JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 35, 245-273 (2019)
DOI: 10.6688/JISE.201903_35(2).0001

# A Buffered Genetic Algorithm for Automated Branch Coverage in Software Testing

T. MANIKUMAR[1,+] AND A. JOHN SANJEEV KUMAR[2]
[1]*Department of Computer Applications*
*RVS College of Engineering*
*Dindigul, Tamilnadu, 624005 India*
*E-mail: stmanickumar@gmail.com*
[2]*Department of Computer Applications*
*Thiagarajar College of Engineering*
*Madurai, Tamilnadu, 625015 India*
*E-mail: ajscse@tce.edu*

Each and every software product has to be tested for assess its quality, which is time-consuming if it is performed manually. Moreover, it is difficult to generate all possible data for finite testing set. Search based Software Testing (SBST) are used to resolve this issue by utilizing metaheuristic algorithms to automate the test data generation. Hence, an efficient test data set could be generated with minimum cost. Among many metaheuristic algorithms, Genetic Algorithm (GA) is widely used for test data generation. This research work implements GA for generating test data to execute all the branches in a program. In the literature, existing approaches for test data generation using genetic algorithms are starts with random test data and find the optimum test data for a targeted branch. Then the entire GA process will be repeated to find the test data for the next target branch and it continues for all the target branches. In this a paper, a novel GA approach with a small buffer space is proposed for automated test data generation for branch coverage. When GA is searching test data for a particular target branch heuristically, it may reach the other target branches, if so happen, then those test data will get stored into the buffer space hence it is not necessary to run GA to cover that branch. Thus the Buffered Genetic Algorithm (BGA) approach outperforms the other GA based automated test data generation approaches in terms of number of iterations and search effectiveness. The proposed approach employs control flow graph to traverse and predicate the branch coverage. Seven benchmark programs are instrumented to evaluate performance of the proposed BGA based approach.

***Keywords:*** software testing, automatic test data generation, genetic algorithm, control flow graphs, buffered genetic algorithm

## 1. INTRODUCTION

Identifying test cases to evaluate a software product is an expensive process, which typically take half of the cost estimated for software development [12]. With the assist from automation tools the software products can be tested efficiently while reducing the time taken for writing test cases and also reduces the cost consumed by the manual test process. From the last two-decades, there are various approaches has been reported in the literature for automated test data generation such as random, path-oriented, goal-oriented and search-based approaches. Though these methods were reported with good results,

Received October 7, 2017; revised November 26 & December 18, 2017; accepted August 22, 2018.
Communicated by Shyi-Ming Chen.
[+] Corresponding author.

there are some limitations which indicate the further scope of the research in automated test data generation for software testing.

- Random generators often unsuccessful as the test data created here doesn't based on any objective function [50, 65, 70, 83, 87].
- Path-oriented generators have to identify the feasible path and then started constructing the test data set, often the paths are infeasible [23, 44, 67].
- Goal-oriented approaches [48, 49, 86] have proved their superiority then the random and path oriented generators. However, these methods are application specific rather than generic automation process.
- With the realization that the process of software test data generation can be cast into a search problem, recently, Search-Based Software Testing (SBST) approaches are performing better than any other techniques [6, 35, 36, 58].

In general, the SBST approaches starts with a random set of test data, then they are tuned or pruned to build an optimum test data set while using the objective function like branch, statement and path coverage criterions [96]. This paper focuses on branch coverage criterion. [1, 58] presented a comprehensive study on search based software testing techniques using metaheuristic techniques. However, the metaheuristic algorithm based test data generation approaches are over-performing the other methods, there some common limitations as well:

- One major issue is that the metaheuristic algorithms could be trapped with local minimum.
- For example, while generating test data for branch coverage using Genetic Algorithm (GA), the population may not contain any test data to reach the target node [27, 60].
- It has been reported in the literature that the percentage of coverage highly depends on nature of the program.
- Moreover, the performance of the metaheuristic techniques is highly influenced by their parameter settings.

Here, the proposed method tries to address most of the common issues reported in the literature as summarized above. This paper focuses on the use of Genetic Algorithms as a search method for automated branch coverage, in a technique referred to as Evolutionary Structural Testing. In order to find a test data that executes a branch, the goal of the search is to find an input vector that takes a path which is driven down the branch of interest. The space of candidate solutions in which the search operates is the input domain of the function under test. In general, GA based test data generation algorithms for branch coverage, executes genetic algorithm independently for every target branch, hence it takes '*n*' cycles to generate test data to cover all the '*n*' branches. In this paper, a novel Genetic Algorithm with additional buffer space is proposed for automated test data generation. While searching test data for a branch, the generated data may end up with other target branch which is yet to be tested. In case of the proposed Buffered Genetic Algorithm (BGA), those test data are buffered and the future targeted branch is marked as tested to save the GA time from exploring the branch again.

The rest of the paper is organized as follows: the following section presents a brief

review on search based software testing. Section 3 discusses the backgrounds related to automated test data generation using genetic algorithm. Section 4 explains the proposed GA for automated test data generation for branch coverage. Section 5 illustrates the experimental setup, the results are quantified and the performance of the proposed method is analyzed in Section 6. Section 7 concludes the paper with the contribution of the proposed system toward search based software testing along with future directions.

## 2. RELATED WORKS

Automating the process of software testing reduces the development cost effectively. SBST approaches outperform the other techniques [58, 65]. In general SBST methods start with random test data set, and they are evaluated based on fitness function to find how close they were to reach the target branch. The test criterions such as branch coverage or statement coverage are used as fitness functions. Based on the fitness measure the test data will be modified according to the metaheuristic algorithms so that they can achieve the coverage. This section summarizes a comprehensive study on search based software testing methods.

One common approach is random testing, where the test data sets are randomly generated and tested simultaneously to check the whether the testing criterion is satisfied or not. This kind of approaches is suitable for any structured programs and for any type of inputs. However, due to the lack of evaluation measure, this approach is expensive in terms of time and space, and the probability of finding the coverage is low [50, 70]. [73] uses GA to guide the search process. [31, 32] proposed a constraint based approach, where all the branching statements are translated into constraints and a logic programming approach is used to find the test data set. [89] developed an evolutionary based approach for structural testing and showed that this approach has better performance than the existing methods. [60] demonstrated a path-oriented approach to find the coverage for branch criterion programs. [22] proposed a search-based testing method to consider pointers and dynamic data structures, which also works for non-pointer input variables.

