Automated Test Data Generation Using Cuckoo Search and Tabu Search (CSTS) Algorithm

Praveen Ranjan Srivastava, Rahul Khandelwal, Shobhit Khandelwal, Sanjay Kumar and Suhas Santebennur Ranganatha

Abstract. Software testing is a very important phase in the development of software. Testing includes the generation of test cases which, if done manually, is time consuming. To automate this process and generate optimal test cases, several meta-heuristic techniques have been developed. These approaches include genetic algorithm, cuckoo search, tabu search, intelligent water drop, etc. This paper presents an effective approach for test data generation using the cuckoo search and tabu search algorithms (CSTS). It combines the cuckoo algorithm’s strength of converging to the solution in minimal time along with the tabu mechanism of backtracking from local optima by Lévy flight. The experimental results show that the algorithm is effective in generating test cases optimally and its performance is better than various earlier proposed approaches.

Keywords. Software Testing, Test-case Generation, Cuckoo Search, Lévy Flight, Tabu Search.

2010 Mathematics Subject Classification. 68N30.

1 Introduction

Software engineering is a well defined approach used from development of software to its retirement [11]. Software testing is a critical part of software development life cycle which aims to ensure software quality by detecting faults. A software development organization expends about 50% of the total project effort on testing [2]. The project effort includes various factors such as manpower, time and cost. Software testing includes the generation of test data which are inputs to the software to check whether it satisfies the requirement or not. Since, exhaustive testing is infeasible in practical situations, so the real challenging task is to automatically generate the optimized test data which minimizes the testing effort incurred without compromising on the quality of the system.

Automated test data generation approaches have been classified into two categories: random techniques and path oriented techniques [22]. Random techniques select random inputs as test data without concerning test requirements, so these
are not effective and may fail to generate desired test data. Path oriented techniques generate test cases for the paths identified using control flow information [3]. Static techniques and dynamic techniques are basically the classification of path oriented techniques [6]. These techniques are often based on symbolic execution and finding test data without executing the program under test, while dynamic techniques obtain the necessary test data by executing the program under test [7]. Thus, static techniques suffer various problems when they handle indefinite arrays, loops, pointer references and procedure calls [18]. Several other problems also exist in static techniques like resolution of computed storage locations or their computational cost [4]. As values of input variables are determined when programs execute, dynamic test data generation can avoid the problems confronted by static methods. Several approaches exist under dynamic techniques which treat a testing problem as a search or optimization problem [25]. These approaches are generally known as meta-heuristic approaches.

Genetic search [10], cuckoo search [33], simulated annealing [13] etc. are such meta-heuristic approaches used for automated and optimized generation of test data. Cuckoo search (CS) is based on breeding behavior of certain species of cuckoo and provides a more reliable and cost effective solution than other meta-heuristic techniques because it offers a fine balance of randomization and intensification and less control parameters [33]. This paper presents an automatic test-data generation technique based on cuckoo search via Lévy flight [33] and tabu search [7, 9]. A search by the cuckoo algorithm [14, 33] is driven by control dependencies [8] existing in the program under test. Lévy flight [5, 20] is used to solve the problem of getting stuck in local optima, thereby exploring the search space more efficiently. Tabu search (TS) is added to reduce the number of iterations and execution time of the algorithm, thus reducing the overall complexity [7]. In the proposed algorithm cuckoo search is used to generate the solutions at the end of every iteration while Lévy flight is used only when the cuckoo generations are stuck in local optima (encountered via tabu lists which keep a memory structure to store good and bad test cases along with their predicates/full paths at the end of each iteration). The experimental results show that the proposed hybrid model optimizes the effort required in software testing.

The rest of the paper is organized as follows. Section 2 provides the overview of previous related work. Section 3 explains the proposed algorithm based on cuckoo search and tabu search for automatic test data generation. Section 4 provides the experimental study of the proposed strategy. Section 5 presents the analysis of the proposed strategy against various other meta-heuristic techniques. Section 6 summarizes the paper with the conclusion and directions for future work.
2 Background

The use of meta-heuristic approaches is not naïve to the world of software testing and specially test data generation. Many different approaches and algorithms have been applied in this field [4, 6, 14, 24, 25, 29]. The meta-heuristic approach of genetic algorithm has been used extensively in software testing [24, 29]. A library of genetic algorithms for structural testing under branch coverage criterion was developed by Jones, Sthamer and Eyres [12]. It used fitness function based on predicates of the branch, having a capability of directing the test towards the fault-critical areas. A hybrid approach of genetic algorithm (GA) with tabu search was presented by Rathore et al. [25]. It uses genetic based approach to gradually intensify the search procedure with efficient mutation step involving tabu search to reduce the randomness and execution time of the search. Cuckoo search on the other hand also emerged as an efficient meta-heuristic approach in software testing. It is being used in the area of unconstrained optimization problems [30]. It is also used to implement multi-objective genetic algorithm in software testing [23]. A high degree of improvement with respect to the conventional genetic algorithm, in terms of iterations and node coverage in test data generation is shown while using cuckoo [14]. Cuckoo is better than various other meta-heuristic algorithms as it has less variables [33]. One disadvantage of a simple cuckoo search technique is its convergence to local optima. One possible strategy to minimize this process is by using Lévy flight which uses a random walk to move the solution space into new territories. Therefore cuckoo search along with Lévy flight has been merged with tabu search technique for test data generation to use the strength of both and generate test data in a minimal number of iterations. An overview of cuckoo search, Lévy flight, tabu search and control dependency graph is given in the following subsections.

2.1 Cuckoo Search

Cuckoo search is an optimization heuristic technique based on the brood parasitism of certain species of cuckoo [33]. These species lay their eggs in other host bird’s nests and remove other eggs to increase the hatching probability of their own eggs [21]. If host birds discover that the eggs are not their own, they will either throw these alien eggs away or simply abandon their nest and build a new nest elsewhere. Some female parasitic cuckoos are often specialized in the mimicry of color and pattern of the eggs of a few chosen host species. This reduces the probability of their eggs being abandoned and thus increases their reproductivity [21].
The cuckoo search algorithm as proposed in [33] works with the assumption that each cuckoo lays one egg at a time, and dumps its egg in a randomly chosen nest. The best nests with high quality of eggs will carry over to the next generation with the number of available host nests being fixed. There might be a case when the egg laid by the cuckoo is discovered by the host with a probability $P \in [0, 1]$. If we treat each egg in the nest as a solution and cuckoo egg as a new solution, then we can replace the not so good solutions with better cuckoo solutions.

