Test sequence optimisation: an intelligent approach via cuckoo search

Praveen Ranjan Srivastava*, Chandolu Sravya, Ashima, Sai Kamisetti and Manogna Lakshmi

Department of Computer Science and Information Systems, Birla Institute of Technology and Science (BITS), Pilani-333 031, Rajasthan, India
Fax: 01596-244183
E-mail: praveensrivastava@gmail.com
E-mail: schandolu91@gmail.com
E-mail: ashima.bits@gmail.com
E-mail: saikamisetti@gmail.com
E-mail: manogna.l.n@gmail.com
*Corresponding author

Abstract: Exhaustive testing is never the appropriate approach in software testing as this violates the principle of cost effectiveness in software development as well as puts time and other resources at stake. This explains the need to optimise the process of software testing. The present paper gives an overview of cuckoo search algorithm and its role in software coverage optimisation. This paper proposes an algorithm to generate optimised test sequence(s), which obtains 100% software coverage based on cuckoo search.

Keywords: cuckoo search; heuristic algorithm; optimisation; test sequences.


Biographical notes: 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 using meta-heuristic techniques. His research areas are software testing, quality assurance, quality attributes ranking, testing effort, software release, test data generation, agent oriented software testing, and soft computing techniques. He has published more than 60 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.

Chandolu Sravya is an undergraduate student presently doing her MSc (Tech.) Information 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, image processing, algorithm design, computer architecture and computer networks.

Ashima is an undergraduate student presently doing her MSc (Tech.) Information 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, Java desktop/mobile application development and SQL/Oracle/MySQL database application development.

Sai Kamisetti is an undergraduate student presently doing her MSc (Tech.) Information Systems in the Computer Science and Information Systems Department at Birla Institute of Technology and Science, Pilani, India. Her areas of interest lie in mobile computing, software engineering and compilers.

Manogna Lakshmi is an undergraduate student presently doing her BE in Electricals and Electronics and MSc (Tech.) Information Systems at Birla Institute of Technology and Science, Pilani, India. Her areas of interest lie in artificial intelligence and pattern recognition.
1 Introduction

Software engineering is an organised approach to the analysis, design, assessment, implementation, test, deployment and maintenance of software (Laplane, 2007). Monolithic development is not effective for modern system development. Hence, various phases have been introduced into the software development process, called software development life cycle (SDLC). Among all the phases of SDLC, testing is most important as it accounts for most of the project resources. The objective of software testing is to find errors and programme structure faults. Software testing must be done in a professional and effective manner (Patton, 2009). Software testing must be performed in an orderly sequence, taking care of the order of events, which control the system to be tested. A sequence of such events, which together define a specific testing purpose, is called a test case (Muhammad, 2008).

Usually, the number of test cases is very high to software that is devoid of errors. Exhaustive testing is not the ideal way of testing because the resources would not be utilised in an optimised way, which is very much necessary for cost effectiveness (Pressman, 2010). Therefore, the test cases that we need to generate should be optimal and also should cover the entire software and reveal possibly all the errors that exist. Through test sequence generation, tester or manager not only can locate very easily any defect/error or any kind of failure in the software systems but also help in reducing the high cost associated with software testing. This depicts the lead role of optimisation in test sequence generation.

The paper is structured as follows. Section 1 is the introduction. The background study for this paper is given in Section 2. The introduction to the concept of cuckoo and its brooding behaviour is described in Section 3. Proposed methodology for the test sequence optimisation, i.e., the algorithm (that is developed based on cuckoo search) is explained in Section 4. The analysis of the procedure is presented in Section 5. A case study, (a real-time example) for the algorithm is explained in Section 6. Section 7 of the paper gives a comparative study between algorithms, and Section 8 contains conclusions.

2 Background work

It is essential to ensure the correctness of the entire software, testing for errors in every part of the software becomes important, which essentially means that, correctness with completeness is what a software development team is looking for. Besides, generation of test sequences with no optimisation adds to the cost of carrying out testing. Various algorithms (Srivastava and Baby, 2010; Srivastava, 2009; Diaz et al., 2003) have been developed to generate optimum test sequences with maximum software coverage, having their own merits and demerits.