Metaheuristic techniques such as simulated annealing [85], Tabu Search [25], genetic algorithms [46, 64, 73], particle swarm optimization [93], quantum particle swarm optimization [2], scatter search [16], ant colony optimization [52], memetic algorithms [8], clonal selection algorithm [24], parallel cat swarm optimization [76, 77] and immune genetic algorithm [53, 82] have been applied to the problem of automated test data generation and provide evidence of their successful application. Among these, many of the article has been published for branch coverage [17, 21, 34, 45, 88, 89]. For a basic metaheuristic solution for branch coverage, initially the source code is instrumented to trace the program flow and find the distance between the current statements to the target statement. The distance is measured by cost functions. The objective of the metaheuristic technique is to minimize this cost so that the test data will reach the target statement. So, the input values are modified according to the search-based approach to minimize the cost. Hence, the test data set is constructed for each target nodes and cumulated to present the final test data set. In this way, the metaheuristic based approaches are capable of generating test data set for any data type.

Among the various metaheuristic approaches, Genetic Algorithm (GA) based approaches are more widely used for search-based test data generation [94]. [85] has implemented evolutionary search to find the coverage for precondition and postcondition statements. GA based test data generation is demonstrated for real-time software for both functional and structural criteria [37, 58, 65, 89].

[94] introduced the approach of GA based test data generation. At first the test data were generated by random generators then GA is used to find the test data for uncovered branches. For that the user has to interact with the system to choose a particular path or branch to be covered, where the branch distance is estimated as fitness function. [74] experimented dynamic and static testing with GA based test data generator and concluded that dynamic testing was not effective while static was not practical.

[4] proposed a test method to explore all possible test data inputs using GA. Similar idea was implemented by [45], here the Control Flow Graph (CFG) is constructed to trace the data flow, and the graph is traversed for each test data to find the coverage by following Breadth First Search (BFS) traversal approach. [64] proposed GADGET (Genetic Algorithm Data Generation Tool), to instrument the program automatically and generate the test data set using GA. However, this method can work only for scalar inputs. [85] proposed a test criteria based on both functional as well as non-functional properties for evolutionary based test data generator approach. Here the exceptions are considered as testing criterion. The experimental results show that optimization based approach's efficiency for test data generation.

[73] extended the work of [46] by introducing control dependency graph to measure the branch distance rather than by using CFG. The result indicates that this approach outperforms the approaches proposed by [46, 64]. In addition to that this approach can automate the test data for both branch as well as path coverage. [54] further extended this work by introducing weighted hamming distance. However, this method performed better only for path coverage criterion. [19] proposed a search based test data generator for path coverage, where they start with classifying the feasible and infeasible paths and then different fitness functions are equipped to generate the test data. [91] developed GA based test data generator by using normalized branch distance added with approximation level as fitness function. This approach is suitable for finding the coverage for local branches, and no description about multiple target coverage [7, 36, 58, 59]. [25] proposed an approach based on Tabu search with the [49] chaining approach. [13] introduced a novel fitness function based on the organisms' fossil record, and the quantified result indicates that this fitness function is unable to produce consistent performance for the benchmark program.

[40] used GA with multiple set of populations for path coverage. The experiments show that this approach is able to achieve better path coverage in terms of exploration and convergence rage. [84] focused on unit testing while using GA for test data generation. Here the chromosomes are encoded in such a way to represent the input vectors as well as objective functions too. The experiments are carried out with standard Java libraries. [13] proposed a technique for long sequence testing. The test cases are divided into small intervals and the test data generators are repeatedly executed to find the coverage. This method is more suitable for testing in large volume, however, the results are analyzed only for very few programs. [66] shows the efficiency of GA based test generation using program dependencies. [95] demonstrated that GA based automated test data

generation is more suitable for both small and large scale programs. [18] proposed an Immune GA as a hybrid evolutionary algorithm, and its performance is analyzed with non-standard programs. [21] proposed two fitness function based on hamming distance and branch distance and implemented with GA based test generation towards path coverage. The simulation results indicate that this method can converge faster than standard GA based test data generation.

[20] developed a GA based test data generator for path coverage, where the fitness function is evaluated as closeness of execution path and target path with overlapping sub path. The experimental results show that the proposed fitness function can achieve better coverage ratio than fitness function based on single specific path. [63] experimented the scalability of test data generators and reported that GA outperforms random generators in terms of number of branches and statements, is increased. [39] reported that the parallel execution of fitness estimation may enhance test data generation performance as programs become large with increased number of branches and search space. [41] demonstrated the issues like composition of fitness function for path coverage and parameter tuning for GA based test data generation. [69] proposed a faster GA based approach by generating more suitable test cases for path testing, and the performance is compared with [21] approach. [59] in his paper also focuses primarily on the design of fitness functions. [71] compared the performance of messy-GA and random search based test data generation and reported that messy-GA outperforms with better coverage. [15] discussed many nature inspired algorithms towards search based software testing process. [43] presents a survey of study on different techniques of test case generation.

In the overall GA is most widely used in test data generation, however, one common limitation is that the algorithm has to be repeated for each target branch. In this paper, the GA based test generator is extended to cover at least more than one branch at every iteration, hence the run time is half way reduced.

## 3. BACKGROUND

Test data generation in white-box testing (source code-based testing) is a process of finding program input on which a selected element (*e.g.* a not yet covered statement) is executed. Finding such input test data manually can be very labor intensive and expensive. Therefore, using metaheuristic algorithms to generate test data is efficient [29, 47, 50, 61, 66]. In this section, an overview of the search-based test data generation and computation of the branch cost functions for the coverage of individual structural targets are presented. However, some basic concepts are introduced first.

### 3.1 Search Based Software Testing

Search-based optimization methods, such as genetic algorithms, ant colony optimization and simulated annealing have successfully been applied in solving a wide range of software testing problems [3, 28, 36, 38, 55, 58, 79]. [92] combines genetic algorithms' characteristics and evolution strategies, using simulated evolution as the model of a search method, employing operations inspired by genetics and natural selection. In essence, the problem of generating test data reduces to the well-understood problem of

function minimization. In general, evolutionary algorithms generate a set of candidate solutions referred as population and iteratively recombine them to find the optimum candidate. At the end of every iteration the candidates are evaluated based on their fitness value and passed for the next generation only if they have better fitness value. In case of test data generation, the fitness functions are to be minimized to find how closer the candidate test data reaches the target node, hence, zero indicates that the algorithm found the test data to cover the target node. There are two categories of software testing: stating and dynamic testing. In static testing, the programs are analyzed without actually executing them, whereas in dynamic testing, the source code is instrumented and executed to find the coverage. This paper follows the dynamic testing approach, thus the target reachability of the test data could be estimated and updated heuristically to reach the target node.

