Regular Paper

Optimal test sequence generation using firefly algorithm

Praveen Ranjan Srivatsava a,*, B. Mallikarjun a, Xin-She Yang b,c

a Department of Computer Science and Information Systems, BITS Pilani, India
b Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, UK
b,c School of Science and Technology, Middlesex University, London NW4 4BT, UK

A R T I C L E   I N F O

Article history:
Received 21 December 2011
Received in revised form 5 August 2012
Accepted 15 August 2012
Available online 25 September 2012

Keywords:
Firefly algorithm
State transition diagram
Objective Function
Cyclometric complexity
Control flow graph

A B S T R A C T

Software testing is an important but complex part of software development life cycle. The optimization of the software testing process is a major challenge, and the generation of the independent test paths remains unsatisfactory. In this paper, we present an approach based on metaheuristic firefly algorithm to generate optimal test paths. In order to optimize the test case paths, we use a modified firefly algorithm by defining appropriate objective function and introducing guidance matrix in traversing the graph. Our simulations and comparison show that the test paths generated are critical and optimal paths.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

In a typical software development life cycle [1], the software testing [2] phase is the most important step to ensure software quality. With the increasing demand for highly reliable, scalable, robust software packages, it is estimated that software testing accounts for almost 50% for the total effort in software engineering [3]. Two of the most commonly used software testing methodologies are Black Box testing and White Box testing [2]. White box testing, also known as glass box or structural testing, consists of the key control flow paths. The main aim in such white box testing is to test the internal structure of the software under consideration thoroughly, while the black box testing focuses only on what the software can do, that is, to test the outputs for a given set of inputs without knowing the internal structure of the software. In principle, testing efficiency can be improved by using a coverage-based approach. The increase in test coverage also increases software reliability [3]. To quantify the amount of testing of the software, coverage-based testing can be used [4].

The task to generate control flow paths is one of the most challenging tasks and in fact generating test cases that can cover the whole software structure is a tough optimization problem. To date, various methods such as constraint-based heuristics, symmetric matrix algorithm [5,6] have been investigated for the generation of control flow paths, but none of those offers satisfactory solutions for generating critical paths and path prioritization [1]. The generation of coverage in structural testing has led to various methods such as path coverage, data flow coverage, and branch coverage [2], but their solutions are still far from optimal. For example, the exhaustive coverage method generates the path coverage, and, due to its exhaustive nature, the constraints on time and cost in a software development project make it impractical to implement the exhaustive test cases. So it is highly needed to develop new methods to generate optimal test sequence or test paths, which can significantly reduce the cost, time and effort of software testing.

The three widely used approaches for structural testing are control flow based testing, data flow based testing and mutation testing [1,2]. Control flow based testing depends upon the control paths of the software under testing. The total number of paths even for testing a small program may be very large, resulting in a very large number of test cases, which increase the testing costs significantly; hence, effective path selection becomes crucially important [7]. The preliminary automation of software testing process using unified modeling language (UML) was already proposed in [31], but complete software coverage and path generation still remain an open research problem. Up to now, automating test data generation has been done using various algorithms and methods such as Ant Colony Optimization (ACO) algorithm [8–11], Genetic Algorithm (GA) [12,13] and Search-based test data generation [14]. All these methods try to generate test data in an automated manner to facilitate the task of software testing. These approaches often use swarm intelligence [14,15]. Recently for path generation optimization, a method based on Cuckoo Search (CS) has been used [16], and good results
were achieved. But node redundancy was still an issue. In another approach, optimal software test data generated by artificial bee colony (ABC) [17] seemed to work well for programs of small sizes. However, as the size of software increases, finding paths and test data becomes more difficult because this ABC method may be trapped in local search space and the number of iterations is quite high. On the other hand, the approach for test data generation using particle swarm optimization (PSO) did not have much discussion for path generation and coverage [18]. Test path generation using ant colony optimization (ACO) was attempted in [10], but there was an issue of transition repetition in the sequences generated by that method.