This algorithm has been applied in several optimization problems, outperforming the results achieved with the well-known meta-heuristic algorithm. Yang and Deb [33] discovered that the performance of the CS can be improved by using Lévy flight instead of simple random walk as the random walk via Lévy flight is more efficient in exploring the search space as its step length is much longer in the long run [32].

### 2.2 Lévy Flight

Lévy flight is the typical characteristic of flight behavior demonstrated by many animals and insects. Generally, the foraging path of an animal is effectively a random walk because the next move is based on both the current location/state and the transition probability to the next location. The chosen direction implicitly depends on a probability, which can be modeled mathematically. So, Lévy flight essentially provides a random walk while the random step length is drawn from a Lévy distribution:

$$ \text{Lévy} \sim u = t^{-\lambda}, \quad 1 < \lambda \leq 3, $$

(1)

which has an infinite variance with an infinite mean. The distribution of step length in this is according to the probability distribution which is heavily tailed. The distance of the random walk from the origin tends to a stable distribution after a large number of steps [31].

The Lévy flight has been applied to diverse range of fields, describing animal foraging patterns, the distribution of human travel and even some aspects of earthquake behavior and light [1]. The behavior of Lévy flight showed promising results when applied to optimization and optimal search problems [27, 28].

### 2.3 Tabu Search

Tabu search (TS) is a memory based adaptive algorithm that guides local heuristic search procedure to explore solution space beyond local optima. Tabu search maintains a tabu list (memory structure) whose content keeps on changing as the search proceeds. This memory used to tabu certain number of solutions which are already explored, so that the search will not be repeated in these solutions. This
confines the search space to a smaller region thereby improving the search. Thus tabu search has properties of local search and climbing ability. Tabu search has been used in the work of [7] for structural software testing. In this, it uses two cost functions, memory and backtracking process. Tabu search starts just like an ordinary local search and traverses to another solution iteratively until a termination criterion is reached.

2.4 Control Dependency Graph (CDG)

Control dependency graphs [8] show the control flow information of a program under test. These are generated from the corresponding control flow graph. These graphs characterize how the predicates in a program govern the execution of other statements and predicates. They determine the statements that would execute after a given statement has been executed.

We can use cuckoo search to form a new test case at the end of every iteration. Tabu search is used to store good test cases as well as bad test cases along with their predicates/full path so that they are not generated redundantly and Lévy flight is used to get out of the local optima. Thus, the employment of all the above techniques ensures less memory wastage and faster convergence to the globally optimal solution.

The next section shows the detailed version of the proposed hybrid model along with the flow graph and pseudo code of the proposed algorithm.

3 Proposed Approach

A hybrid approach has been made to generate test cases for the program under test using a combination of cuckoo search and tabu search techniques. The aim is to generate test cases that cover all possible paths in the program. The cuckoo algorithm generates solutions which are guided towards the better than the ones in the nest; this can lead the algorithm into a local maximum. To overcome this weakness, a backtracking mechanism is performed to get the algorithm into the previous good solutions stored in the tabu lists.

Figure 1 describes the architecture of the proposed approach. The source program is converted into a CDG which is taken as the input to the algorithm. It shows the sequence of events that occurs in the proposed strategy from generating a random solution. The outputs that can be measured from the approach are shown as test data generated, nodes covered, number of iterations and execution time.

Concepts and terminologies. curr_Population: current set of eggs (solutions) in the host nest. It is partially problem dependent, for a problem with small number
of solutions it is not ideal to give the initial population a very large value as the required solutions are often covered in the first iteration itself (because of the low number of solutions). Also for a problem with a large number of solutions it is again not ideal to set the initial population to be a large value as it would result in a brute force attack (or getting struck in the local optima in the first iteration itself) and would not use the algorithm to the fullest potential. So the initial population size is set conveniently according to the problem.

pop_Size: Represent the number of solutions (eggs) currently in population.

frac_popSize: Fraction of the population size replaced by new random solutions.

max_Gen: It is a measure of stopping criterion.

Short term tabu list (STL): Used to store the worst solutions along with their predicates at the end of every iteration, so as to forbid some moves that are likely to drive us back to recently visited worst solution and hence are of no use. The size of STL is equal to the product of number of iterations and the value of frac_popSize (the number of solutions to be removed from the current population into the number of iterations).
**Long term tabu list (LTL):** Used to store local optimal solutions along with their full path for intensification of search procedure. This helps in moving the search towards other possibilities (diversification) intelligently, in order to find out global optimum solution. The size of the LTL depends upon the number of iterations to the cuckoo population and the type of solutions in the current population.

**threshold:** This represents a value at which Lévy flights are triggered. Generally threshold is set as half of pop_size. One way of getting convergence, i.e., local optima, is when all the solutions in the nest have the same fitness value. This means that at the end of every iteration a solution is added to the STL and another solution with the same fitness value is added to the LTL (or its counter incremented), this would nullify the presence of the tabu lists now we have to backtrack to get out of this local optimum. In order to prevent the above effect, the value of threshold is set to be half the value of the nest size for optimal performance.

The strategy for numeric input data is also proposed in the form of flow chart (Figure 2) and pseudo code is given in Algorithms 1 and 2 below.
3.1 Strategy for Numeric Input Data

1. Initialize variables like pop_Size, frac_popSize and max_Gen and data structure of solutions. Set threshold and keep short term tabu list (STL) and long term tabu list (LTL) empty.

2. Now generate pop_Size number of random solutions which will be treated as an initial population. The solutions in the current population are executed and the nodes covered by these solutions are marked as covered. These solutions are stored in the data structure of solutions which will contain the final set of test cases generated for the problem.

3. The uncovered nodes are chosen as target node on the basis of their height in CDG. The one with the lowest height will be chosen first.

