, i.e., J, C, D, E, I, F, G, H connected to it. This is done using the depth first traversal algorithm (Cormen et al., 2001). Hence the value of the subgraph of all nodes is:

- A = 8
- C = 6
- D = 3
- E, I, G, H, and J have a subgraph value of 0.

Now, the decision condition for the node is checked. A, C and D are decision nodes so they get 1 and rest of them get a value of 0. The various iterations of the IWD are shown below.

Iteration 1: The IWD starts from the starting node A as shown in Figure 5. As the water-drop, have two paths to travel, i.e., either node J or node C. So, in order to find out which node to be traversed next, the probability of two nodes is computed using the respective link soil and on the basis of the next node for the 1st path is selected.

Figure 5 Path selection by IWD during first iteration



Note: Visited path (V_c): A → C

The calculation of the various parameters is as shown:

$$\text{Soil}(A, C) = 2*6+1*1 = 13$$

$$\text{Soil}(A, J) = 2*0+1*0 = 1$$

$$(\text{as: subgraph(j) \&\& condition(j)} == 0)$$

$$P(A, J) = \text{Soil}(A, J) / \{\text{Soil}(A, J) + \text{Soil}(A, C)\} = 1/14$$

$$P(A, C) = \text{Soil}(A, C) / \{\text{Soil}(A, J) + \text{Soil}(A, C)\} = 13/14$$

The node with the higher probability is chosen first. In this case probability of node C is higher than that of node J so node C is selected. The probability is calculated using the Step 2.2 of the algorithm mentioned above. The time taken by the IWD to move from A–C is calculated using Step 2.3 of the algorithm mentioned above:

$$\text{Time}(A, C) = \max\{(8-6), 1\} / \text{velocity}(V_{iwd}) = 2/100 = 0.02 \text{ sec}$$

The soil carried by the IWD and its velocity is computed which is then subtracted from the soil of the link A-C to compute the remaining soil on the link using Steps 2.3 and 2.4 and 2.5:

$$\Delta\text{Soil}(A, C) = 1/(1+1*.02) = 0.98$$

$$\Delta\text{Velocity}(A, C) = 100+1/1+1*13 = 100.07$$

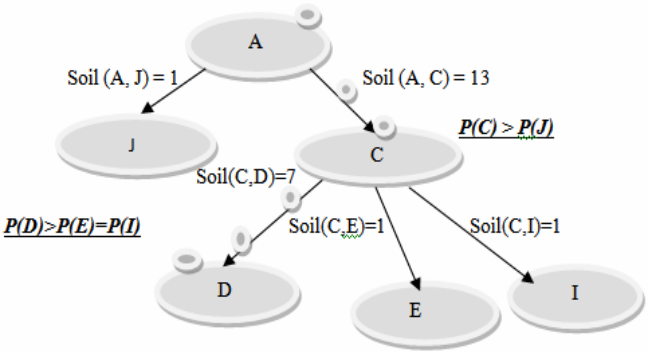
$$\text{UpdatedSoil}(A, C) = 0.9*13-0.1*0.98 = 11.6$$

$$\text{Soil (iwd)} = 0.98$$

So the path will be : A – C.

Now, after reaching node C the water-drop has three available nodes for selecting the next node, which is shown in Figure 6.

Figure 6 Path selection by IWD



Note: Visited path (V_c): A → C → D

Again, the probability of all the available nodes is calculated and on the basis of that the next node is chosen.

$$\text{Soil}(C, D) = 2*3 + 1*1 = 7$$

$$\text{Soil}(C, E) = 1$$

$$\text{Soil}(C, F) = 1$$

$$P(C, D) = \text{Soil}(C, D) / \{\text{Soil}(C, D) + \text{Soil}(C, E) + \text{Soil}(C, F)\} = 7/9 = 0.77$$

$$P(C, E) = \text{Soil}(C, E) / \{\text{Soil}(C, D) + \text{Soil}(C, E) + \text{Soil}(C, F)\} = 1/9 = 0.11$$