Functional and structural testing are two major methods in the dynamic testing technique. In case of Functional (black-box) testing, it is not necessary to explore the internal structure and behavior of the source code. Instead, the test cases are to be identified for which the source code doesn't producing the correct output. In case of structural (white-box) testing, the logic, control flow and data dependency of the source code has to be tested. This paper proposes an algorithm for structural testing. There are two approaches of structural testing: control flow and data flow. Control flow testing is based on the control graph of the program and considers how to select the testing path for discovering more errors. On the other hand, data flow testing derives test data by considering how data are defined and how the data are used in a program. However, path testing is one of the white-box testing techniques. Via test cases, one intends to execute all possible paths of control flow through the program, then possibly the program can be tested completely [80, 81].

### 3.2 Control Flow Graph

Fig. 1 demonstrates the proposed framework for automated test data generation for branch coverage. At first, the source code is scanned to construct the instrumented code and the Control Flow Graph (CFG).

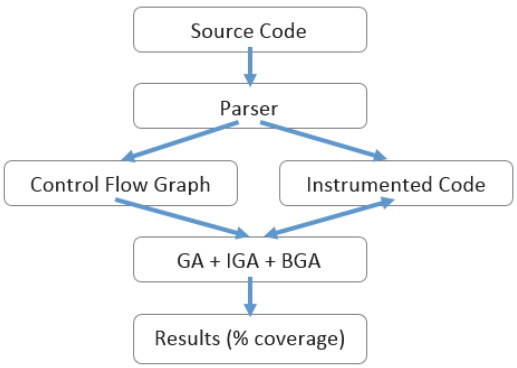


Fig. 1. Proposed framework for automated test data generation.

A CFG is a directed graph, $G = (N, E, s, e)$, where $N$ is a set of nodes, $E$ is a set of edges, with one entry node ($s$) and an exit node ($e$) respectively. Each node in the graph $n \in N$ corresponds to a programming statement. Each edge $e \in E$, represents control flow of the program from one statement (node) to another, $e = (n_i, n_j)$. The nodes can be either normal data manipulation or conditional statements. In case conditional or decision statements, it will have two paths: one exists when the condition is true, the other path is for the false case, both are known as branches. The condition for taking either true or false branch is called as branch predicate. For example consider the following code as shown in Fig. 2, which receives three sides of the rectangle as input and classify the type of the rectangle as it return 1, 2, 3, or 4 stands for scalene, isosceles, equilateral or not-a-triangle respectively [65]. Fig. 3 shows the corresponding CFG for the code, where Node-1 represents the first branching predicate, for this condition Node-2 will be executed for the true predicate and the control will continue with Node-3, for the false predicate the control will skip Node-2 and directly move on to Node-3. In case of Node-10, Node-12 and Node-13 are the true and false predicates respectively. Here the target nodes are 2, 12, 13, 14, 16 and 19.

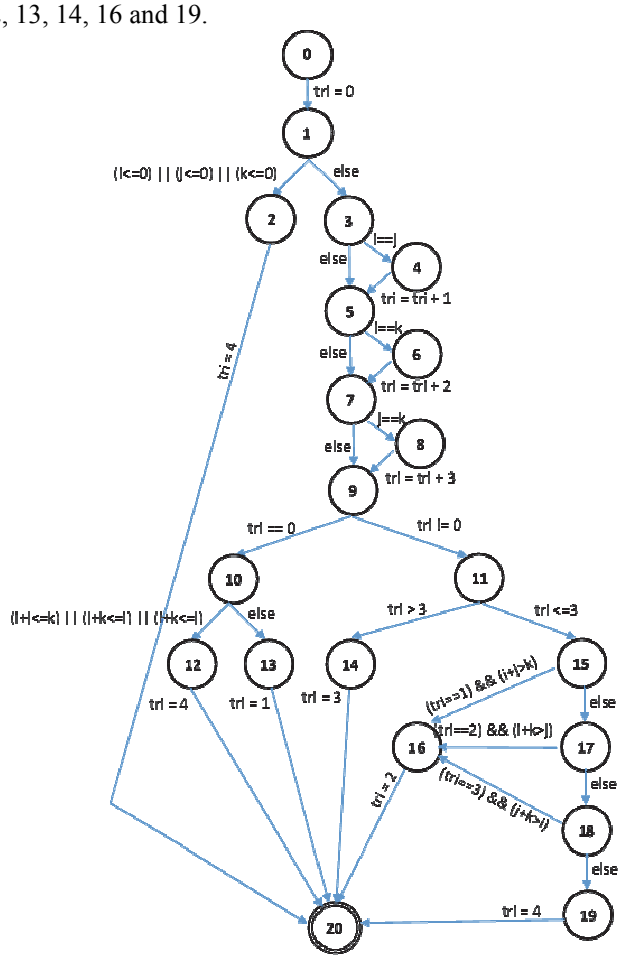


Fig. 3. Control flow graph for Michaels' triangle classification problem.

```
function funMichaelTriangle(i, j, k)

% Michaels' Triangle Classification Problem

   tri = 0;                                          % Node 0
   if ((i <= 0) || (j <= 0) || (k <= 0))             % Node 1
      tri = 4;                                       % Node 2
   end
   if (i == j)                                       % Node 3
      tri = tri + 1;                                 % Node 4
         end
         if (i == k)                                 % Node 5
                  tri = tri + 2;                     % Node 6
         end
         if (j == k)                                 % Node 7
            tri = tri + 3;                           % Node 8
         end

         if (tri == 0)                               % Node 9
                  if ((i+j<=k) || (j+k<=i) || (i+k<=j))  % Node 10
               tri = 4;                              % Node 12
            else
            tri = 1;                                 % Node 13
            end
         else
                  if (tri > 3)                       % Node 11
               tri = 3;                              % Node 14
            elseif ((tri == 1) && (i+j > k))         % Node 15
               tri = 2;                              % Node 16
            elseif ((tri == 2) && (i+k > j))         % Node 17
               tri = 2;                              % Node 16
            elseif ((tri == 3) && (j+k > i))         % Node 18
               tri = 2;                              % Node 16
            else
               tri = 4;                              % Node 19
            end
         end
         disp(tri)                                   % Node 20
```

Fig. 2. Michaels' triangle classification program.

For every sample test data, the program will be executed with the instrumented code and the corresponding path in CFG is traced to test whether the test data is able to reach the specified target. The path (*bp*) visited by the test data is saved as sequence of nodes

$bp = (n_1, n_2, \ldots, n_m)$ such that for each $i$, where $1 \leq i \leq m$, $(n_i, n_{i+1}) \in E$.