4. Once the target node is selected, the fitness value for each solution in the current population is calculated with respect to the target node. The fitness value for any solution is calculated according to the fitness function defined. Here the concept of predicates comes into picture. The predicate of a solution is the path traversed by the solution until it reaches the same level as that of the target node. The fitness function \( F_i \) gives the fitness value of the solution \( (i) \). The fitness value is equal to the number of common branches between the predicate of the solution and the full path of the target node from the root.

In the CDG shown in Figure 3, if the target node is 3, then the predicates for node 3 are \{ET, 1T\}. If we have a solution which covers the path \{ET, 1F, 4T\} then its fitness value is considered to be 1 because only one of the paths matches in the predicate values of the target node and the corresponding solution.

5. For a particular target node a solution is generated randomly and its fitness value is calculated after an STL check. In STL we store the predicates of the worst solutions along with the solutions itself so as to prevent the similar kind of solutions for further processing. If the fitness value of the randomly generated solution is greater than a randomly selected solution from the current population, they are replaced; otherwise it is discarded. Whenever a new solution is added to the current population, target node or any uncovered node coverage is checked. If the target node is covered, the algorithm goes for another target node until all nodes of the CDG are covered. If any uncovered node other than the target node is covered, it is removed from the list of uncovered nodes, reducing the number of nodes to be taken as target for the next generation. If a solution is covering any of the node, either the target node or the uncovered node will be stored in the data structure of the solution.
6. In every iteration, a frac_popSize number of worst solutions is selected from the current population and replaced by randomly generating solutions which involve the STL check before entering the nest.

7. After this, a best solution (the one with highest fitness value) among the current population is selected and its predicates and its solution along with a counter are stored in the long tabu list (LTL).

8. The above algorithm loops in the same manner until the number of iterations reaches max_Gen or the target node is covered. In the former case the corresponding node will be marked as unreachable and in the latter the solution which is covering a target node will be stored in the list of final test cases. Otherwise, if the threshold of the counter for the best solution in LTL is reached or all the solutions in the current population have the identical fitness value, then this provides us the information that we are stuck in local optima and have to backtrack to get the solution to the problem. For example in Figure 3, if the threshold for the solution having the predicates \{ET, 1F\} is reached, it tells us that the solution has to be generated by backtracking because the desired predicates are \{ET, 1T\}. This backtracking mechanism is provided by Lévy flight, which generates a solution better than the current locally best solution. It is shown in the flow chart of Figure 4.

9. Lévy Flight. In Lévy flights, the proposed strategy tries to generate a better solution \(x_{i+1}\) from the local best solution \(x_i\) by setting an optimized value for the step size \(s\) and a problem dependent Lévy function \(f(x)\). The step size in equation (2) should be related to the scales of the problem of interests.

\[
  x_{i+1} = x_i + s \oplus f(x). \tag{2}
\]

The equation (2) is essentially the stochastic equation for random walk. A random walk is a kind of Markov chain where current location (first term in the above
Figure 4. Flow chart of backtracking with Lévy flight.

equation) and transition probability (the second term) determines the next status/location. The product \( \oplus \) means entry wise multiplications. In this case, we have taken transition probability as 1.

For every solution generated through Lévy flight, fitness value calculation is done. If fitness value, in other words, predicates of the new solution are the same as that of the current best solution but has a different full path (the path from the root to the leaf node), then this solution is stored for further processing. This is done when no better solution was generated for the current best solution.

Once a solution is generated via Lévy, the LTL and STL are reset and the solution is added to the current population and the algorithm goes on similarly. If a better solution is not generated via Lévy flight and the whole input range is over using the predefined step size, the particular node is marked as unreachable and the algorithm continues for other uncovered nodes as a target node.
3.2 Strategy for Character Input Data

The proposed strategy can be applied to the programs with character string input like string comparison etc. The major problem in this case is the number of variations possible in a particular input and the generation of string inputs. The approach taken to calculate the fitness of a solution will be the same as in numeric data. There can be different methods to calculate the predicates for string which can be general or problem dependent for further optimization.

The approach for string input data is problem dependent. For a given problem with string input, the fitness function calculation along with the proposed approach will be the same with an assumption that the length of the strings taken as inputs is predefined. The generation of string inputs via Lévy flight is not a trivial task and hence there is a change in the Lévy flight to generate the solution. The Lévy flight functionality will be problem dependent and can further be optimized for enhancing the performance. So till now the approach is same as for numeric input data. The major change will only be in Lévy flight.

String Input Generation Using Lévy Flight. The solution taken into Lévy, i.e., the current best solution in the population, is worked upon character by character. For example, for the problem statement of string comparison, where a one string can be smaller, greater or equal to the other (assuming the length of the strings to be predefined); only one of the strings has to be modified because the other will act as reference, i.e., for comparing each character with the mutated string.

1: for each character in first string do
2:    if the ASCII value of character is less than the one in second
3:       string then
4:       Increment its ASCII value by 2 instead of incrementing by 1 as it is
5:       quite tedious to get the desired solution in case when string is lengthy.
6:    end if
7:    if fitness value of the solution is better than the previous solution then
8:       Return the new cuckoo solution to the current population.
9:    end if
10:   if while incrementing the character, its ASCII value becomes greater
11:      than the ASCII value of the corresponding character in second string
12:      then
13:         Decrement its ASCII value by 1.
14:    else
15:       Repeat steps 5 to 9.
16: end if
17: end for
4 Experimental Study

For explaining the flow of the proposed approach, we have considered example programs such as the triangle classifier problem, roots of a quadratic equation and binary search. While the triangle classifier program and roots of a quadratic equation contain the if-else structures, the binary search contains a while loop and a break statement. All of the sample programs have been implemented using the JAVA language.

4.1 Triangle Classifier Problem

The triangle classifier problem is the most widely used test program for evaluating the results. It takes the three sides of the triangle as input denoted by \((a, b, c)\) and classifies them as either an equilateral triangle or an isosceles triangle or a scalene triangle. The other conditions such as invalid input range and not a triangle case are also handled in the sample program. The basic idea of the proposed approach is to apply the cuckoo search technique in generating the inputs to the triangle classifier program wherein only the better results are forwarded into the next iteration by removing the worse results from the population. The tabu search technique is used to store the removed worse solutions thus preventing the generation of further test cases of these types and also storing the best solutions from each iteration for backtracking purposes. Once the proposed strategy starts to generate a similar set of test cases and hence being pulled into the local optima, the Lévy flights functionality kicks in by generating solutions near the search space of the best solutions stored in the tabu lists.