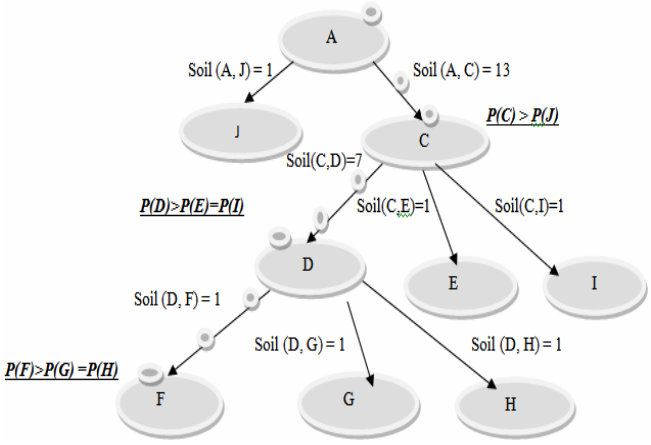
As the probability of node D is greater among all so the water drop will flow to node D.

$$\text{Time (C, D)} = (6-3)/100.07 = 0.03$$
$$\Delta\text{Soil (C, D)} = 1/(1 + 1*.03) = 0.97$$
$$\Delta\text{Velocity (C, D)} = 100.07 + 1/1+1 *7 = 100.195$$
$$\text{UpdatedSoil (C, D)} = 0.9*7 - 0.1*0.97 = 6.2$$
$$\text{Soil (iwd)} = 0.98 + 0.97 = 1.95$$

Sothe path will be : A – C – D.

Similarly, after reaching node D the water-drop has to decide the next node to be traversed as in Figure 7.

Figure 7 First final path followed by IWD during 1st iteration



The calculations for the various parameters are shown below:

$$\text{Soil (D, F)} = 1$$
$$\text{Soil (D, G)} = 1$$
$$\text{Soil (D, H)} = 1$$
$$\text{P (D, F)} = \text{Soil (D, F)} / \{\text{Soil (D, F)} + \text{Soil (D, G)} + \text{Soil (D, H)}\} = 1/3 = 0.33$$
$$\text{P (D, G)} = \text{Soil (D, G)} / \{\text{Soil (D, F)} + \text{Soil (D, G)} + \text{Soil (D, H)}\} = 1/3 = 0.33$$
$$\text{P (D, H)} = \text{Soil (D, H)} / \{\text{Soil (D, F)} + \text{Soil (D, G)} + \text{Soil (D, H)}\} = 1/3 = 0.33$$

Now, as the probability of all the three available nodes is same. A random selection is to be done among these three nodes. So, node F is randomly chosen as the next node.

$$\text{Time (D, F)} = (3-0)/100.195 = 0.03$$
$$\Delta\text{Soil (D, F)} = 1/(1 + 1*.05) = 0.97$$
$$\Delta\text{Velocity (D, F)} = 100.195 + 1/1+1 *1 = 100.695$$
$$\text{UpdatedSoil (D, F)} = 0.9*1 - 0.1*0.97 = 0.803$$
$$\text{Soil (iwd)} = 1.95 + 0.97 = 2.92$$

the path will be : A – C – D – F.

So, the first path that is traversed by IWD is:

Path1 : A – C – D – F

Now, the calculations for the second iteration for the IWD are shown below. The water drop starts again from the root node of the CDG, i.e., node A.

Iteration 2:

$$\text{P (A, C)} = 11.6/12.6 = 0.92$$
$$\text{P (A, J)} = 1/12.6 = 0.08$$

Since probability of node C is more we traverse to node C.

$$\text{Time (A, C)} = 8-6/100 = 0.02$$
$$\Delta\text{Soil (iwd)} = 1/(1+1*0.02) = 0.98$$
$$\text{Soil (A, C)} = 0.9*11.6-0.1*0.98 = 10.34$$
$$\text{Soil (iwd)} = 0.98+2.92 = 3.9$$
$$\text{Vel (iwd)} = 100+1/(1+1*11.6) = 100.08$$

It goes from A – C

$$\text{P(C, D)} = 6.2/8.2 = 0.75$$
$$\text{P(C, E)} = \text{P(C, I)} = 1/8.2 = .12$$

Since node D has greater probability, we select node D.

$$\text{Time (C, D)} = (6-3)/100.08 = 0.03$$
$$\Delta\text{Soil} = 1/(1+1*0.03) = 0.97$$
$$\text{Soil (C, D)} = 0.9*6.2-0.1*0.97 = 5.48$$
$$\text{Vel (iwd)} = 100.08 +1/(1+1*6.2) = 100.22$$

It moves from node C to Node D.