Genetic algorithm (GA) (Cooper et al., 1999) based on Darwin’s theory of evolution generates test sequence automatically which maximises the software coverage by using multiple test paths instead of single test path (Cooper et al., 1999).

In research paper (Dounga-ard et al., 2008), according to the transition diagram shown, the optimum test sequence that is generated with maximum software coverage does not ensure the coverage of all transitions (edges). Therefore, this does not ensure the validation of entire schema, which makes the GA (Dounga-ard et al., 2008) an inefficient method for generating optimised test sequence(s) (particularly with respect to completeness).

Meta-heuristic ant colony optimisation (Srivastava and Baby, 2010) is another technique developed with the same motive. Here entire software coverage is ensured but the initial parameters to be set as an input are many in number, like heuristic value ($\eta$) for every transition, pheromone level ($\tau$) for every transition, visited status ($Vs$) for every state, probability ($P$) for each transition, $\alpha, \beta$. Besides, there is possibility of repetition of transitions (though limited), thus adding to the cost of testing.

Meta-heuristic technique particle swarm optimisation (Windischet al., 2007) depending on the value of the parameters (Aloka et al., 2011) involved, due to which the solution does not converge in some specific cases.

In the research paper, automated software testing using a meta-heuristic technique based on tabu search (Diaz et al., 2003), an effort was made to generate test sequences that ensure high branch coverage. The technique does not ensure full path coverage, full loop coverage and all conditions coverage (predicate conditions coverage).

In research paper (Mala and Mohan, 2010), according to the comparative study done, it has been shown that the solution that is obtained from GA (Cooper et al., 1999) gets stuck in local optima, because of which the output solution is not the global optimal solution. From the above, it can be seen that many researchers have been trying to optimise the test sequences with maximum software coverage, but every research has its own merits, demerits and constraints that results in either full software coverage and not fully optimised solution (Srivastava and Baby, 2010) or an optimised solution but without full transition coverage (Diaz et al., 2003; Doungsa-ard et al., 2008).

This paper fills in the loopholes of the previous works, as it provides a solution that is optimised and ensures full transition (edge) coverage. The present paper is inspired from the cuckoo search algorithm (Yang and Deh, 2010), which is based on the cuckoo brooding behaviour as explained in the next section. Cuckoo behaviour takes the advantage of lesser parameters and assurance that a better egg (cuckoo egg or host egg depending upon fitness) is being passed to the next iteration, which makes it a better and efficient algorithm. Reducing the number of candidate host eggs in each iteration prevents the cuckoo’s egg from getting stuck at the local optima. The next section proceeds with the introduction to cuckoo and its brooding behaviour.
3 Introduction to cuckoo and its brooding behaviour

Cuckoos are generally known for their sweet voices, but they have an aggressive reproduction strategy. Some types of cuckoos lay their eggs in communal nests, though they may remove others’ eggs to increase the hatching probability of their own eggs (Payne et al., 2005; Lotem et al., 1992). The female lays her eggs in the nest of another species so that the surrogate parents unknowingly raise her offspring. Sometimes the cuckoo’s egg in the nest is discovered and the surrogate parents throw it out or abandon the nest and start their own brood elsewhere in some other place (Lotem et al., 1992). Exploiting this brooding behaviour of cuckoo, the following algorithm was developed.

Cuckoo search (Yang and Deb, 2010) is a technique for optimisation of the test sequences (Devadas et al., 1989). The algorithm is a bio-inspired heuristic algorithm (Chun et al., 1998), supporting the fact of ‘survival of the fittest’ (Masters, 2004). Some algorithms work by starting with a basic solution and gradually adding more information that leads to formation of the best solution from the pool of solutions, while some algorithms start with a pool of solutions and boil down to the best solution, by discarding the worst solutions from the pool, by comparing among the solutions. Cuckoo search belongs to the second kind of algorithms.