In order to generate more complete software coverage, we have to find an optimal test path, which is still a challenging task. In this paper, we try to explore the possibility of generating optimal and minimal test cases with full coverage by using a recently developed Firefly Algorithm (FA) [19]. FA [19] was developed by Xin-She Yang, inspired by the flashing behavior of fireflies. Mechanisms of firefly communication via bioluminescent flashes and their synchronization have been imitated effectively in various techniques of wireless networks design [20], dynamic market pricing [21], mobile robotics [22], economic dispatch problem [32], and structural optimization problems [33].

In this paper, we use FA to obtain the optimal set of test sequence generation using state transition diagram (STD) [23] and control flow graphs (CFG).

In software engineering, the so-called state transition diagram can be generated from the requirements or specifications of the software to be developed. The state transition diagram (STD) [24] indicates how the system behaves because of external events. To accomplish this, the STD represents the various modes of behaviors (called states) of the system, and the manner in which transitions are made from state to state. We will show that FA can generate an optimal set of paths from the state transition diagram.

The approaches by Srivastava and Baby [25] and Lam and Li [9] used ant colony optimization to find test paths, but they still have redundancies in the paths generated. Redundancy in generation of test paths requires more transitions to cover the graph and thus more testing efforts are needed.

This paper proposes a FA-based method of optimal test sequence generation for state-based and control flow graph. The proposed FA can generate an optimal set of paths without redundancy. This is a significant advantage over other approaches.

The paper is organized as follows. In Section 2, we will describe the fundamentals of Firefly Algorithm and appropriate formulation of objective function [26]. In Section 3, we provide an overview of proposed method using firefly algorithm and other relevant characteristics. In Section 4, we show the proposed algorithm by examples. Then, in Section 5, we apply that the proposed approach to path generation and provide a brief comparison with other approaches such ACO in the literature. Finally, in Section 6, we draw some conclusions and discuss topics for future research.

2. Related concepts

2.1. Firefly algorithm [19]

The flashing light of fireflies is an amazing sight in the summer sky in the tropical and temperate regions. The pattern of rhythmic flashes is often unique for a particular species. The flashing light can be formulated in such a way that it is associated with the objective function to be optimized, which makes it possible to formulate new optimization algorithms. According to Yang [19], there are three assumptions in firefly algorithm:

1. All fireflies are unisexual and every firefly attracts/gets attracted to every other firefly.
2. The attractiveness of a firefly is directly proportional to the brightness of the firefly. (The brightness decreases as the distance traveled increases.)
3. They move randomly if they do not find a more attractive firefly in adjacent regions [19].

In the above firefly algorithm, the light intensity \( I(r) \) decrease as the distance between two fireflies increases. Intensity varies with distance \( r \)

\[
l(r) = L_0 \exp(-\gamma r),
\]

where \( \gamma \) is the light absorption coefficient, which can often be taken as 1. FA has been applied in spatial fields with different dimensions with promising efficiency and superiority over other algorithms. FA is a metaheuristic algorithm, which assumes that a solution of an optimization problem is encoded as the location of the agent/firefly, while the objective function is encoded as the light intensity. In FA, there are two important issues that make the algorithm successful:

1. variations in light intensity, and
2. the right formulation of attractiveness, which is determined by brightness function, which in turn is associated with the objective function. Attractiveness of a firefly in the search space is directly proportional to the objective function value of the firefly.

2.2. Objective function

The objective function of an optimization problem defines the objective to be maximized or minimized in the search space and domain of design variables [19,26]. A well-formulated objective function will help the search process, while a poorly formulated objective function may lead to inappropriate or wrong solutions. A well-defined objective function reduce the search effort tremendously. Here as the objective function determines the attractiveness/brightness of firefly’s in a state or a node of a graph, where each state or node’s objective function value can be different, derivations of objective function should use key properties of a STD or a CFG.

In the standard FA, light intensity and variations in brightness play a crucial role in encoding the objective function, while designing a good objective function is important to a good problem formulation to be solved by any algorithms. In this paper, the proposed algorithm can adapt and extend the above two entities by considering cyclometric complexity [28], and graph traversal, as discussed in more detail later. We will apply the proposed algorithm into software testing domain. In addition, the intensity factor can be determined by using cyclometric complexity of the graph’s nodes and variation of attractiveness can be implemented by having the absorption coefficient at every edge to be discussed in Section 3.