So, path will be A – C – D

$$\text{P (D, G)} = 1/2 = 0.5$$
$$\text{P (D, E)} = 1/2 = 0.5$$

Since, the probabilities of both the nodes are equal, so we select the node randomly. Here we select node G.

$$\text{Time} = 3-0/100.22 = 0.03$$
$$\Delta\text{Soil} = 1/(1+1*0.03) = 0.97$$
$$\text{Soil(D, G)} = 0.9*1-0.1*0.97 = 0.803$$
$$\text{Vel(iwd)} = 1/(1+1*1) + 100.22 = 100.72$$

It moves from node D to node G.

So,the path traversed is A – C – D – G.

Applying the algorithm in a similar fashion all the available paths are obtained by starting from node A. The path with maximum weight indicates the most complex path.

- Path 1: A → C → D → F
  (Total weight of path is: 21)
  (Total weight of path is: 21 [13+7+1])
- Path 2: A → C → D → G
  (Total weight of path is: 18.60)
- Path 3: A → C → D → H
  (Total weight of path is: 16.82)

- Path 4: A → C → E
  (Total weight of path is: 10.20)
- Path 5: A → C → I
  (Total weight of path is: 9.18)
- Path 6: A → J
  (Total weight of path is: 0.81).

This data can be shown in Table 1.

Table 1 Path strength table

Path no.	*Path*	*Weight*	*Priority*
1	A-C-D-F	21	1
2	A-C-D-G	18.6	2
3	A-C-D-H	16.82	3
4	A-C-E	10.2	4
5	A-C-I	9.18	5
6	A-J	0.81	6

The proposed algorithm uses IWD concept to generate all the possible independent paths with their priority, which is depicted in the path strength in Table 1. The weight of each path is calculated by adding the soil on all the links of the path. It shows that the path with highest weight has the highest priority. This data can help the tester to decide which paths to test first. Since, the weight is assigned directly based on the complexity, so this information can be useful for the tester.

6 Comparison study and analysis

6.1 Comparison with GA and ACO

In this section, the proposed IWD algorithm is compared with other AI techniques such as GAs and ACO (Srivastava and Baby, 2010; Dahal et al., 2007) in terms of percentage of coverage and redundancy in the paths traversed.

The work done by Dahal et al. (2007) uses the GA in the field of test data generation, using UML state diagrams. The above UML model suffers from a common problem of inappropriate and lesser coverage. Among the various state machine diagrams they used, one of the UML diagram, i.e., student enrollment state transition diagram (Dahal et al., 2007) is shown in Figure 8. It is used for the purpose of comparing the coverage achieved by GA and also by the proposed IWD algorithm.

Experiment done by Dahal et al. (2007), they select only the best solution which covers most of transitions. It is observed that their system works very well with the system, which does not contain the final state. In addition, they select only one best solution for the problem. So, all transitions with a final state cannot be reached in the diagrams (i.e., an infeasible transition). In addition, the looping problem is of another concern, i.e., transition 't4' to state 'full' cannot be fired unless the number of students who enrol this course is equal to the maximum number of seats in the class. Whereas, the proposed IWD algorithm works appropriately on systems with final state, and gives maximum coverage. The looping problem is also resolved, because of using CDG instead of CFG.

After analysis of the two algorithms on the above state machine diagram, the percentage of coverage is calculated which is shown below in Figure 9. The results clearly states that by using GA algorithm only around 34% of coverage is possible (Dahal et al., 2007), while by using the proposed IWD algorithm 100% coverage is achieved.

The 100% coverage achieved using the proposed IWD algorithm is very effective in the field of software testing. Thus, the proposed IWD algorithm gives highly impressive software coverage as compared to GA.