Cuckoo search (Yang and Deb, 2010), as described by the authors, has some assumptions, which shall remain the same here. For ease of understanding, we state them:

1. Each cuckoo lays a single egg (solution in the present perspective) at any given point of time, which is dropped by that cuckoo into any randomly selected nest.
2. The nest that has better eggs (better solution) as compared to the remaining, shall remain, get hatched and carry on to the future generations.
3. The number of nests that are considered as host nests is a constant, and the probability with which the host finds an outsider’s egg is [0, 1].

This paper briefs about cuckoo search algorithm, which is used to find optimised test sequence(s) in the design phase of the SDLC. The following section explains how these conditions are applied to the scenario and the process of achieving the optimised test sequences.

4 Proposed methodology

Before using the cuckoo algorithm, the user needs to know the different modules that are present in the design phase, the way these modules are related (connected). An activity diagram (an UML representation) serves this purpose (Larman, 2009). The user then converts this activity diagram into a graph, which will be the input graph for the proposed algorithm. This is followed by the process of applying the cuckoo algorithm. This flow of data is represented in the following Figure 1.

An optimal solution, with maximum coverage is to be determined. Coverage refers to traversal of edges and nodes present in the graph. The classical optimisation theory states that, to obtain an optimal solution for an objective function, to reach the final optimal solution, only the optimal sub-solutions must be considered. If any non-optimal sub-solutions are considered, then the actual optimal solution cannot be reached. The intent of the algorithm is to obtain such optimal sub-solutions to build a final optimal solution. This fact becomes evident in the sections ahead.

Figure 1 Process flow diagram

The activity diagram needs to be converted into a graph by assigning weights to each edge based on the factors that needs to be taken into consideration during testing, like time, cost, etc. These weights that are being assigned by the user are later called static weights. Thereby, a graph is obtained from an activity diagram. This obtained graph is given as an input to the cuckoo algorithm, which is further engineered to obtain sequences. The cuckoo algorithm performs its calculations on sequences. To create these sequences, one has to understand certain terms that are used in the algorithm.

- **Static weight**: A weight is assigned by a user for each edge (based on the factors that are to be taken into consideration during testing, like time, cost, etc.). This value remains constant throughout.

- **Dynamic weight**: It is a quantum, which helps in differentiating an edge between two modules in different sequences. It forms the crux of the working. Based on this quantum, the optimality is actually met. It is calculated by forming a dependency on the adjacent modules of the graph to determine the value. This dynamic weight of the edge is computed by a formula,

\[
\text{Dynamic weight of the edge from } N_i \text{ to } N_{i+1} = \text{indegree of } N_{i+2} \times \text{outdegree of } N_i
\]

where \(N_i, N_{i+1}, N_{i+2}\) are three consecutive nodes in the path that is under consideration within sequence.

The dynamic weight function is formulated on the basis of the in-degree and out-degree of the vertices for the primary
reason that each edge has a dependency on the next forthcoming module, and in a graph representation, one of the many ways to make use of this dependency is through degree of the node. To make the stated point more clear, assume a case of design where initially say a module \( N_i \) has its design completed and module \( N_{i+1} \) is to be designed. To design the module \( N_{i+1} \), the engineer needs to know what is expected as an output from that module, because the output of module \( N_{i+2} \) will be given as input of \( N_{i+2} \). The dependency of adjacent modules in the design phase can be obtained from the requirement analysis phase, which comes before the testing phase in SDLC.

The above dynamic weight function works for all nodes other than end node. At the end node, there exists no other node following it, thus, the value of the term, in-degree of \( N_{i+2} \) in the formula is assumed to be 1, so that it does not nullify the effect of the other term in the formula. Hence, the calculation of the dynamic weights is performed.

- **Main path:** A path from the designated start node to the designated end node. This path is referred to as the main path for the sequence.
- **Sub path:** A sub path is a path, which starts from a node that belongs to the main path and ends at another node that belongs to the main path itself.
- **Sequence:** A sequence is a set of one main path and few sub paths, which together ensure that all nodes and edges in the graph are traversed. All edges are traversed exactly once, but one needs to observe the fact that a node is traversed each time an edge is either leaving out from it or entering into it.