Therefore, our proposed algorithm uses the following entities in designing the objective function:

1. Cyclometric complexity.
2. Random function.

Cyclometric complexity (CC) [28] is a software metric, which is used to indicate the complexity of a program. On testing strategy of McCabe Basic Path Testing, the number of independent paths generated is equal to the Cyclometric complexity of the graph.

Let \( M \) be an upper bound for the number of test cases that are necessary to achieve branch coverage, and \( N \) be a lower bound for number of test cases to achieve path coverage. CC can be defined as:

Cyclometric Complexity \( CC = M < CC < N \).
Cyclometric complexity is also known as $V(G)$, where $V$ refers to the Cyclometric number in graph theory, and $G$ indicates that the complexity is a function of the graph. In a strongly connected directed graph $G$, the Cyclometric number can be defined as $V(G)=E-N+2$, where $E$ is the number of edges in the control flow graph (CFG), and $N$ is the number of nodes.

Control flow graphs (CFG) represent the control flow of programs [27]. It has nodes connected by edges and has one or more entry or exit points. Nodes are expressed as labeled circles, representing one or more statements. Edges are directed arrows that represent the control flow of the program. CFG is widely used in the analysis of software test cases.

Random function [29]: A random function or a Randomization function is a vector with random variables defined according to the dimensions in which the algorithm is used. Randomization along with scaling parameters is employed to differentiate the dimensions they travel.

The above elements can be used to formulate the objective function with cyclometric complexity as the main element of design in the proposed approach. The attributes such as cyclometric complexity and the random function act as local variables of software testing in determining the objective function. In the following sections, we will describe the main elements of the proposed approach and the formulation and derivation of the proper objective function.

### 3. FA-based approach

The proposed approach (Fig. 1) can generate optimal test sequences that will cover all the transitions. The approach uses the Firefly algorithm, developed by Yang [19]. This approach can provide the optimal set of paths from the STD and CFG [28] of the given software under test. The Data Flow Diagram (DFD) [9] of the algorithm is shown in the Fig. 1.

The proposed algorithm starts with the objective function for the STD provided. Fireflies/solutions are generated at each node with different intensity of brightness at each node using cyclometric complexity [28] and a random function [19]. Every node has been made to have a guidance factor, as discussed later, in guiding the fireflies. Fireflies can traverse the graph by the guidance factor to record the optimal paths.

If an optimal path can be achieved, the fireflies [19] essentially reverse each independent (generated form optimal path) path, starting from the start node of the graph. As fireflies traverse through the path, they undergo changes in their brightness which also varies due to variations in absorption coefficient of edges and simultaneously get attracted to brighter fireflies of other nodes. The proposed algorithm extends the basic FA algorithms assumptions described in Section 2 of the paper with the following assumptions:

**Assumption 1.** Fireflies tend to move towards a brighter firefly as a process of attraction.

In Yang’s original firefly algorithm [19,29], the distance traveled by a firefly is measured by a Cartesian distance/Euclidean distance or time lag between processes/proximity of vertices in a hyperspace/job scheduling/social networking. To date FA has been applied in many areas, however, it has not been applied in software testing. Thus, in our proposed modification of FA, we have to redefine the distance as described below. In contrast to the Cartesian distance

$$r_{ij} = |X_i - X_j| = \sqrt{\sum_{k=1}^{d} (X_{ik} - X_{jk})^2}$$

where $i, j$ correspond to the fireflies at $X_i$ and $X_j$, and $X_{ik}$ is the $k$th component of the spatial coordinate of the firefly, while in our proposed algorithm, we define the distance traveled by a firefly in a graph as

$$r_{ij} = \text{Distance between node/state } i \text{ and } j = \sum e_{ik} = \text{sum of edges}$$

where $r$ is the distance between node $i$ and $j$ and $e_{ij}$ is the sum of edges between $i$ and $j$. Assumed an ideal edge weight $e_{ij}=1$ for every edge in CFG/STD because it is conventionally assumed the tested distance is unity between the nodes.