The terminologies used in the case study is described below:

- \(\text{max\_Gen}\) is one of the stopping criterion used. For the triangle classifier program this value is set as 1000.
- \(\text{pop\_Size}\) is the number of solutions in the cuckoo nest. For the triangle classifier case study this value is set as 5.
- \(\text{Threshold}\) is the value which triggers the Lévy flight. For example, if \(\text{pop\_Size}\) is 5, threshold is 2, which means that 40% of the solutions in the current population are the best solution (having same predicates) and we have got stuck in local optima, hence Lévy flight needs to be triggered. In this example this value is set as 3.
- \(\text{StepSize}\) is a configured value, which determines the delta change needed to be performed on the good solutions stored in the tabu lists. For this example, this value is set as 1.

The range of input for the triangle classifier program is taken to be between 1 and 100.
Figure 5. CDG of triangle classifier.

The program under test is first converted into CDG format as shown in Figure 5 and represented as a two-dimensional matrix. This matrix is the input to the proposed strategy along with the configurable values stated before.

The values of the two tabu lists STL and LTL before the first iteration will be NULL, that is, it will not contain any solutions. The initial values of the cuckoo population which are randomly generated triplets are $[5, 9, 4]$, $[6, 2, 2]$, $[1, 7, 0]$, $[7, 10, 1]$, $[7, 6, 5]$. Since the value contained in STL is NULL, they are allowed into the cuckoo nest without a STL check. These solutions are executed with the help of the CDG matrix. From Figure 5 all the nodes are covered by the above generated solutions, except the nodes $[13]$ and $[17]$. The final set of test cases currently will be $\{[5, 9, 4], [1, 7, 0], [7, 6, 5]\}$.

**First Iteration.** The target node chosen is $[13]$ as its height is lower than that of node $[17]$ according to the CDG. The predicates of the target node calculated will be $\{ET, 3T, 9T\}$. The predicates of the solutions in the cuckoo nest currently with respect to the target node will be $\{ET, 3F, 11T\}$, $\{ET, 3F, 11T\}$, $\{ET, 3F, 11T\}$, $\{ET, 3T, 9F\}$. Fitness value of each solution is calculated for each
test case by comparing their predicates with that of the current target node. The calculated fitness values of the current population with respect to the target node are \{1, 1, 1, 1, 2\}. Now, a random cuckoo solution is generated which will be \{4, 1, 4\}. STL check will take place to check whether the randomly generated solution’s predicates are already present. Since STL is empty (NULL) at this moment, this solution will be allowed to proceed to the next step for further processing. Going by the cuckoo algorithm now, the fitness value of \{4, 1, 4\} is compared with that of a solution randomly selected from the cuckoo nest. The fitness value of \{4, 1, 4\} with respect to the current target node is 2 as \{ET, 3T\} matches those of the target node. The random cuckoo solution selected from nest would be \{6, 2, 2\} which has a fitness value of 1. The fitness value of \{6, 2, 2\} is less than that of \{4, 1, 4\} which will replace \{6, 2, 2\} in the cuckoo nest. The program under test is executed with the help of CDG for the solution \{4, 1, 4\} which covers the uncovered node \{17\}. This solution is added to the list of final test cases since a new node has been covered with this solution. The final test cases currently are \{5, 9, 4\}, \{1, 7, 0\}, \{7, 6, 5\}, \{4, 1, 4\}. The worst solution from the population \{1, 7, 0\} with the predicates \{ET, 3F, 11F\} and having the least fitness value of 1 is selected randomly of the other solutions having the same fitness value and is stored in the STL along with its predicates. To replace the removed solution from the nest, a new solution is generated randomly which is \{2, 14, 0\}. The STL check does not allow the solution \{2, 14, 0\} to enter the cuckoo nest and hence a new solution has to be created. On generating a solution \{7, 6, 5\}, the STL check allows this to enter the cuckoo population. Of all the solutions in the current cuckoo nest, the best solution so far is selected. The solution \{7, 6, 5\} is the current best solution which is added to the LTL along with its predicates and the counter corresponding to these predicates in LTL is incremented by 1. End of first iteration.

Second Iteration. During the start of the second iteration, the solutions in the cuckoo nest are \{5, 9, 4\}, \{4, 1, 4\}, \{7, 6, 5\}, \{7, 10, 1\}, \{7, 6, 5\}, the values stored in STL are \{1, 7, 0\} and its predicates \{ET, 3F, 11F\}, the LTL contains the solution \{7, 6, 5\} along with its predicates \{ET, 3T, 9F\} and its counter value as 1. As in the previous iteration, solutions which are generated randomly must pass the STL check before any further processing can be done on them. One such solution which passes the STL check is \{7, 1, 7\} with predicate values \{ET, 3T, 9F\}. The fitness value of this solution is 2 and this fitness value is compared with the fitness value of \{7, 6, 5\} which is a solution selected randomly from the cuckoo nest. Since the fitness value of \{7, 1, 7\} is not greater than the solution \{7, 6, 5\}, the former is discarded. The next step in the algorithm is to update the values of STL and LTL by the worst solutions and the best solutions in the nest which are \{7, 10, 1\} with
predicates \([\text{ET}, 3\text{F}, 11\text{T}]\) and \([\{7, 6, 5\}]\) with predicates \([\text{ET}, 3\text{T}, 9\text{F}]\) respectively. Since the predicates of the best solution in this iteration are already present in the LTL, only its counter value is incremented so that it becomes 2 now. \textit{End of second iteration.}