For every branching condition (branch node) $n_i$, there will be two exists like $n_j$ and $n_k$, and there are control dependent on $n_i$. From the above figure, Node-12 is control dependent on Node-10, and Node-10 is control dependent on Node-9. Node-12 is not directly control dependent on Node-9, however, Node-12 is transitively control dependent on Node-9. For structured programs, control dependence shows the nesting structure of the program.

### 3.3 Branch Ordering

After the construction of CFG, it is important to choose by which order the nodes in CFG are going to be visited. There are four types of branch ordering methods are reported in the literature such as: Breadth first strategy (BFS), Depth first strategy (DFS), Path prefix strategy (PPS), Random strategy (RNS). The BFS starts exploring the nodes level by level, DFS visits the nodes in depth wise, the random strategy traverse the graph in random, and the PPS traverse the graph with predefined paths [72]. To achieve branch coverage, the objective is to find an input vector that satisfies all the condition through the path and ends at the target node. The test data often fails to reach the target node because of infeasible paths. To resolve this issue, Prather & Mills [67] suggested the use of an adaptive strategy in which one new test path, or sub-path is added at a time and previous paths serve as a guide for selection of subsequent paths using some inductive strategy, called as Path Prefix Strategy (PPS). This paper follows the PPS as [72] reported that this outperforms other branch ordering strategies. Fig. 4 illustrates PPS based branch ordering.

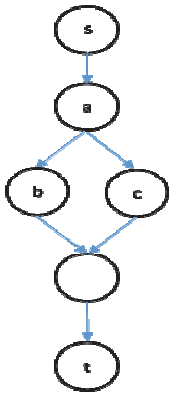


Fig. 4. Path prefix strategy.

For a path $bp_1$ that is traversed in an execution, a reversible prefix $bp_2$ is defined as the minimal initial portion of path $bp_1$ to a target node '$t$', whose branches are not yet covered, and the branch that is covered by $bp_1$. For example, in Fig. 4, if path traversed is $bp_1$: ($s$, …, $a$, $b$, … $t$), branch ($a$, $c$) is not covered and ($s$, …, $a$) is the minimal initial portion of $bp_1$ that satisfies the condition above, then ($s$, …, $a$, $b$) is a reversible prefix. Accordingly, branch ($a$, $c$) is a candidate for selection for coverage. If branch ($a$, $c$) is selection for coverage, then the path ($s$, …, $a$, $c$) is said to be the reversal of path ($s$, …, $a$,

*b*). In the path prefix strategy, at any stage, if ($n_1$, …, $n_{k-1}$) are the traversed paths, the idea is to find an input $x_k$ to cause the reversal of shortest reversible prefix, with reversal, amongst all the nodes in the path. In this case prefix $bp_2$ identifies the next branch to be considered for coverage.

### 3.4 Genetic Algorithm

Genetic algorithm (GA) [30] is one of the most popular evolutionary-based algorithms. It has been successfully applied to numerous problems both at the level of structural and parametric optimization [9], and to software testing, for example [10, 58, 90]. GA is a search method utilizing the principles of natural selection and genetics [42]. Concisely, GA operates on a set of candidate solutions, called a population, to a given problem. The candidate solutions are evaluated based on their ability to solve the problem. The results of the evaluation are used in a process of forming a new set of solutions. The choice of individuals that are passed to the next population is performed in a process called selection. This process is based on 'goodness' of candidate solutions. Additionally, genetic operators, *i.e.* crossover and mutation, are employed to modify selected candidates. Such sequence of actions is repeated until some final criterion is fulfilled. GA based test data generation process starts with random population of test set chromosomes. Each chromosome of test data is evaluated for their branch distance, then the selection operator is applied to choose most feasible parents from the set, with them the replacement procedure such as crossover and mutation operators are applied to generate new set of population. The next iteration of GA is started with this newest population, once again the genetic operator are applied with this newest population to regenerate the next set of chromosomes. At every iteration the termination condition is checked to stop the GA procedure. The reachability from every test input to the target branch node is measured with two metrics called approach level and the normalized branch distance as discussed in [89]. This distance measure is used as fitness function for the GA procedure to be minimized [11, 48, 60, 72].The following figure summarizes the classical GA procedure used to automate the test data generation for branch coverage.

---

**Input:** Instrumented source code
**Output:** Set of test data for the target nodes.
**Initialization:** GA parameters, TestData (TD) array
for each target node
  Initial population of real values (P) ← Generate random population of test data
  while (not termination condition) do
    Calculate the fitness value (branch distance) for each test data, $f$(P)
    if (target is reached)
      Update TD with the chromosome which reaches the target node
      Continue with the next branch
    end
    C ← { }  // Initialize children population
    Sort the chromosomes in descending order based on their fitness value
    while |C| < |P| do

```
        Select a pair of parents for matting (using any of the selection methods)
        Mate the parents to create children c1 and c2
        Perform crossover and mutation on children
        C ← C ∪ {c1, c2}
      end
    end          // Next generation
end              // Next Target
```

Fig. 5. Genetic algorithm procedure.

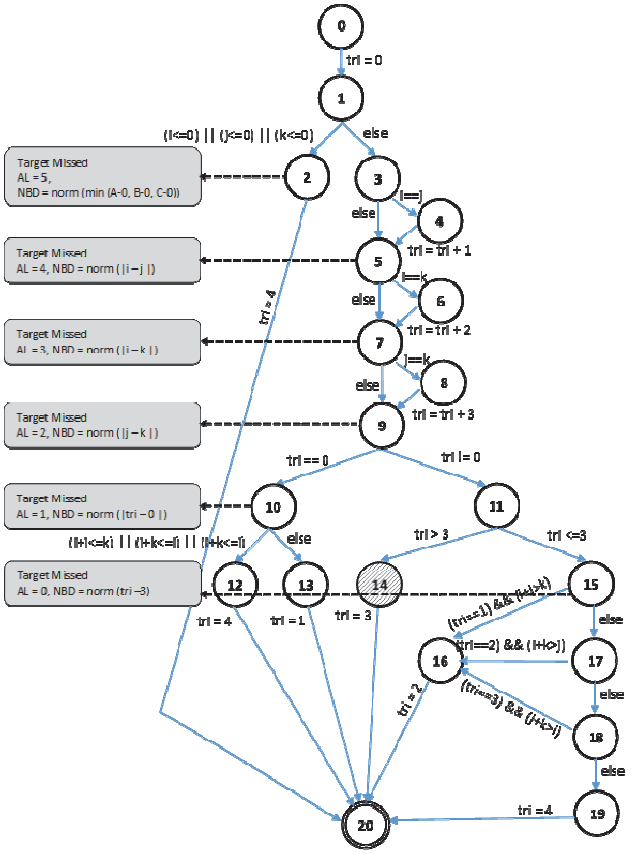


Fig. 6. Computation of branch distance.