**Assumption 2.** Fireflies lose their intensity of brightness as they move through the medium of graph.

Yang [19,29] assumed space absorption coefficient $\gamma > 0$

$$X_{i}^{t+1} = X_{i}^{t} + \beta e^{-\gamma} (X_{j}^{t} - X_{i}^{t}) + \alpha d^{t}$$

where in the special case of $\gamma=0$ can lead to a variant of partial swarm optimization PSO [29] which implies a firefly can be seen anywhere in the domain. On the other hand, at other extreme when $\gamma$ tends infinity, we have a random search [29], which implies that fireflies can move randomly.

The proposed algorithm has been tested with absorption coefficient $\gamma=0.1$ at every edge of a CFG per transition in STD. This value of 0.1 is based on a parametric study, which usually provides more idealized paths of traversal on the graph.

The movement of fireflies in the present approach is directed by the guidance factor and guidance matrix to be discussed below. The guidance factor approach makes the fireflies bound to a limited region of the graph and helps fireflies in making decisions at predicate nodes. Traversed fireflies from node to node move as a swarm in a process of attraction, and gets accumulated at the end node. In our algorithm, it prioritizes the paths by

---

**Fig. 1.** Schematic view of the proposed algorithm.
calculating the mean brightness at the end node. The path with the highest mean brightness can be taken as the highest priority path.

Now we describe the entities of the proposed algorithm in more detail, though the basic design of the objective function and random function are described in Section 2. Now we discuss the detailed formulation of the objective function and random function and the definition of Adjacency matrix and Guidance factor/Guidance matrix.

**Objective function or fitness function:** The objective function $f(x)$ corresponds the goal, which associates each solution of the search space with a real value that describes the quality or fitness.
of the solution [13]. Firefly algorithm utilizes a relative fitness association to rank the individual in the population. In this paper, we use a modified objective function in the following form:

Objective Function/Brightness Function \( F(x) \):

\[
F(x_i) = \left[ \frac{1000}{(CC_{i} \cdot \text{rand})} \right]
\]

where \( CC_{i} \) is the Cyclometic Complexity of Node \( i \), and \( \text{rand}() \) is a random value generator. Here the constant 1000 is a scaling factor to maintain brightness values above zero, to avoid purely random search. Random Function \( \text{Rand}() \) is drawn by \( \text{Rand}(\cdot) = N = \left\lfloor \frac{N - i}{0.1} \right\rfloor \). For example, if the total number of nodes is \( N = 8 \), then at Node 1 random function generates \( \text{Rand} = (8 - 1) = 7.9, 7.8, 7.7, \ldots, 7.90, 7.91, 7.92 \ldots \)

Random values of a node are used to generate fireflies with different values of brightness at a particular node.

After defining the objective function, cyclometic complexity, absorption coefficient and distance \( (r) \) values, a given graph or STD is converted to an adjacency matrix and is then used to generate guidance matrix for the graph.

**Adjacency Matrix:** Adjacency matrix is a two dimensional or a three dimensional matrix, depending on the dimensions in which the graph is constructed. It provides the relationship between nodes and edges. Edges between nodes are represented as 1 and rest by zero.

**Guidance matrix:** As described above, a guidance matrix holds guidance factors to probe the fireflies in taking decisions at predicate nodes in choosing the path. It is used for the decision matrix for a given graph. For a firefly at a predicate node, the decision if a path is chosen or not is carried out by referring the guidance factor in the guidance matrix. It blocks the global view of domain or the graph.

The guidance matrix can be defined as

\[
\text{GF} = 1000[(CC_{i} \cdot (N - i) - 0.1)]
\]

where \( CC_{i} \) is the Cyclometic Complexity of Node \( i \), and \( N \) is the total number of nodes. The out degree of a node is the total number of edges that move out from a node \( [1,2] \) and a node with out-degree greater than 1 is defined as a predicate node. Fireflies at a predicate node use the guidance factor discussed above to traverse the node. Therefore, we can define brightness as

Brightness function \( = \frac{1}{\text{Guidance factor}} \).