Figure 8 The enrolment state-machine diagram

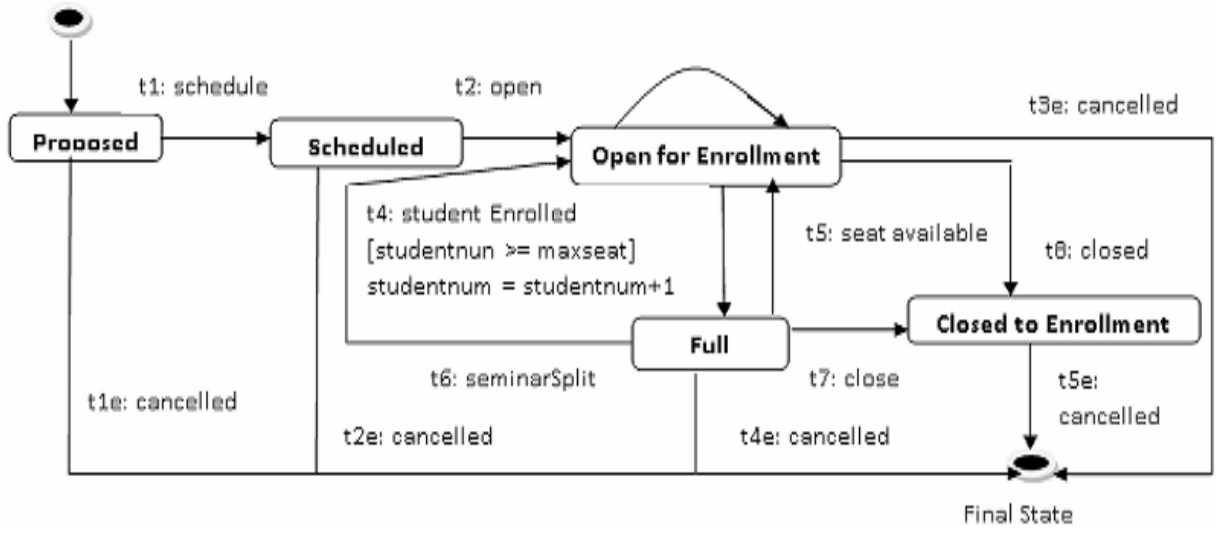
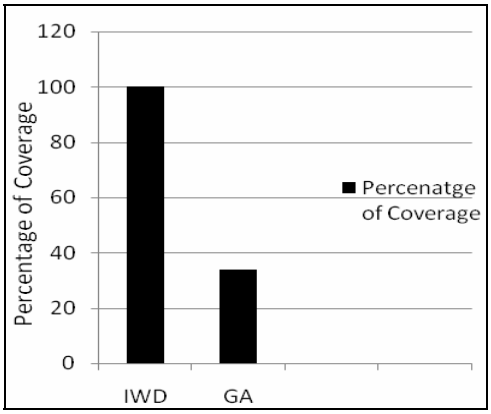


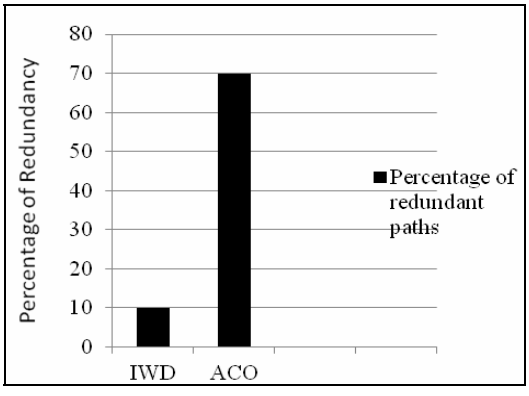
Figure 9 Comparison of IWD and GA techniques for the percentage for coverage achieved



The proposed IWD algorithm is also compared with the ACO technique in terms of the percentage of redundant paths, using the Binary search problem (Cormen et al., 2001). Srivastava et al. (2010) has worked on Binary search problem for finding out all the possible paths using the ACO technique. As per their analysis, all possible paths are computed, but the amount of redundancy is high. For example, the path A–C–D–F is redundantly computed by ACO in Srivastava et al. (2010). This problem of redundancy is eliminated in the proposed algorithm. The proposed IWD algorithm uses an array of visited nodes V_C(IWD), which keeps on updating every time a path is calculated. Using such array helps in eliminating the redundancy during the computation of the possible paths for any problem. A comparison is made in order to analyse the output of the ACO technique and the proposed algorithm.

As seen from Figure 10, the number of redundant paths for ACO technique (Srivastava et al., 2010) is far more than the proposed IWD algorithm. Thus, the proposed IWD algorithm eliminates the problem of redundant paths.

Figure 10 Comparison of IWD and ACO techniques for the percentage of redundancy in paths obtained



On the basis of the above analysis, it can be inferred that the proposed IWD algorithm gives very high software coverage with minimal redundancy.

7 Conclusions

The major problem associated with software analysis is the generation of automated test data for efficient software coverage. There are also various issues regarding the instrumentation of the input programme. The instrumented source code is a necessary step towards test case generation.

This paper presents a strategy for generation of automated test sequences using the IWD approach. The output of the algorithm consists of all available independent paths, which in turn are used for finding the percentage code coverage. Higher the percentage of coverage better is the algorithm, i.e., the number of test data required for traversing the all available paths should be as minimal as possible.