**Third Iteration.** During the start of the third iteration, the solutions in the cuckoo nest are \([\{5, 9, 4\}, \{4, 1, 4\}, \{7, 6, 5\}, \{5, 5, 7\}, \{7, 6, 5\}]\) with STL containing \([\{1, 7, 0\}, \{7, 10, 1\}]\) and its predicates. The LTL contains the solution \([\{7, 6, 5\}]\) and its predicates with its counter value set as 2. A solution which passes the STL check is \([\{8, 5, 8\}]\) with fitness value of 2. This is compared with the fitness value of a random solution picked from the nest which is \([\{4, 1, 4\}]\). Since the fitness value of \([\{8, 5, 8\}]\) is not greater than that of \([\{4, 1, 4\}]\), it is discarded. One feature of the cuckoo search noticed here is that even though both the solutions compared are of the same type, no overhead is taken in switching the solutions. The STL is updated with the solution \([\{5, 5, 7\}]\) which is selected randomly from all the solutions as all of them have a fitness value of 2. Its full path and the solution itself is now added to STL. A new solution \([\{19, 17, 20\}]\) replaces the removed solution. Since the best solution is still \([\{7, 6, 5\}]\), its counter value is incremented in LTL which is now 3. At this time the counter value of the LTL for the best solution has crossed the threshold value Lévy flights functionality is triggered. \textit{End of third iteration.}

**Lévy Flights.** The full path of the solution \([\{7, 6, 5\}]\) in LTL is calculated as \([\text{ET}, 3\text{T}, 9\text{F}, 14\text{F}]\) which is the path traversed by the solution in the CDG. Now the first value of the triplet is selected and incremented by the step size to generate a new solution \([\{8, 6, 5\}]\). Since its fitness is not greater than \([\{7, 6, 5\}]\) nor does it have a different full path than \([\text{ET}, 3\text{T}, 9\text{F}, 14\text{F}]\), it is discarded. The first value is again incremented till 100 (maximum of input range) till a good solution is obtained. In this case all the solutions from \([\{7, 6, 5\}]\) till \([\{100, 6, 5\}]\) do not generate a better solution with a better fitness value and a different full path. Now the second value in the triplet is selected and incremented by the step size to generate a better solution. During one of its iterations, a solution \([\{7, 7, 5\}]\) is generated which contains the same fitness value but different full path. This solution is stored for further use. After exhausting all the three values there is still no solution generated which will cover the current target node. Now the solution \([\{7, 7, 5\}]\) which was stored earlier is treated to the Lévy flights functionality. This generates a solution \([\{7, 7, 7\}]\) during the iteration of the third value. This solution has a fitness value greater than the solution \([\{7, 7, 5\}]\) on which the Lévy flights are being applied. This solution is returned back to the cuckoo nest by replacing the original solution \([\{7, 6, 5\}]\) after flushing the tabu lists.
The program under test is executed for the new solution which covers node \(13\) which is an equilateral triangle. Hence the solution \([7, 7, 7]\) is added to the list of final test cases. Since all the nodes are covered, the algorithm stops.

The data generated by the proposed strategy are the number of iterations used for covering all nodes. The list of nodes covered (if any when the number of iterations reaches the max_Gen value) and the list of final test cases needed to cover the nodes of the program under test. The algorithm also outputs the number of test data generated. The list of final test cases for the current experimental study is \([5, 9, 4], [1, 7, 0], [7, 6, 5], [4, 1, 4], [7, 7, 7]\). *End of program.*

The result of applying the proposed approach to triangle problem for different input ranges and different parameter values is shown in Tables 1–4.

It can be inferred from the above results that unique test data generated are nearly constant for different population sizes. For example, for a population of size 5, it hovers around 27 for different input ranges. In Figures 6 and 7, the above results are shown in the form of bar graphs as well.

The bar graph in Figure 6 shows that as the population size increases, the number of iterations decreases (input range being constant).

The bar graph in Figure 7 shows that the execution time partially depends upon the number of iterations and the input range. The deviation in execution time is due to the uncontrolled STL iterations. The execution time also depends upon the system configuration, operating system and language used. The above results have been computed over M380 2.53GHz dual core processor on Windows 7 64-bit operating system using Java as a programming language on Eclipse Indigo IDE.
<table>
<thead>
<tr>
<th>Input range</th>
<th>Mean of test data</th>
<th>Mean of execution time (ms)</th>
<th>Mean of total iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>26.4</td>
<td>51.1</td>
<td>413.8</td>
</tr>
<tr>
<td>200</td>
<td>27.7</td>
<td>90.7</td>
<td>922.5</td>
</tr>
<tr>
<td>500</td>
<td>27</td>
<td>227.5</td>
<td>2350.8</td>
</tr>
<tr>
<td>1000</td>
<td>27.7</td>
<td>209.5</td>
<td>5159</td>
</tr>
<tr>
<td>5000</td>
<td>26.8</td>
<td>11688.2</td>
<td>28552.3</td>
</tr>
</tbody>
</table>

Table 1. Pa = 1, PopulationSize = 5, StepSize = 1, LTLThresholdValue = 3.

<table>
<thead>
<tr>
<th>Input range</th>
<th>Mean of test data</th>
<th>Mean of execution time (ms)</th>
<th>Mean of total iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>39.7</td>
<td>52.2</td>
<td>640.8</td>
</tr>
<tr>
<td>200</td>
<td>37.3</td>
<td>61.1</td>
<td>1115.2</td>
</tr>
<tr>
<td>500</td>
<td>36.1</td>
<td>90.2</td>
<td>2677.3</td>
</tr>
<tr>
<td>1000</td>
<td>36.6</td>
<td>133.7</td>
<td>5992.1</td>
</tr>
<tr>
<td>5000</td>
<td>34</td>
<td>3670.2</td>
<td>23578.1</td>
</tr>
</tbody>
</table>

Table 2. Pa = 1, PopulationSize = 10, StepSize = 1, LTLThresholdValue = 3.

<table>
<thead>
<tr>
<th>Input range</th>
<th>Mean of test data</th>
<th>Mean of execution time (ms)</th>
<th>Mean of total iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>39.5</td>
<td>50.2</td>
<td>245.6</td>
</tr>
<tr>
<td>200</td>
<td>41</td>
<td>55.6</td>
<td>765.3</td>
</tr>
<tr>
<td>500</td>
<td>44.4</td>
<td>94.6</td>
<td>2838.4</td>
</tr>
<tr>
<td>1000</td>
<td>42</td>
<td>3021</td>
<td>4766.7</td>
</tr>
<tr>
<td>5000</td>
<td>38.9</td>
<td>22928.9</td>
<td>21576.7</td>
</tr>
</tbody>
</table>

Table 3. Pa = 1, PopulationSize = 10, StepSize = 1, LTLThresholdValue = 6.