Thus, a firefly at a predicate node follows the guidance factor with a lower value. We now apply our proposed approach to the generation of software test paths, and we will demonstrate this by an example in Section 4 below using an OFTSG tool in which we have implemented our test algorithm.

### 4. Test example using OFTSG

We implemented our method in our software testing tool, OFTSG. The data flow diagram of the implementation is shown in Fig. 2, and a state diagram of a STD test case is shown in Fig. 3. OFTSG is the implementation of proposed approach as a software testing tool coded in C [ANSI], which is explained below.

Data flow diagram: optimal firefly test sequence generator (OFTSG) (Fig. 2)

1. Generate an XML file from drawn state-based diagram or a CFG
2. Input the XML to the OFTSG
3. XML file generated can used to transform state-based diagram to a CFG
4. OFTSG parses XML and then:
   - Calculates the number of nodes, edges and Cyclometic complexity.
   - Adjacency matrix.
   - Guidance matrix.
   - Out-degree of every node
   - Cyclometic complexity
5. Path traversal [as described in Section 3]
6. Path prioritization.
   - All the paths are made to travel by the fireflies
   - For every path, a mean value of brightness function is calculated and the paths with higher means can have higher priority.

The Fig. 2 above shows the data flow diagram (DFD) of the OFTSG tool; it starts with reading an XML file and then generates an adjacency matrix (see Table 1). Adjacency matrix analysis provides the cyclometic complexity and the out degree of each node which is then used for building the guidance matrix (Table 2). After all required elements are assembled; the adjacency matrix is processed to generate test sequences. Then, the generated test sequences are stored and processed. Every node is associated with a number \( (n) \) of fireflies, and through these paths, fireflies can migrate to the next node of the path by the attraction process. During this process of attraction, fireflies will be able to reach the end node. Thus the mean brightness of the accumulated
fireflies can be calculated to prioritize the path. Fig. 3 below shows a sample STD built to generate a test case.

State-diagram:
Tool Used: Eclipse[IDE] [Unimod – Plug-in].
Unimod plug-in has been used for the generation of state-based diagrams.
Export as: The Generated State diagram can be exported as XML.

**OFTSG TOOL:** OFTSG is the implemented tool of the proposed algorithm.

**Methods/Modules**
**Read (.)** -XML parsing.
–It reads the XML file, parses it and extracts nodes and edges to file.

**Draw (.)**
–It reads the file generated by read function and generates the following elements:
The number of nodes read from XML file is 8.
The number of edges read from XML file is 11
Evaluated Cyclometric Complexity of STD: 5
Calculating Cyclometric Complexity of Each Node:
CC at S1: 5
CC at S2: 5
CC at S3: 4
CC at S4: 4
CC at S5: 3
CC at S6: 2
CC at S7: 1
CC at S8: 100 [END node affinity]
Adjacency Matrix: Adjacency matrix

**Out degrees:**
Out degree for node 3 is 2 a decision state
Out degree for node 5 is 2 a decision state
Out degree for node 6 is 2 a decision state
Out degree for node 7 is 2 a decision state

**Guidance Factor: GF**
GF at node 1
GF = Cyclometric complexity of S1 * (N – i) * 0.1
= 10 * (8 – 1) * 0.1
i = State;
= 345

Node 1: 345
Similarly, we have
S2: 295
S3: 196
S4: 156
S5: 87
S6: 38
S7: 9
S8: 1000

Guidance matrix (Table 2) is just as a look-up/decision table of adjacency matrix with each guidance factor corresponding to every edge.

**Guidance Matrix:**
The next step is the graph processing by using the adjacency matrix and guidance matrix. Fig. 4 describes the processing of Adjacency Matrix using iterations and loops.

The pseudo-code thus generates path 1 as listed below in iteration 1:
Path 1 = 1, 2, 3, 5, 6, 7, 6, 8.