Fig. 6 demonstrates an example for computing the fitness function for a test data. Node-17 (shaded node in the figure) is considered as the target node here, the following are the possible situations and their corresponding fitness values:

- If the condition ((*i*<=0)||(*j*<=0)||(*k*<=0)) satisfies at Node-1, the target node is 5 level away from it, hence the approach level is 5,
- If the condition (*i*==*j*) at Node-3 fails, then the approach level is 4,
- Suppose, if the condition (*i*==*k*) at Node-5 is failed then the AL is 3,

- If the condition (*j*==*k*) at Node-7 is failed then the AL is 2,
- If the condition at (*tri*==0) at Node-9 is failed then the AL is 1,
- And, if the condition (*tri*>3) at Node-11 is failed then the AL is 0, *i.e.* the target will never be reached.

For all the above situations, the branch distances are calculated as discussed in [72]. The fitness function will result 0 when all the aforementioned conditions are satisfied, that is the termination condition for GA.

Generating the test data inputs with minimum time is always an open problem in software testing towards cost reduction. Though GA-based SBST approach with path-prefix strategy significantly improves the performance of automated the test data generation [72], still it traps into local minima. [57] proposed an Incremental Genetic Algorithm (IGA) to further improve the GA's test data generation performance. The IGA based test-data generation has two-phases and works as follows:

- In the first phase, a classical GA starts with a random set of test data and finds the feasible data for each branch node of CFG independently.
- In the second phase, a classical GA starts by making use of the feasible parents from the previous phase as initial population and then proceed to automate the test data generation process to cover all the targets.

This incremental version of GA is followed in this research along with a novel modification to extend its performance.

## 4. BUFFERED GENETIC ALGORITHM (BGA)

From the previous work [57], it is shown that IGA outperforms GA for automated test data for branch coverage. [65] makes use of coverage table to trace the conditions to be satisfied to reach a target branch. In common GA has to be executed for each target branch for finding the test data. For each target, at every iteration of GA, the coverage table is updated to guide the search. The authors reported that sometime the test data might satisfy a condition which is not required to be true for the current target branch, in such case, those data are stored as they can be used in future. This approach influence to propose the Buffered Genetic Algorithm (BGA). Here, while GA searches for a particular target branch, the test data (chromosome) might end up with another target which is to be explored further. In general, GA based software test data generation would take more iterations as if the chromosomes (test data) are unable to reach the target branch means they are ending with other target branches. In such situation, it is proposed to add a buffer space to GA, so that, those non-specific target data could be saved in that space, in order to indicate GA not to generate test for that branch again. Moreover, this approach could be adapted to any search-based software testing approaches. Hence, while searching test data for a target branch may result in minimum of 2 to 3 branches, which in turn, reduces the computation time.

The proposed Buffered Genetic Algorithm (BGA) is implemented in Phase 2, *i.e.* while executing the Incremental Genetic Algorithm (IGA). IGA starts with the initial

population which is derived from the first phase. The proposed enhancement adds an extra buffer space to IGA, where a coverage array $B_t$ is used to trace the coverage of each target '$t$' and an array of TestData (TD) to store the collection of test data for all the branches. The number of cells in that coverage array are equal to the number of targets in the program. Initially, all the cells are assigned with the value 0 to represent that no test data has been generated for any of the target branch, whenever a test data is found for a branch (might be a target or non-target) '$i$' its corresponding coverage flag is set to 1.

$B_t = 0$, for $t = 1, \ldots, n$

And for the array of the test data set, it is initialized with a null set, and whenever a test data is found for a branch the array is updated with the test data along with the target branch (node) number at the first column of the array.

TD = [ ]

Then, the IGA starts with the first target branch. While GA searches the test data for the first target branch, the current population may contain a chromosome to reach another target branch. In such case, the chromosome of test data will be stored in a buffer space, and the corresponding target branch flag is set in coverage array. For example, while searching the test for the 1st target branch, if the population contains a test data to reach 4th target branch, then the test data store into a buffer area, and the corresponding cell in the coverage table is set to 1.

$B_4 = 1$, and TD = {4, chromosome of test data}

Once the target branch is reached, then the next iteration of GA starts by choosing the next uncovered target from the coverage array, among the uncovered list of branches, the next target will be chosen based on branch metrics as discussed in section 3.3. The GA process will be continued till it finds the test data for all the target branches. Fig. 7 summarizes the proposed BGA algorithm for automated test data generation for branch criterion.

---

**Input:** Instrumented source code
**Output:** Set of test data for the target nodes.
**Initialization:** GA parameters, TestData (TD) array, Coverage Array (B)
for each target node
  Initial population of real values (P) ← Feasible Parents List from Phase-1
  while (not termination condition) do
    Calculate the fitness value (branch distance) for each test data, $f$(P)
    if (target is reached)
      Update TD array with the chromosome which reaches the target node
      Update the coverage array B
      Continue with the next branch (based on branch metrics)
    end
    Sort the chromosomes in descending order based on their fitness value

```
        C ← { }     // Initialize children population
        while |C| < |P| do
            Select a pair of parents for matting (using any of the selection methods)
            Mate the parents to create children c1 and c2
            Perform crossover and mutation on children
            C ← C ∪ {c1, c2}
        end
    end         // Next generation
end             // Next Target
```

Fig. 7. The proposed BGA for automated test data generation.