<table>
<thead>
<tr>
<th>Input range</th>
<th>Mean of test data</th>
<th>Mean of execution time (ms)</th>
<th>Mean of total iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>56.2</td>
<td>46</td>
<td>231.7</td>
</tr>
<tr>
<td>200</td>
<td>59.36</td>
<td>64.9</td>
<td>744.5</td>
</tr>
<tr>
<td>500</td>
<td>65.1</td>
<td>98.1</td>
<td>2276.6</td>
</tr>
<tr>
<td>1000</td>
<td>60.5</td>
<td>148.8</td>
<td>4482.3</td>
</tr>
<tr>
<td>5000</td>
<td>58.3</td>
<td>7159.8</td>
<td>21071.6</td>
</tr>
</tbody>
</table>

Table 4. Pa = 1, PopulationSize = 20, StepSize = 1, LTLThresholdValue = 11.
The above results (Tables 1–4) and analysis (Figures 6–7) show that the approach can be applied to a specific problem for different values of the variables used and an optimized value for the parameters can be found to be used in future to generate test data.

4.2 Roots of a Quadratic Equation

For the roots of a quadratic equation, the variables used will be the same as the one used in the triangle classifier problem. It takes as input a triplet of positive integers \((a, b, c)\) in the range of 0 to 100. The output may have one of the following: Not a quadratic equation, real roots, imaginary roots, equal roots. It is assumed that the value of \(a\) will not be equal to zero during this process.

\[
ax^2 + bx + c = 0. \tag{3}
\]

The CDG for the above problem is shown in Figure 8.

Initially, \(\text{STL} = \{\text{NULL}\}\) and \(\text{LTL} = \{\text{NULL}\}\). The generated population is \{[10, 108, 8], [12, 4, 4], [0, 7, 1], [7, 10, 1], [7, 6, 5]\}. After the execution of the initial population, the node uncovered is \(\{13\}\). The final test cases are \{[10, 108, 8], [12, 4, 4], [0, 7, 1], [7, 10, 1]\}.

In the first iteration, the target node chosen will be 13, and the predicates of the target node are \([1T, 3T, 9T]\). Fitness values of the solutions will be calculated based on comparing their predicates with that of the target node. The predicates of the solutions in the population with respect to target node are \{[1T, 3F, 11F],...\}.
[1T, 3T, 9F], [1T, 3F, 11T], [1T, 3T, 9F], [1T, 3T, 9F] and their fitness values being \{1, 2, 1, 2, 2\} respectively. A randomly generated solution \([4, 1, 4]\) is first checked in the STL to check if the predicates of the solution already exist. Since this is the first iteration, the STL will be NULL and hence it allows the algorithm to proceed with the successive step. Predicates of the random solution \([4, 1, 4]\) are \([1T, 3T, 9F]\) with a fitness value of 2. A cuckoo solution is randomly selected from the nest \([12, 4, 4]\), and since the fitness value of the solution \([4, 1, 4]\) is equal to that of \([12, 4, 4]\), the solution \([4, 1, 4]\) is ignored and the algorithm proceeds to the next step. Now one of the worst solutions from the nest is picked and placed in the STL. Here, the solutions \([10, 108, 8]\) and \([0, 7, 1]\) both have the least fitness values of all the solutions in the nest. \([0, 7, 1]\) is selected randomly and placed in the STL with a new solution \([1, 8, 4]\) replacing the removed solution after passing the STL check. Of all the solutions present in the nest the best solutions have a fitness value of 2, one among the best solutions \([1, 8, 4]\) is selected randomly, the LTL is updated with its value along with its predicate \([1T, 3T, 9F]\) and the counter value for this predicate value is incremented by one.
The second iteration also happens in a similar fashion to the first one with the STL being updated with the value of \([10, 108, 8]\) as it is the one with the least fitness value. Since the remaining solutions have a fitness value of 2, and all of them have the same predicate values the counter present in the current LTL list for this predicate value is incremented by one. At the end of the second iteration, the cuckoo solutions in the nest are \{\([10, 18, 7]\), \([12, 4, 4]\), \([1, 8, 4]\), \([7, 10, 1]\), \([7, 6, 5]\)\}.

The third iteration starts off with the population in the nests as shown above. A randomly generated solution \([14, 25, 9]\) with the predicates \([1T, 3T, 9F]\) and fitness value 2 is checked with a solution picked up randomly from the nest \([7, 6, 5]\). Since the fitness values of both the solutions are equal to 2, the solution \([14, 25, 9]\) is ignored.

The worst solution among the current population \([10, 18, 7]\) is updated in the STL and a new solution replaces the removed one. Now since the solutions in the nest all have a fitness value of 2 with the predicate values \([1T, 3T, 9F]\), its corresponding counter in LTL is incremented by one. At this point the counter value crosses the LTL threshold (threshold) value set. At this time the Lévy flights functionality is triggered. The full path of the solution in LTL \([1, 8, 4]\) is \([1T, 3T, 9F, 14T]\). Lévy flights generate the solution \([4, 8, 4]\) as proposed in the algorithm and its full path being \([1T, 3T, 9T]\) is different from the one in the LTL list. This solution is returned. The solution \([4, 8, 4]\) is checked to see if it covers our target node or any other uncovered node. Since the solution \([4, 8, 4]\) covers the node 13, which was our target node, the algorithm proceeds to see if any other uncovered nodes are remaining. Since all the nodes are covered, the algorithm ends with the solution \([4, 8, 4]\) being added to the list of final test cases.

The minimal number of test cases required traversing all nodes is

\([\{10, 108, 8\}, \{12, 4, 4\}, \{0, 7, 1\}, \{7, 10, 1\}, \{4, 8, 4\}]\).

The result of applying the above approach to the roots of the quadratic equation problem is shown in Table 5.