All visited nodes are replaced with zero in the matrix after the first iteration. The next iteration generates the rest of the paths.
For iteration 2. Table 3 below depicts the status of adjacency matrix after visited nodes are removed from the matrix.

**Pseudo Code:**

For k = 1 to N
If Adjacency mat[i][j] not equal to NULL
For i = 1 to N
Count = 0, temp1 = 0, temp = 0
For j = 1 to N
If i greater than or equal to 1 and
j greater than zero
Temp = Adjacency mat[i][j]
If temp is greater than 0
Count + 1
If count is equal to 1
Temp1 = temp
If temp is equal to N
I = N + 1
Else if count is greater than 1
Temp = guidance[i][temp1] > guidance[i][j]
path[i][j] = path[i][temp1]
Adjacency mat[i][temp] = 0;

I = temp – 1; break
End if
End if
End if
End for j

**Fig. 4.** Pseudo-code for adjacency matrix.
Iterations continue until the matrix becomes NULL. Hereafter two iterations, the optimal paths can be obtained and are shown in Table 4.

Table 4 lists the optimal test sequences generated by the OFTSG tool. There is no redundancy in generated paths, and the independent paths generated from the optimal paths to test the total path priority are listed below.

### 4.1. Independent paths generated

| Path 1: | 1,2,3,5,6,7,6,8 |
| Path 2: | 1,2,3,5,5,6,7,6,8 |
| Path 3: | 1,2,3,5,6,7,8 |
| Path 4: | 1,2,3,5,6,7,8 |

Once all independent paths are double-checked, the paths are then used for software testing. Since testing depends on fixed resources, path prioritization is needed.

Test path prioritization involves scheduling over test cases in an ordered manner so as to improve the performance of regression testing. Path prioritization is to find the critical paths that a tester might want to test and/or prioritize. In the current test case, ten fireflies are used at each state/node and they can traverse the independent paths listed above. In the following tables, each box corresponds a firefly with a brightness value at each state/node linked with the cyclometic complexity and random function.

**Firefly generation:** For all the test cases in the paper, ten fireflies are deployed at every node as inputs.

The following tables represent fireflies with brightness value at every state:

<table>
<thead>
<tr>
<th>States</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

After the generation of fireflies at each state, the mean of the fireflies’ brightness is calculated at every end state of a path.

**Mean calculation:**

<table>
<thead>
<tr>
<th>Path 1:</th>
<th>1,2,3,5,6,7,6,8</th>
</tr>
</thead>
<tbody>
<tr>
<td>At S1:</td>
<td>CC=5;</td>
</tr>
<tr>
<td>$F(x_{i1})=100/(5\varphi r(\theta))$; here $\varphi$ and $\theta=[7.9...,7.1]$</td>
<td></td>
</tr>
<tr>
<td>Sample calculation</td>
<td>$F(x_{i1})=100/(5\varphi r(\theta))=2.53$</td>
</tr>
<tr>
<td>Similarly, we have</td>
<td>$F(x_{i2})=F(x_{i1})$</td>
</tr>
<tr>
<td>Here $F(x_{i}) =$ brightness function of firefly $i$ at S1</td>
<td></td>
</tr>
</tbody>
</table>

2.53 2.56 2.6 2.63 2.67 2.7

2.74 2.78 2.82 2.85

At S2: CC=5;