Buffered GA – based test data generation could cover more than one target branch at a time. Table 1 shows a sample number of generations taken for a triangle classifier problem, the boldfaced and underlined labels are the primary targets for the corresponding run. At first, classical GA is executed to find the optimum test data for each branch nodes. Here we have 10 branch nodes (Node-1, 3, 5, 7, 9, 10, 11, 15, 17, and 18), hence GA will result in 10 optimal set of chromosomes. Then they are used as initial population for IGA and it is executed for every target node. As shown in Fig. 3, here we have five target nodes: Node-13 for Scalene (SCA), Node-14 for Equilateral (EQU), and Node-16 for Isosceles triangle (there are three possible reach for isosceles, which are from the nodes-15, 17, and 18, they are labelled as IS1, IS2, and IS3. as a result it took 160 fitness evaluation *i.e.*, 16 populations to generate the test data for all the target branches. It is noticed that the first run of IGA towards the Target-1 (EQU), unexpectedly reaches the targets IS1 and IS3, however those samples are avoided as the current run is focused only on EQU target. This is where the proposed BGA approach makes the difference. In BGA approach,

- Trial-1 completes by single run, where the primary target is EQU, but while trying the reach that target, the other chromosomes in the population produces the test data for the other target nodes too. So the execution stops at first run itself, which in turn takes 80 fitness evaluation, *i.e.*, 8 populations, which is 0.5 times faster than the IGA execution.
- In Trial-2 of BGA, first run takes 1-population to reach the primary target EQU, which also generated data for IS3. As there are remaining targets to be covered, the second run continues, which reaches the next target IS1 with 3-populations. Then the third run started for target IS2 and reaches by 6-populations which also generated the data for the final target SCA. In total, this trail of BGA took 10-populations to find the test data for all five targets it is 0.37 times faster than IGA.
- Trial-3, first run reaches the EQU target node with 2-populations, also covers the targets IS1, IS2 and I3, the second run covers the one remaining target SCA in 2-populations. Totally, in this trail, BGA covers all the branch targets within 4-populaitons, which is 0.75 times faster than IGA.

These sample results encouraged apply the proposed BGA approach to other benchmark programs and the results are quantified in the following section.

**Table 1. Sample test data generation outputs from IGA and BGA.**

| Approach | Runs | Output of Test data | #Test Data |
|---|---|---|---|
| IGA | Target-1 | NoT, NoT, NoT, IS3, IS1, IS3, NoT, NoT, NoT, NoT, NoT, IS1, EQU, NoT, NoT, NoT, IS3, **EQU**, NoT, IS3 | 20 |
| | Target-2 | NoT, NoT, **IS1**, EQU, NoT, NoT, NoT, IS3, EQU, IS3 | 10 |
| | Target-3 | NoT, NoT, NoT, NoT, NoT, IS1, EQU, NoT, NoT, NoT, IS3, EQU, NoT, IS3, IS1, EQU, EQU, IS3, IS3, NoT, NoT, NoT, NoT, NoT, NoT, NoT, IS3, IS3, IS1, EQU, EQU, IS3, IS3, IS3, IS3, IS3, IS3, IS3, EQU, NoT, IS3, IS3, IS1, EQU, EQU, EQU, IS3, IS3, IS3, IS3, IS3, IS3, NoT, IS1, EQU, EQU, IS1, EQU, IS1, IS1, EQU, IS1, SCA, IS3, IS1, IS1, IS1, EQU, EQU, EQU, IS1, EQU, IS1, IS1, EQU, IS1, IS1, IS1, **IS2**, IS1 | 80 |
| | Target-4 | NoT, NoT, NoT, NoT, NoT, IS1, IS1, EQU, EQU, IS1, EQU, NoT, NoT, NoT, **IS3**, EQU, NoT, IS3, EQU, EQU | 20 |
| | Target-5 | NoT, NoT, NoT, NoT, NoT, IS1, EQU, NoT, NoT, NoT, IS3, EQU, NoT, IS3, IS3, IS3, IS1, EQU, EQU, NoT, NoT, NoT, NoT, EQU, NoT, EQU, IS3, IS3, **SCA**, SCA | 30 |
| | | Number of Test data Sets | 160 |
| BGA Trial-1 | Run-1 | NoT, NoT, NoT, NoT, NoT, **IS1**, IS1, **IS3**, **IS2**, IS3, NoT, IS3, IS1, IS1, IS2, IS3, IS3, IS3, IS3, NoT, **SCA**, IS2, SCA, IS1, IS1, IS1, IS2, IS2, IS1, IS3, NoT, NoT, NoT, IS3, IS1, IS2, IS1, IS1, IS1, IS1, IS1, IS2, IS2, IS1, IS1, IS1, IS1, SCA, SCA, IS1, IS1, IS1, IS1, IS2, IS2, IS1, IS1, IS1, IS1, IS2, IS1, IS1, IS1, IS1, IS1, IS1, IS1, IS1, IS1, IS2, IS2, IS1, IS1, IS1, IS1, IS2, IS2, **EQU**, SCA | 80 |
| | | Number of Test data Sets | 80 |
| BGA Trail-2 | Run-1 | NoT, NoT, NoT, NoT, NoT, **EQU**, **IS3**, NoT, IS3, EQU | 10 |
| | Run-2 | NoT, NoT, NoT, EQU, IS3, NoT, IS3, EQU, EQU, NoT, NoT, EQU, IS3, IS3, NoT, NoT, NoT, IS3, IS3, EQU, EQU, EQU, EQU, IS3, IS3, IS3, **IS1**, EQU, EQU, IS1 | 30 |
| | Run-3 | NoT, NoT, NoT, NoT, NoT, NoT, EQU, IS3, NoT, IS3, EQU, EQU, EQU, IS3, IS3, NoT, NoT, NoT, IS3, IS3, IS1, EQU, EQU, EQU, IS1, EQU, IS3, IS3, IS3, EQU, IS1, EQU, EQU, EQU, EQU, IS1, EQU, EQU, IS1, EQU, EQU, EQU, IS1, EQU, EQU, IS1, EQU, EQU, EQU, EQU, EQU, IS1, EQU, EQU, IS1, EQU, EQU, IS1, IS1, IS1, EQU, EQU, IS1, EQU, EQU, IS1, EQU, EQU, **SCA**, SCA, **IS2** | 60 |
| | | Number of Test data Sets | 100 |
| BGA Trail-3 | Run-1 | NoT, NoT, NoT, NoT, **IS1**, **IS3**, IS3, IS1, NoT, NoT, NoT, **IS2**, IS3, IS3, IS1, IS2, IS3, NoT, **EQU**, EQU | 20 |
| | Run-2 | NoT, NoT, NoT, IS1, IS3, EQU, IS3, IS2, IS3, IS3, IS3, IS3, IS3, IS3, IS1, EQU, IS1, IS3, IS3, **SCA** | 20 |
| | | Number of Test data Sets | 40 |

## 5. EXPERIMENTAL SETUP

The proposed automated test data generation using Incremental Genetic Algorithm (IGA) is experimented and the results are observed in this section. The performance is analyzed with eight benchmark programs found in the literature and compared with the recently proposed algorithms in the same domain. The benchmark programs are: Line in Rectangle [26], Number of Days [26], CalDay [5], Complex Branch [89], and four trian-

gle classification programs [65, 68, 78, 89]. The following table summarizes the parameters and their values used for the evaluation of proposed GA based automated test data generation.