The proposed IWD algorithm provides the most optimum results, as compared with other approaches like ACO, GAs, and other. However, further research work needs to be done in optimising the test case generation. Also, the random test data generation technique used in this paper can be replaced by some other metaheuristic approach.

Acknowledgements

The authors would like to thank the anonymous referees and editors for their valuable suggestions, which helped us in improving the quality of this paper.

References

Agrawal, H. (1994) 'Dominators, super blocks, and program coverage', *Proceedings of the 21st Symposium on Principles of Programming Languages*, ACM, pp.25–34, Portland.

Beizer, B. (1990) *Software Testing Techniques*, 2nd ed., International Thomson Press, USA.

Bertolino, A. (2007) 'Software testing research achievements, challenges, dreams', *Proceedings in 'Future of Software Engineering' (FOSE)*, IEEE Computer Society, Washington DC, pp.2–5, USA.

Chen, Y. et al. (2009) 'Comparison of two fitness functions for GA-based path-oriented test data generation', *Fifth International Conference on Natural Computation*, IEEE Computer Society, pp.2–4, Tianjian, China.

Cormen, T.H. et al. (2001) *Introduction to Algorithms*, 2nd ed., MIT Press, USA.

Dahal, K. et al. (2007) 'Test data generation from uml state machine diagrams using gas', *International Conference on Software Engineering Advances (ICSEA)*, IEEE Computer Society, p.47, France.

Glover, F. and Laguna, M. (1989) 'Tabu Search, Part I', *ORSA Journal on Computing*, Vol. 1, No. 3, pp.190–206.

Instrumentation (Computer Programming), available at http://en.wikipedia.org/wiki/Instrumentation_%28computer_programming%29.

Jandaghi, G.H. (1994) 'Monte Carlo estimation of the pedigree likelihood statistics using Gibbs Sampler', Ph.D. thesis, Department of Preventive Medicine and Biostatistics, University of Toronto, Canada.

Li, H. and Lam, C.P. (2005) 'An ant colony optimization approach to test sequence generation for state based software testing', *Proceedings of the Fifth International Conference on Quality Software (QSIC)*, IEEE Computer Society, pp.255–264, Melbourne.

Lin, J-C. and Yeh, P-L. (2000) 'Using genetic algorithms for test case generation in path testing', *Proceedings in 9th Asian Theological Seminary (ATS)*, IEEE Computer Society, pp.241–246, Taipei.

Mala, D.J. and Mohan, D.R. (2008) 'Intelligent test sequence optimization framework using multi-agents', *Journal of Computers*, Vol. 3, No. 6, pp.5–10.

Mathur, A.P. (2007)'*Foundation of Software Testing*, 1st ed., Pearson Education, USA.

Myers, G.J. et al. (2004) *The Art of Software Testing*, 2nd ed., John Wiley & Sons Inc, USA.

Nirpal, P.B. and Kale, K.V. (2011) 'Using genetic algorithm for automated efficient software test case generation for path testing', *International Journal of Advanced Networking and Applications*, Vol. 2, No. 6, pp.911–915.

Pargas, R.P. et al. (1999) 'Test-data generation using genetic algorithms', *Journal of Software Testing, Verification and Reliability*, Vol. 9, No. 4, pp.263–282.

Shah-Hosseini, H. (2007) 'Problem solving by intelligent water drops', *Proceedings in IEEE Congress on Evolutionary Computation*, IEEE Computer Society, pp.3226–3231, Singapore.

Shah-Hosseni, H. (2009) 'Optimization with the nature-inspired intelligent water drops algorithm', *International Journal of Bio-Inspired Computation*, Vol. 1, Nos. 1/2, pp.105–125.

Srivastava, P.R. and Baby, K. (2010) 'Automated software testing using metaheuristic technique based on an ant colony optimization', *Proceedings of International Symposium on Electronic System Design (ISED)*, IEEE Computer Society, pp.1–7, Bhubaneswar, India.

Srivastava, P.R. et al. (2008) 'Generation of test data using meta heuristic approach', *TENCON IEEE Region 10 Conference*, IEEE, pp.1–6, Hyderabad.

Srivastava, P.R., Baby, K. and Rasguhurama, G. (2010) 'An approach of optimal path generation using ant colony optimization', *Proceedings of IEEE TENCON 2009*, IEEE Computer Society, pp.1–7, Singapore.

State-based Testing, available at http://www.cs.helsinki.fi/u/paakki/software-testing-s05-Ch611.pdf.