<table>
<thead>
<tr>
<th>Input range</th>
<th>Mean of test data</th>
<th>Mean of execution time (ms)</th>
<th>Mean of total iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>29.4</td>
<td>49.1</td>
<td>423.1</td>
</tr>
<tr>
<td>200</td>
<td>31.2</td>
<td>92.5</td>
<td>945.1</td>
</tr>
<tr>
<td>500</td>
<td>33.4</td>
<td>240.5</td>
<td>2225.3</td>
</tr>
<tr>
<td>1000</td>
<td>29.7</td>
<td>267.9</td>
<td>5250.2</td>
</tr>
<tr>
<td>5000</td>
<td>28.3</td>
<td>11487.2</td>
<td>28667.3</td>
</tr>
</tbody>
</table>

Table 5. \(Pa = 1\), \(PopulationSize = 5\), \(StepSize = 1\), \(LTLThresholdValue = 3\).
4.3 Binary Search

Binary search also works in the same way as described above. Since binary search is a relatively smaller program compared to the triangle classifier problem, we reduce the size of the population to see the proper working of the proposed algorithm. The CDG for the while loop is created with the assumption that once the control enters the while loop, it is considered as a separate region where this region is represented by a different CDG. After the end of the while construct, the loop is re-iterated back to the starting node of the while loop until the condition becomes false. Once the loop exits, the region is connected to the next statement executed immediately after the loop. The CDG for binary search will be represented as the same as explained in the previous example where the while region is considered to be a part of the bigger CDG of the program and hence avoiding changes to the CDG representation. The binary search has three outputs: search key found, search key not found and invalid input range. In a sparse distribution of an input array to the binary search, one of the outputs such as the search key not found or invalid input range can be covered by the initial population and a very few successive iterations. The search key found scenario can be obtained by the Lévy flights functionality by taking the best solution from the LTL and incrementing it with the step size to get the value of the key.

The proposed approach has also been applied on some real world problems which include various industry standard projects and software involving different styles of programming and using different types of programming constructs like switch case, function calls, continue and input types [19]. Experimental results obtained so far match with the expected results. The next section shows the comparison of the proposed approach with the existing work in the form of tables, line graphs and bar graphs.

5 Analysis

A genetic based approach of test case generation is used by Pargas, Harrold and Peck [19]. Here a random set of test cases is generated to form initial population. The program under test is executed with these test cases, and the uncovered nodes are noted. Out of these nodes one of them is made target node which has to be reached in subsequent iterations. This process is continued until all nodes are covered. Fitness value calculation is based on current target node in CDG. As the current target node keeps changing the fitness value calculation is dynamic.

A hybrid approach by Rathore et al. [25] combines some features of genetic and tabu for an optimal test case generation. This approach uses the genetic based approach to gradually intensify the search procedure. Genetic algorithm generates
new test data from previously generated good candidates. It guides the direction of search by evaluating candidate test data and using control-dependence graph (CDG) of the program. The tabu search is added to the mutation step of genetic algorithm to reduce the randomness and execution time of the search.

An approach using basic cuckoo search and Lévy flights with tabu search has been proposed by Krish et al. [14]. The reason behind the proposed approach outperforming the one in [14] is the presence of STL in the proposed approach which reduces the search space by avoiding the solutions that are already rejected and hence not counting them in the final iteration whereas the solution in [14] does not use this mechanism. Also the proposed approach uses the concept of LTL which triggers the Lévy flights. Since the LTL stores the best solutions, the search space again is reduced by performing a search in the neighborhood of the best solution which yields the optimum results.

The experimental results shown above for the triangle classifier are compared with existing approaches of test data generation like GA, GA and tabu, cuckoo and tabu in Table 6. The factors taken into account for comparison are the number of iterations and the number of nodes uncovered. It is to be noted that the proposed algorithm is more efficient in terms of the number of iterations and that the nodes are also covered. For instance, input range of 0 to 100, the number of iterations is 467, while in [14] it is 24233, in [19] it is 65535 and in [25] it is 765. The proposed approach also covers all the nodes in the given problem while equilateral triangle condition is not covered in [19] for the same input range. The most recent and similar kind of work done in [14] has not customized the generation of a new solution using Lévy flight which has been covered in the proposed approach. Moreover [14] has not restricted the entry of bad solutions in the population using the tabu lists, which has added great strength to the performance of the presented work.
Figure 9 shows a graph of the comparisons made on the number of iterations between the proposed approach and other approaches in Table 6 using the similar techniques. Simple GA based approach [19] uses 65,535 iterations for an input range of 0 to 100. The GA and tabu based approach [25] uses 765 iterations where as the simple cuckoo and tabu approach [14] uses 24,233 iterations for the same range of input. The proposed approach uses only 467 iterations. The efficiency of the proposed approach can be observed by the less number of iterations taken to cover all the nodes of the triangle classifier program for the input range of 0 to 100.

Ant colony optimization is another meta-heuristic technique to automate the generation of test data in software testing. Automation of test data is not only generating the test data, it is more important that the data generated must be unique.
Table 7. Observations.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple problem</td>
<td>Test data generated</td>
</tr>
<tr>
<td>(Range = 100)</td>
<td>$\sim 5 \times$ minimum number of test cases possible</td>
</tr>
<tr>
<td>Execution time</td>
<td>$\sim 20$ ms</td>
</tr>
<tr>
<td>No. of iteration</td>
<td>$&lt; 100$</td>
</tr>
<tr>
<td>Average complex problem</td>
<td>Test data generated</td>
</tr>
<tr>
<td>(Range = 100)</td>
<td>$\sim 5 \times$ minimum number of test cases possible</td>
</tr>
<tr>
<td>Execution time</td>
<td>$\sim 50$ ms</td>
</tr>
<tr>
<td>No. of iteration</td>
<td>400–750</td>
</tr>
<tr>
<td>Complex problem</td>
<td></td>
</tr>
<tr>
<td>Expected similar behavior</td>
<td></td>
</tr>
</tbody>
</table>

and optimized. Figure 10 shows a comparison in terms of test data generated between the proposed approach, ant colony optimization which uses pheromone evaporation factor of 0.1, 0.3 and 0.5 with ant colony size varying between 5 and 1000 over 23 iterations (ACO) [15], UML communication diagram [26] and the genetic algorithm [16].