Once these terms are understood, the sequences can be obtained by following the procedure stated below.

To generate the sequences, the user has to begin by generating the main paths. To generate these main paths, the user needs to determine and designate the start node and end node. With the designated start and end nodes, all possible paths between these two nodes would be generated. Each of these paths is an individual main path. For each main path, sequence(s) will be generated with the following procedure. For every main path, there would be one or more sub paths, which constitute the corresponding sequence.

To obtain a sub path from a main path, begin by placing a flag ‘marked’ for all the nodes in the graph that are included in the main path. The traversal starts from the start node in the main path.

A sub path is started from any node in the main path whose out-degree is greater than one and ends with a node in the main path whose in-degree is greater than one. Once a sub path is generated, the above step is performed again but with a slight change in the main path. The main path now will be the main path in the previous iteration together with the sub path recently generated. Iterating this procedure for every main path, sub-paths are generated till all the nodes and edges are traversed. All these sub-paths along with the main path constitute a sequence.

Say for example, for a graph \( G \), a main path \( A \) is determined. Following the method stated above produces a sub path \( B \), where the start and end nodes both belong to \( A \). Considering that a few nodes and edges are yet to be covered, requires the user to carry out more iteration. To carry out the next iteration, the main path for that iteration shall be a union of paths \( A \) and \( B \). Thus, sequences are generated. A practical example is demonstrated in further sections.

### ALGORITHM: (The cuckoo algorithm)

**Input:** A graph (which can contain self-loops, bridge edges and nodes) representation of the modules with their interconnectivity, weights given to the edges between modules, and with the start node and end node designated/mentioned.

**Output:** Optimised test sequence(s).

**Procedure:**

1. **Step 1:** The graph that is given as input is first engineered to obtain sequences from it, ensuring completeness, in the sense that every edge and node is covered at least once, say we obtained \( n \) sequences (\( n \) nests). These \( n \) nests correspond to the \( n \) host nests.

2. **Step 2:** Define a function \( f(x) \) as \( \min \{ f(x) \} \) (here it is \( \min \) because we are considering costs involved as static weights), where \( f(x) \) is the fitness function that shall depend on the sequences.

3. **Step 3:**
   - **Step 3a:** Calculating the dynamic weights of the edge transitions in the sequence.
   - **Step 3b:** Calculate the product of the static weight (the weight assigned in the graph explicitly for the edges) and the dynamic weights calculated in Step 3a. This step is done over all the edges.
   - **Step 3c:** Calculate the sum of the products (calculated in Step 3b) over all the edges in the sequence.

4. **Step 4:** Update the function value based on the fitness function, and pass those sequence(s) that were used in updating the function \( f(x) \) value, to the next iteration and discard the remaining sequences (the sequences that were used in comparison), which is analogous to discarding some fraction of the worst nests and retaining the best nests. In case of tie, in fitness values, all the corresponding sequences participating in the tie shall be passed to the further iteration.

5. **Step 5:** If all the sequences have been checked and compared, then end the procedure. Else, go to Step 3 with another sequence.

The way how the algorithm works and how the computations are made, is illustrated by an example in the sections ahead.
5 Analysis of the proposed approach

The proposed algorithm in this paper follows the basic cuckoo behaviour and uses this genetic based approach to solve the test sequence optimisation problem. Many meta-heuristic algorithms (Srivastava and Baby, 2010; Srivastava, 2009; Díaz et al., 2003) do not offer total path coverage (as mentioned in the background section) whereas the risk of not obtaining a total coverage of all transitions is not there in the present algorithm.

In the algorithm that has been specified, the fitness function has to be a minimising function because the edge weights are costs involved in the process under evaluation, and intuitively it is desirable that the cost is minimum in the very sense. If the parameters in question were about reliability (say) instead of cost, then the fitness function would have been changed to a maximising function, since maximum reliability is desired from any product. The initial fitness value shall be –infinity (minus infinity), instead of infinity. The rest of the algorithm shall remain the same. This is the beauty of the algorithm.