Each program is tested with 10 different population sizes, and the Tournament Selection, arithmetic crossover and uniform mutation are the type of genetic operators used. The crossover and mutation operations are performed with 0.8 and 0.01 probability respectively. And a maximum limit of $10^4$ iterations are set for each GA procedure. The proposed framework is compared with the Incremental-GA [57] and another GA based approach [72].

## 6. RESULTS & DISCUSSIONS

**Complex Branch –** The average performance on number of generations and percentage coverage for the Complex Branch benchmark program is depicted in Figs. 8 and 9 respectively. For this program the proposed BGA algorithm is able to construct the test data set with the minimum of 12 generations when the population size is 100 and achieves 99.78% of coverage which is a one step ahead performance than IGA based approach. Table 2 justifies the superior performance of BGA algorithm in terms of better F scores.

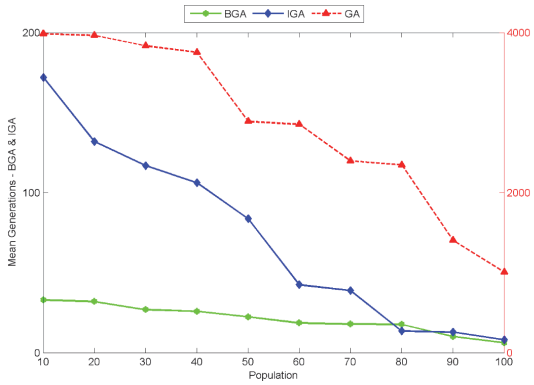


Fig. 8. Performance analysis BGA, IGA& GA for complex branch program with mean generations.

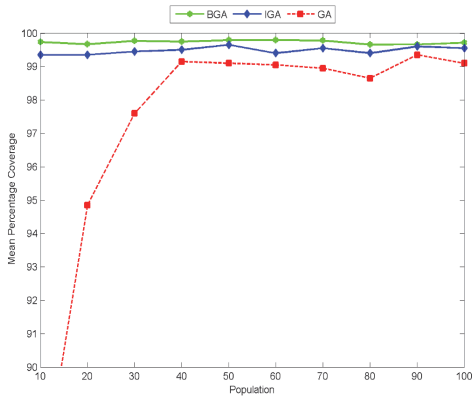


Fig. 9. Performance analysis BGA, IGA & GA for complex branch program with mean coverage.

**Table 2. ANOVA test evaluation of BGA, IGA & GA for complex branch program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| CB | BGA | F Value | 19.04 | 8.94 | 5.55 | 4.37 | 4.14 | 3.57 | 4.73 | 3.17 | 2.48 | 2.29 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0003 | 0.0007 | 0.0007 | 0.0021 | 0.0016 |
| | IGA | F Value | 27.09 | 11.94 | 6.85 | 5.08 | 4.74 | 3.89 | 5.62 | 3.29 | 2.25 | 2.42 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0005 | 0.0014 | 0.0014 | 0.0041 | 0.0032 |
| | GA | F Value | 62.21 | 28.54 | 17.24 | 13.31 | 12.56 | 10.67 | 14.51 | 9.32 | 7.02 | 7.39 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0010 | 0.0030 | 0.0030 | 0.0090 | 0.0070 |

**Calendar Day –** The performance comparison is illustrated in Figs. 10 and 11 for the CalDay (CD) program. For the greater population size the performance of BGA and IGA are similar in terms of mean number of generations and the mean percentage of coverage, but BGA wins as it stays ahead in for all the population sizes. Here the BGA takes less than 3 generations to complete the test data generation process which is predominantly higher, then the IGA and GA too. The F-scores also reflect the same in Table 3.

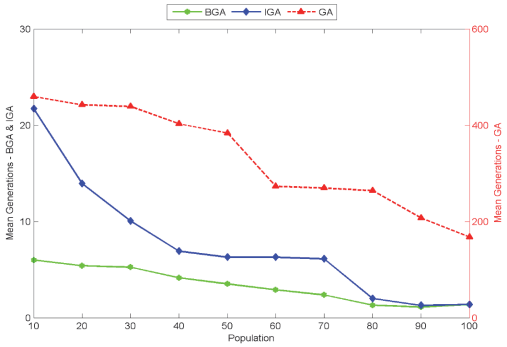


Fig. 10. Performance analysis BGA, IGA & GA for CalDay program with mean generations.

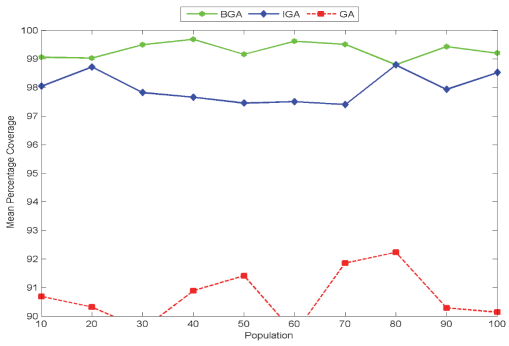


Fig. 11. Performance analysis BGA, IGA & GA for CalDay program with mean coverage.

**Table 3. ANOVA test evaluation of BGA, IGA & GA for CalDay program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| CD | BGA | F Value | 3.49 | 2.77 | 1.89 | 0.10 | 0.11 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | | P Value | 0.0 | 0.0 | 0.0005 | 0.0143 | 0.0138 | 0.0762 | 0.0379 | 0.1393 | 0.1289 | 0.1273 |
| | IGA | F Value | 3.77 | 2.68 | 1.37 | 0.20 | 0.21 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | | P Value | 0.0 | 0.0 | 0.0009 | 0.0284 | 0.0275 | 0.1523 | 0.0757 | 0.2784 | 0.2578 | 0.2546 |
| | GA | F Value | 10.40 | 7.98 | 5.06 | 2.47 | 2.49 | 1.14 | 1.71 | 0.61 | 0.69 | 0.70 |
| | | P Value | 0.0 | 0.0 | 0.0020 | 0.0620 | 0.0600 | 0.3320 | 0.1650 | 0.6070 | 0.5620 | 0.5550 |

**Line in Rectangle (LR) –** Figs. 12 & 13 depicts the performance comparison of BGA algorithm with IGA and GA based test data generation algorithms. Compare to previous

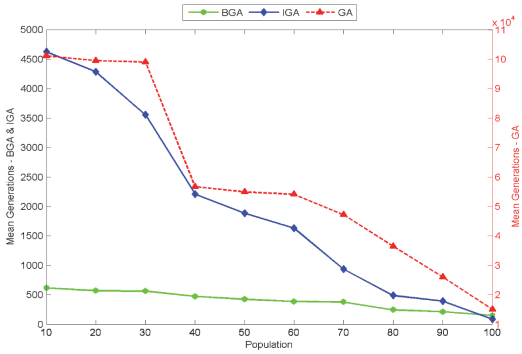


Fig. 12. Performance analysis BGA, IGA & GA for line in rectangle program with mean generations.