$F(x_{i1})=100/(5\varphi r(\theta))$; here $\varphi$ and $\theta=[6.9...,6.1]$ |

2.9 2.94 2.99 3.03 3.28 3.08

3.13 3.17 3.23

At S3: CC=4 |

$F(x_{i1})=100/(4\varphi r(\theta))$; here $\varphi$ and $\theta=[5.9...,5.1]$ |

3.38 3.44 3.5 3.57 3.63 3.7

3.77 3.84 3.91 4

At S4: CC=4 |

$F(x_{i1})=100/(4\varphi r(\theta))$; here $\varphi$ and $\theta=[4.9...,4.1]$ |

5.1 5.2 5.3 5.4 5.55 5.68

5.81 5.95 6.09 6.25

At S5: CC=3 |

$F(x_{i1})=100/(3\varphi r(\theta))$; here $\varphi$ and $\theta=[3.9...,3.1]$ |

8.54 8.77 9 9.25 9.52 9.8

10.1 10.4 10.7 11.1

At S6: CC=2 |

$F(x_{i1})=100/(4\varphi r(\theta))$; here $\varphi$ and $\theta=[2.9...,2.1]$ |

11.4 11.9 12.34 12.82 13.12 13.33

13.88 14.49 15.15 15.87

At S7: CC=1 |

$F(x_{i1})=100/(4\varphi r(\theta))$; here $\varphi$ and $\theta=[1.9...,1.1]$ |

26.3 27.7 29.4 31.25 33.33 35.71

38.46 41.66 45.66 50

The mean of the brightness of fireflies from Node 1 to Node 8 covering 1→2→3→5→6→7→6→8 is the sum of intensities of
brightness of fireflies accumulated at end node by number of fireflies. That is,

\[ \text{Arithmetic mean of brightness} = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} f[X_{ij}]}{\sum_{i=0}^{n} \sum_{j=0}^{m} f} = 9.271429 \]

where \( f = 1 \) is state/node j is firefly \( f[X_0] \) brightness function \( f \) is constant and \( n \) and \( m \) represent the numbers of nodes and number of fireflies, respectively.

Similarly for Path 2: 1, 2, 3, 5, 5, 6, 7, 6, 8,
we have the mean of brightness of fireflies from Node 1 to Node 8 covering 1 → 2 → 3 → 5 → 5 → 6 → 7 → 6 → 8. So the arithmetic mean of brightness = 8.79.

Path 3: 1, 2, 3, 5, 6, 7, 8
At node 8:
The mean of brightness of fireflies from Node 1 to Node 8 covering 1 → 2 → 3 → 5 → 6 → 7 → 8
Arithmetic Mean of brightness = 9.33

Path 4: 1, 2, 3, 4, 6, 7, 8
At node 8:
The mean of brightness of fireflies from Node 1 to Node 8 covering 1 → 2 → 3 → 4 → 6 → 7 → 8
Arithmetic Mean of brightness = 8.75.

After the calculation of all means, the path with the highest mean value is prioritized as a critical path. In general, the path, which has a higher number of predicate nodes, is assigned with high priority as shown in Table 5.

Table 5 depicts the priority order of the paths, which are categorized as critical paths for test. In the next section, we present a comparison of our proposed approach with ACO.

5. Comparison with existing work

The comparison and analysis of the proposed algorithm with other proposed approaches such as ACO can be carried out for testing software coverage. Nowadays, in the software industry, the testing cost can be approximately 50% of the total cost of a software project, efficient ways of testing software are crucially important in reducing costs.

Ant colony optimization: In the literature, only one approach by Praveen R. Srivastava’s approach using ACO [10,25] has all available data for comparison. Tables 6 and 7 lists the test sequences achieved by our approach and ACO for generating test sequences as shown in Fig. 5 [10]. It can be reduced to a STD by method of graph reduction [30].

Fig. 6 shows the STD derived by the method of graph reduction for the CFG as shown in Fig. 5.

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Path prioritization.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Paths</td>
</tr>
<tr>
<td>1</td>
<td>Path3: 1,2,3,5,6,7,8</td>
</tr>
<tr>
<td>2</td>
<td>Path 1: 1,2,3,5,6,7,8</td>
</tr>
<tr>
<td>3</td>
<td>Path 2: 1,2,3,5,6,7,6,8</td>
</tr>
<tr>
<td>4</td>
<td>Path 4: 1,2,3,4,6,7,8</td>
</tr>
</tbody>
</table>

Table 6
Optimal paths generated by OFTSG.