To run the algorithm smoothly, the user needs to give an input in the format described as follows. The graph that was obtained from the activity diagram (this can be achieved by software generating graph from activity diagram, or can be worked out carefully by hand), should have weights assigned to the edges between modules. These weights shall be dependent on factors like time spent, labour cost, and the like factors. The start and end nodes are to be marked as well.

If the graph contains self-loops, algorithm generating sequences must ensure that the loop is included in only one path of the sequence and that too is traversed exactly once, because the module if tested once, would serve the purpose. It need not be checked every time. This is a constraint on the sequence generation. Another constraint is that the present algorithm is limited to just a single start and a single end case in a graph (input graph).

The real-time examples, which have followed the above specified constraints, are considered and worked upon.

GAs can be used to optimise problems based on the principle of survival of the fittest in which better designs are always preferred. Many algorithms are hence proposed in this area to solve optimisation problems like GA, particle swarm optimisation, and ant colony optimisation, etc. Cuckoo algorithm gives optimised results with a high performance as illustrated in the original published paper (Yang and Deb, 2010).

6 Case study

In order to find out the actual working of the algorithm, a real-time example called ‘warning system’ (National Instruments, 2009) has been tested in the implementation done in the C language. There are seven total states (which are equivalent to seven modules in a programme) and the edge transition weights are also given. The complete state transition diagram is given in the following Figure 2.

The abstract (National Instruments, 2009) of the problem goes like this ‘This is a temperature warning system and it checks temperature changes for every 1 sec. After the acquisition has been made, analysis is done regarding the max temperature initialised and current room temperature. If the warning comes out as true, then the system stores the read data into data log and performs time check. If the time elapsed is neither neither 1sec nor the user has stopped the system, then it stays in that state and performs time check again. If time elapsed is 1 sec and user still does not stops the system then it goes to the acquisition part again and entire process starts over. But if user stops the system then the device stops. If warning comes out to be false then it directly goes to the time check state and process repeats as explained above.

Figure 2  CFG for the warning system

Hence, following the algorithm proposed in section, a total of three main paths were generated. Also there are six test case sequences that are possible around these main paths. The data pertaining to the input graph is as given below.

<table>
<thead>
<tr>
<th>Node</th>
<th>Out degree</th>
<th>In degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
The sequences in the following table are generated following the procedure described in Section 4. The meaning of each sequence, as was explained earlier is maximum coverage with minimum repetitions. For example, Sequence 1 consists of three transitions, which cover all transitions with minimum possible repetition in the nodes and no repetition in the transitions.

**Table 2** Sequences

<table>
<thead>
<tr>
<th>Seq no.</th>
<th>Sequence transitions</th>
<th>Fitness value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 6 -&gt; 7</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 5 -&gt; 6 -&gt; 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 -&gt; 6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 6 -&gt; 7</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 5 -&gt; 6 -&gt; 3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 6 -&gt; 6 -&gt; 7</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 5 -&gt; 6 -&gt; 3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 5 -&gt; 6 -&gt; 6 -&gt; 7</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 6 -&gt; 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 5 -&gt; 6 -&gt; 7</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 6 -&gt; 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 -&gt; 6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1 -&gt; 2 -&gt; 3 -&gt; 4 -&gt; 5 -&gt; 6 -&gt; 7</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>4 -&gt; 6 -&gt; 6 -&gt; 3</td>
<td></td>
</tr>
</tbody>
</table>

**Initialise:** The F(x) value is set to be infinity (a very high value). This is done to ensure that every existing possible solution can be checked and accommodated into the procedure.

**Iteration 1:** A sequence is chosen randomly; let it be the 3rd one. The sequence says that the main path (initial path used to generate the complete sequence) is 1, 2, 3, 4, 6, 6, 7. To calculate the dynamic weights of say the edge between 2 and 3, the Nᵢ is node 2, Nᵢ+1 is node 3 and Nᵢ+2 is node 4. Hence, the dynamic weight as per the formula mentioned in the previous sections, evaluates to 1 (the calculation below shows explicit calculation of dynamic weight as well). Similarly, the dynamic weight is calculated for the other edges. The product of static weights and dynamic weights is calculated and summed over all edges to give the fitness of that sequence.