From the above analysis it can be noted that the time taken by the proposed approach is better than the approaches using GA [19, 25] as the number of iterations taken is less than 65,535 in the case of simple GA and 765 in a hybrid GA approach, but the proposed algorithm uses only 467 iterations for an input range of 0 to 100 for solving the triangle classifier problem. Also the proposed algorithm fares better in the number of test data generated as compared to other approaches like the ACO in [15].

Table 7 describes the parameters considered for analysis of the proposed approach. It categorizes problems into different complexities namely simple, average and complex. Simple problems include programs such as binary search, addition of two numbers, etc. Similarly, the average complexity problems include the triangle classifier, roots of a quadratic equation. Complex problems include industry standard projects and software. The proposed approach has been evaluated against three parameters (number of iterations, test data generated and execution time) in all the mentioned categories. The observations reveal very promising results for the simple as well as average complex problems while the complex problems have also been explored and a similar kind of behavior is expected.

For better understanding the proposed approach, the pseudo code of the program is given below – divided into the main Cuckoo search algorithm (Algorithm 1) and the Lévy flights (Algorithm 2).
Algorithm 1 Main Algorithm

1: Set the values for populationSize, threshold, stepSize, inputRange
2: Initialize population
3: Check the nodes covered by these solutions by executing each and every solution in the population
4: Save the solutions which cover the uncovered nodes in the final set to test cases
5: The nodes which are not covered after executing the initial population are the candidates for the target node
6: Initialize max. generation
7: for each uncovered node (choose the one at lower level) do
8:    Find the fitness function of the population for the selected target node
9:    Set the value of the generation counter as zero
10:   Empty long tabu list(LTL) and short tabu list(STL)
11:   while generationCounter is less than maxGeneration do
12:     if Long tabu list is not empty then
13:        if Counter for any particular solution in long tabu list
14:           has reached its threshold value then
15:           LévySolution = Lévy flight(LTL, stepSize)
16:           if LévySolution is not null then
17:              New CuckooSolution is solution via Lévy flight
18:           else
19:              The target node is unreachable
20:              Break
21:        end if
22:     else
23:        new CuckooSolution is randomly generated whose predicate
24: end if
25:     else
26:        new CuckooSolution is randomly generated whose predicate not in STL
27: end if
28: if target node is reached then
29:    break
30: end if
31: Find FitnessValue(CuckooSolution)
32: Choose a hostSolution randomly from host nest
Algorithm 1 Main Algorithm (continued)

32: if FitnessValue(CuckooSolution) > FitnessValue(hostSolution) then
33:    Replace that CuckooSolution with hostSolution
34: else
35:    Discard that CuckooSolution
36: end if
37: Choose a worst hostSolution and remove it from the current population
38: Add the worst hostSolution predicates into STL
39: Choose a random solution until its predicate not in STL
40: if any uncovered node is covered then
41:    Mark it as covered
42:    Add this solution to the list of final test cases
43: end if
44: if target node is reached then
45:    break
46: end if
47: Find the best hostSolution for the current population
48: if its predicate value is already present in LTL then
49:    increment the counter for that FitnessValue
50: else
51:    Store its predicates according to the fitness value and solution in LTL and increment the counter for that fitness value
52: end if
53: if all the solutions in the current population are of same fitness value then
54:    Make the counter for the particular fitness value in the long term tabu list to its threshold
55: end if
56: Increment generationCounter
57: end while
58: if generationCounter is equal to maxGeneration then
59:    Mark the target node as unreachable
60: end if
61: end for
Algorithm 2 Lévy Flight (LTL, stepsize)

1: Take the complete data structure from the LTL corresponding to the FitnessValue whose threshold has been reached
2: Take the first value in the solution
3: while the first value has not reached the maximum input range do
4: Increment this value by StepSize
5: if the fitness value of this new solution is greater than the fitness value of the original solution then
6: Return this solution back to the main algorithm
7: else if the fitness value of the new solution is equal to the original solution and it has a different full path than the original solution then
8: Add this solution to the queue for further processing
9: end if
10: end while

6 Conclusion and Future Work

This paper presents a heuristic method for automation of test data generation using cuckoo search along with Lévy flight and tabu search. This approach uses cuckoo algorithm for selection and generation of candidate solutions guided by CDG to obtain path coverage criterion. Diversification in the search is achieved using backtrack operation with the help of tabu lists which explores new horizons of the search space and Lévy flight is used to prevent the solution to being stuck in local optima.

The results from the experimental study as explained above clearly support the fact that the proposed algorithm performs better than the other meta-heuristic approaches such as GA, ACO, ABCs, etc. in terms of node coverage, number of iterations and number of parameters. Thus test data can be generated efficiently and optimally. However, by using efficient data structures for tabu lists and Lévy flights, the time complexity can be reduced in the proposed approach for high value range integers (16, 32 bits). Our experiments conducted so far are based on relatively small examples and more research needs to be conducted with larger commercial examples which will further refine the proposed approach. Though the strategy for string input is laid out, the application of the proposed approach for string input is yet to be explored through experimental comparisons so that it could become more generic and robust as compared to other meta-heuristic algorithms.
Bibliography


Received March 22, 2012.

**Author information**

Praveen Ranjan Srivastava, Department of Computer Science and Information System, Birla Institute of Technology and Science, Pilani, Rajasthan 333031, India. E-mail: praveenrsrivastava@gmail.com

Rahul Khandelwal, Department of Computer Science and Information System, Birla Institute of Technology and Science, Pilani, Rajasthan 333031, India. E-mail: 1nmiitrahul@gmail.com

Shobhit Khandelwal, Department of Computer Science and Information System, Birla Institute of Technology and Science, Pilani, Rajasthan 333031, India. E-mail: shobhit.khandelwal.87@gmail.com

Sanjay Kumar, Department of Computer Science and Information System, Birla Institute of Technology and Science, Pilani, Rajasthan 333031, India. E-mail: er.sanjaykr1989@gmail.com

Suhas Santebennur Ranganatha, Department of Computer Science and Information System, Birla Institute of Technology and Science, Pilani, Rajasthan 333031, India. E-mail: suhassr@gmail.com