<table>
<thead>
<tr>
<th>No</th>
<th>Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Path 1: 1,2,3,4,6,8,2,9</td>
</tr>
<tr>
<td>2</td>
<td>Path 2: 4,5,9</td>
</tr>
<tr>
<td>3</td>
<td>Path 3: 6,7,2</td>
</tr>
</tbody>
</table>

Table 7

<table>
<thead>
<tr>
<th>Paths</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1</td>
<td>0-1-2-3-4-13</td>
</tr>
<tr>
<td>Path 2</td>
<td>0-1-2-4-5-6-7-8-9-13</td>
</tr>
<tr>
<td>Path 3</td>
<td>0-1-2-3-4-5-6-10-11-4-5-6-7-8-9-13</td>
</tr>
<tr>
<td>Path 4</td>
<td>0-1-2-3-4-5-6-10-12-4-5-6-7-8-9-13</td>
</tr>
</tbody>
</table>

Fig. 5. Control flow graph.

The above stated node graph can be reduced to fit into state-based diagram, converging linear nodes to a single node. The reduction is based on syntactical information of the graph, not by the semantic information.

<table>
<thead>
<tr>
<th>Nodes clustered/transformed</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2-3</td>
<td>S1</td>
</tr>
<tr>
<td>4</td>
<td>S2</td>
</tr>
<tr>
<td>5</td>
<td>S3</td>
</tr>
<tr>
<td>6</td>
<td>S4</td>
</tr>
<tr>
<td>7-8-9</td>
<td>S5</td>
</tr>
<tr>
<td>10</td>
<td>S6</td>
</tr>
<tr>
<td>11</td>
<td>S7</td>
</tr>
<tr>
<td>12</td>
<td>S8</td>
</tr>
<tr>
<td>13</td>
<td>29</td>
</tr>
</tbody>
</table>
Table 6 shows the optimal paths generated by OFTSG on the STD as shown in Fig. 6. Paths 1, 2, 3 cover the entire STD and the sequence has no redundant paths.

Tables 6 and 7 represent the test sequence paths generated by two different algorithms the proposed FA approach and the ACO, respectively. By using the reduced graph, Table 6 can be rewritten as Table 8.

As we can see from these results that the proposed approach does not repeat any part of the path that has already been traversed. This is a major advantage of the firefly algorithm based approach. For example, in Table 8, the path traversed by the firefly indeed found the optimal paths. So the proposed FA approach minimizes the redundancy in the paths covered. Compared with ACO, the path segment 0→1→2→3→4 is traversed only once here. By comparing above two tables, we can find redundancy in the Table 7 in traversing the paths generated by ACO, Fig. 7 depicts a comparison of transition graph of ACO and FA versus the number of repetitions, and Fig. 8 shows the redundancy convergence percentage in FA and ACO. This means that FA generates better paths with lower or no redundancy.

In Fig. 8, as the number of states increases, the redundancy as the number of states for processing also increases, this is due to the fact that high cyclometric complexity in the graph can lead to strict dependency between nodes. It is worth pointing out that the proposed FA-based approach eliminates such a problem and improve the efficiency of finding critical paths among the test case paths. The convergence rate of FA is faster, compared with ACO and other similar techniques. Table 9 lists the test case, representing the percentage of redundancy with respect to the number of states and the cyclometric complexity of the state machine. FA exhibits only a small amount of redundancy due to high cyclometric complexity of the graph. High cyclometric complexity results in strong dependencies among the nodes, thus making some transition paths redundant.
In addition, our new approach produces zero repetition of paths, and thus makes it more efficient than other algorithms.

6. Conclusion

By using a good combination of firefly algorithm and graph reduction, we have extended the standard firefly algorithm to generate optimal discrete and independent paths for software testing. The proposed approach is the first of its kind in the field of software testing, which minimizes the number of test cases by optimizing the test paths for test cases. Graph reduction and state-based transformation also help to ensure the right code coverage for testing. The simulation results suggest that our firefly algorithm based approach has produced the optimal paths below a given number of independent paths. Subsequently, it can minimize the test efforts and provides the best critical test paths. It can be expected that this method can also be applied for other expensive testing procedure by choosing the best possible test paths for other applications. This can form a topic for further research. In addition, extensive parametric studies on the algorithm dependent parameters many provide further insight for new ways to fine-tune the algorithm so as to suit a wider range of applications.

References