Fitness(seq 1) = Σ f(x) of all the sub paths in the seq.

Fitness(seq 3) = (1×2×2+1×1×3+1×3×4+2×3×1 +1×1×6+3×1×3)+ (2×3×8+3×2×5+3×1×2) = 115

Update F(x) to be 115, since it is less than the value infinity previously stored in F(x). Pass this sequence to be compared in the next iteration. Mark Sequence 3 as checked.

**Iteration 2:** Choose a sequence randomly from remaining unchecked sequences; let it be the 5th one. Repeating the process done in previous iteration,

Fitness(seq 5) = (1×2×2+1×1×3+1×1×4+2×3×1 +1×1×6+3×1×3)+ (2×2×8+3×1×2)+(3×1×5) = 85

Update F(x) to be 85, since it is less than the value 115 previously stored in F(x). Discard Sequence 3 and pass the Sequence 5 to next generation. Mark Sequence 5 as checked.

**Iteration 3:** Choose a sequence randomly from the remaining unchecked sequences; let it be the 2nd one. Repeating the process done in previous iteration,

Fitness(seq 2) = (1×2×2+1×1×3+1×3×4+2×1×8 +3×1×3)+ (2×3×1+1×3×6+3×2×5+3×1×2) = 104

No updating is required F(x), since it is less than the value 104 calculated by fitness function. Discard Sequence 2 and pass the Sequence 5 to next generation. Mark Sequence 2 as checked.

**Iteration 4:** Choose a sequence randomly from the remaining unchecked sequences; let it be the 1st one. Repeating the process done in previous iteration,

Fitness(seq 1) = (1×2×2+1×1×3+1×3×4+2×1×8 +3×1×3)+ (2×3×1+1×3×6+3×2×5+3×1×2) = 83

Update F(x) to be 83, since it is less than the value 85 previously stored in F(x). Discard Sequence 5 and pass the Sequence 1 to next generation. Mark Sequence 1 as checked.

**Iteration 5:** Choose a sequence randomly from the remaining unchecked sequences; let it is the 6th one. Repeating the process done in previous iteration,

Fitness(seq 6) = (1×2×2+1×1×3+1×1×4+2×3×1+1 ×1×6+3×1×3)+ (2×3×8+3×2×5+3×1×2) = 116

No updating is required F(x), since it is less than the value 116 calculated by fitness function. Discard Sequence 6 and pass the Sequence 1 to next generation. Mark Sequence 6 as checked.

**Iteration 6:** Choose a sequence randomly from the remaining unchecked sequences; let it is the 4th one. Repeating the process done in previous iteration,

Fitness(seq 4) = (1×2×2+1×1×3+1×1×4+2×3×1+1×1×6+1×3×6+3×1×5+3×1×3)+ (2×2×8+3×1×2) = 97

No updating is required F(x), since it is less than the value 97 calculated by fitness function. Discard Sequence 4 and pass the Sequence 1 to next generation. Mark Sequence 4 as checked.

**Iteration 7:** On re-iterating the procedure of randomly choosing sequences, there are no sequences left unchecked. This completes all the iterations and the sequence with the least fitness value, here sequence 1 with value 83 is the optimal solution.

The output given after performing cuckoo search is the sequence number 1. Next section describes the apparent
advantages of this algorithm compared to others in the same field.

7 Comparison study

7.1 Comparison with ant colony optimisation

The comparison study in the present scenario can be well explained with respect to the ant colony optimisation technique presented in the paper (Srivastava et al., 2009). Although both these algorithms ensure full path coverage, there are some advantages of the cuckoo search over the other. The aspect of redundancy is well checked in the present algorithm. The example to which the ant colony optimisation technique (Srivastava et al., 2009) is applied is presented in Figure 3.

Figure 3 CFG used for comparison of ACO and cuckoo

The optimal path sequence, generated via ant colony optimisation technique proposed (Srivastava et al., 2009) is as follows:

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
<th>Strength</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1</td>
<td>Start, 1, 2, 3, 4, end</td>
<td>7.5</td>
<td>4</td>
</tr>
<tr>
<td>Path 2</td>
<td>Start, 1, 2, 3, 4, 5, 6, 10, 11, 4, 5, 6, 7, 8, 9, end</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>Path 3</td>
<td>Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, end</td>
<td>18.25</td>
<td>3</td>
</tr>
<tr>
<td>Path 4</td>
<td>Start, 1, 2, 3, 4, 5, 6, 10, 12, 4, 5, 6, 7, 8, 9, end</td>
<td>28.785</td>
<td>1</td>
</tr>
</tbody>
</table>

The above solution implies that, in order to achieve optimal path sequence along with 100% software coverage, every path, comprising of the sequence solution begins from the start module of the control flow graph (CFG) and ends with the end module.

The sequences generated for the proposed cuckoo algorithm in this paper, for the above CFG are as follows:

Sequence 1

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, 5, 6, 10, 11, 4, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>6, 7, 8, 9, end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>10, 12, 4</td>
</tr>
</tbody>
</table>

Sequence 2

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, 5, 6, 10, 12, 4, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>6, 7, 8, 9, end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>10, 11, 4</td>
</tr>
</tbody>
</table>

Sequence 3

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>6, 10, 11, 4, end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>10, 12, 4</td>
</tr>
</tbody>
</table>

Sequence 4

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>6, 10, 12, 4, end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>10, 11, 4</td>
</tr>
</tbody>
</table>

Sequence 5

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>4, 5, 6, 7, 8, 9, end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>6, 10, 11, 4</td>
</tr>
<tr>
<td>Sub 3</td>
<td>10, 12, 4</td>
</tr>
</tbody>
</table>

Sequence 6

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>4, 5, 6, 7, 8, 9, end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>6, 10, 12, 4</td>
</tr>
<tr>
<td>Sub 3</td>
<td>10, 11, 4</td>
</tr>
</tbody>
</table>

Sequence 7

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Start, 1, 2, 3, 4, end</td>
</tr>
<tr>
<td>Sub 1</td>
<td>4, 5, 6, 10, 11, 4</td>
</tr>
<tr>
<td>Sub 2</td>
<td>10, 12, 4</td>
</tr>
<tr>
<td>Sub 3</td>
<td>6, 7, 8, 9, end</td>
</tr>
</tbody>
</table>
On comparing the sequences generated by ACO and the sequences in the present case (cuckoo), a lot of redundancy can be observed within the ACO generated paths. This is because for every path, the ant has to start from the initial point, unlike the case in cuckoo. Whereas in cuckoo generated sequences, the only redundancy is the start and end of the sub paths, which are nodes of the main path generated for every sequence.

**Figure 4** Comparison graph (see online version for colours)

![Comparison graph](image)

Note: For cuckoo and ant colony optimisation

Any sequence generated in this proposed algorithm, when compared to the optimal path sequence generated by ACO, has very low redundancy, which leads to lesser number of iterations. Based on this, a graph (Figure 4) can be plotted on the number of iterations for both the algorithms. Number of iterations, here refers to the redundancy. The more the redundancy, the higher is the number of iterations required to encapsulate the features of an algorithm.

From the graph, it can be concluded that number of iterations in cuckoo algorithm is surely less than in ACO algorithm.

### 7.2 Comparison with GA

The present algorithm is also compared with GA, as follows.

The solution given by the GA, according to Doungsa-ard et al. (2008) is generated by selecting the best test data from each possible path, which is used to find the overall transition coverage (Doungsa-ard et al., 2008). GA can be compared with the present algorithm with an example, the enrolment system (Figure 5, state chart diagram of enrolment system), taken from Doungsa-ard et al. (2008).

Enrolment system process is depicted in Figure 5. An enrolment system diagram describes the activity of the enrolment for each course. The students enrol for the course. When the course is full, no more students can enrol for the course. The course can be closed for enrolment anytime.

The possible paths generated by GA for class management system (Doungsa-ard et al., 2008) are:

- null, t1, t2, t4, t5
- null, t1, t2, t5e, t8
- null, t1e
- null, t1, t2, t3e
- null, t1, t2, t3, t4, t5e, t6
- null, t1, t2, t3, t4, t4e
- null, t1, t2, t2e.

**Figure 5** The enrolment state machine diagram

![Enrolment state machine diagram](image)
The test sequences for transitions covered by cuckoo are represented by the dotted lines as shown in Figure 6.

Sequences generation by the proposed intelligent cuckoo approach can be as follows. This sequence is one of the many possible sequences generated using the intelligent cuckoo approach.

The nodes are numbered for convenience. Start state – start, proposed – i, scheduled – ii, open for enrolment – iii, full – iv, closed for enrolment – v, final state – end. The following paths are shown in the following format: ‘Node A – Edge AB – Node B’ (where A and B is states and AB is edge)

<table>
<thead>
<tr>
<th>Path</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub 1</td>
<td>i – t1e – end</td>
</tr>
<tr>
<td>Sub 2</td>
<td>ii – t2e – end</td>
</tr>
<tr>
<td>Sub 3</td>
<td>iii – t8 – v – t5e – end</td>
</tr>
<tr>
<td>Sub 4</td>
<td>iv – t6 – iii</td>
</tr>
<tr>
<td>Sub 5</td>
<td>iv – t7 – v</td>
</tr>
<tr>
<td>Sub 6</td>
<td>iv – t4e – end</td>
</tr>
</tbody>
</table>

As seen from the above sequence, the cuckoo search algorithm ensures zero redundancy in the transitions. Whereas, the result generated for the same example using GA shows redundancy of transitions. Based on this observation, a graph (Figure 6) can be drawn to show the redundancy of transitions for both the algorithms.

The transition coverage, in case of GA is dependent on chromosome length which does not ensure 100% coverage all the time (Doungsa-ard et al., 2008), whereas the intelligent cuckoo approaches always ensures 100% coverage as seen from Figure 6.

Besides, the maximum coverage of GA for the enrolment example is just 64.29% of the entire transitions (Doungsa-ard et al., 2008).

By ensuring 100% coverage all the time, the proposed intelligent cuckoo approach overcomes the drawbacks observed in GA.

The following graph demonstrates how the proposed intelligent cuckoo approach is better off than other algorithms compared in this paper.

Figure 7 shows that average number of transitions covered by the solution generated by the intelligent cuckoo approach is 100% where as due unreachable transitions in GA (Doungsa-ard et al., 2008), the average total number of transitions covered by GA reduces and becomes less than 100%.

Figure 8 explains that the solution provided by GA has redundancy in the path coverage (repeated traversal of edges) which is relatively high when compared to the zero redundancy (every edge is traversed just once) in the final solution generated by intelligent cuckoo approach. The only redundancy that occurs in the solution of the proposed cuckoo search is that of the bridge nodes which need to be
traversed repeatedly to cover all their adjacent edges. This redundancy is inevitable and is the minimum possible in case of redundancy in traversal of nodes.

Figure 8  Graph comparing level of redundancy with cuckoo and GA (see online version for colours)

8 Conclusions

The constraints like large number of test cases (although avoiding exhaustive testing, ensuring the completeness), time factor, cost, limited skilled labour show the necessity and importance of optimisation in the process of software testing. This paper describes in detail the role of testing, its importance and how well the proposed algorithm, ‘test sequence optimisation – an intelligent approach via cuckoo search’ accommodates the need of optimisation. The algorithm is applied to real-time examples. A vivid description of its working is given. The paper also shows a comparison study of cuckoo search with various other leading algorithms. The algorithm in this paper may be further enhanced when worked with taboo search.

Acknowledgements

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

References