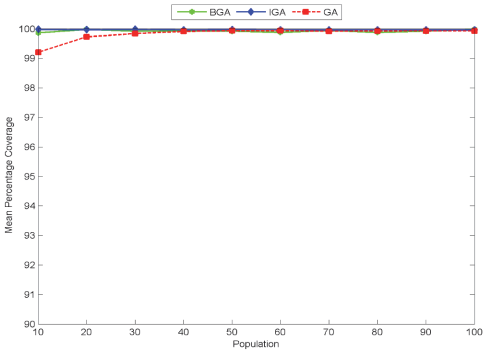


Fig. 13. Performance analysis BGA, IGA & GA for line in rectangle program with mean coverage.

two programs here the BGA is well consistent than the IGA approach, and the minimum of 150 generations to find the test data with 99.98% of coverage, which is almost close to 100% of coverage accuracy. Table 4 manifests the same with greater F scores.

**Table 4. ANOVA test evaluation of BGA, IGA & GA for line in rectangle program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| LR | BGA | F Value | 1.88 | 1.74 | 2.01 | 2.05 | 2.11 | 0.00 | 1.92 | 0.24 | 3.22 | 0.00 |
| | | P Value | 0.0005 | 0.0010 | 0.0003 | 0.0003 | 0.0003 | 0.0289 | 0.0005 | 0.0062 | 0.0000 | 0.1271 |
| | IGA | F Value | 1.36 | 1.15 | 1.54 | 1.61 | 1.70 | 0.00 | 1.41 | 0.48 | 3.36 | 0.00 |
| | | P Value | 0.0009 | 0.0018 | 0.0005 | 0.0005 | 0.0005 | 0.0578 | 0.0009 | 0.0124 | 0.0 | 0.2541 |
| | GA | F Value | 5.03 | 4.57 | 5.45 | 5.60 | 5.79 | 1.92 | 5.15 | 3.08 | 9.48 | 0.70 |
| | | P Value | 0.0020 | 0.0040 | 0.0010 | 0.0010 | 0.0010 | 0.1260 | 0.0020 | 0.0270 | 0.0 | 0.5540 |

**Meyer's Triangle Classifier (MT) –** The automated test data generation performance with MT program has been illustrated in Figs. 14 and 15. Here, once again the BGA approach overtakes the IGA and GA approaches with the record of 711 minimum number of generations to construct the test data set and promisingly reaches 99.9% of coverage with stable performance. The ANOVA results in Table 5 report the significant performance improve of BGA approach.

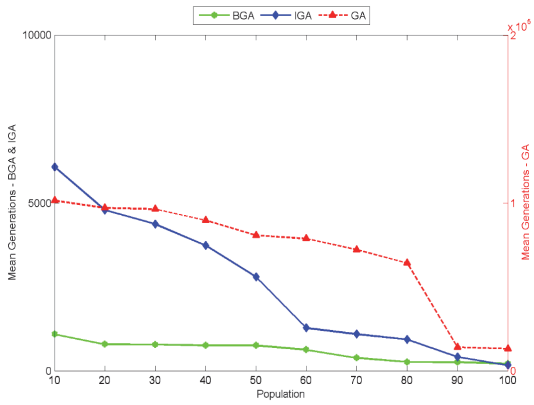


Fig. 14. Performance analysis BGA, IGA & GA for Meyer's triangle program with mean generations.

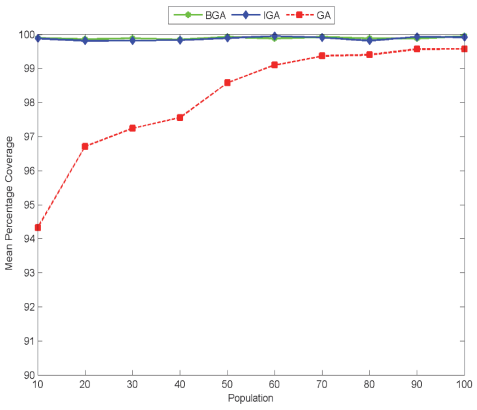


Fig. 15. Performance analysis BGA, IGA & GA for Meyer's triangle program with mean coverage.

**Table 5. ANOVA test evaluation of BGA, IGA & GA for Meyer's triangle program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| MT | BGA | F Value | 1.82 | 1.52 | 4.31 | 3.49 | 3.55 | 2.22 | 1.07 | 0.29 | 0.00 | 0.00 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0007 | 0.0046 | 0.0604 | 0.0645 |
| | IGA | F Value | 2.02 | 1.86 | 5.00 | 3.77 | 3.86 | 2.77 | 1.19 | 0.58 | 0.00 | 0.00 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0014 | 0.0092 | 0.1206 | 0.1289 |
| | GA | F Value | 6.50 | 6.14 | 13.12 | 10.39 | 10.60 | 8.17 | 4.66 | 3.30 | 1.34 | 1.28 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0030 | 0.0200 | 0.2630 | 0.2810 |

**Michael's Triangle Classifier (TM) –** Figs. 16 and 17 demonstrates the greater performance of BGA approach as it takes a minimum number of 211 generations and results 99.82% of coverage for the TM program. Whereas the IGA takes 5 times greater number of generations to find the test data set. Table 6 report the same with superior F scores than the other two approaches.

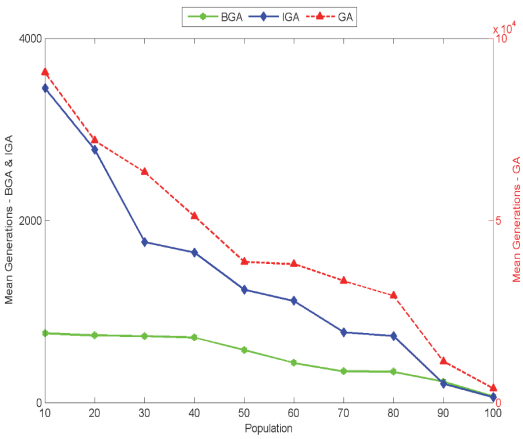


Fig. 16. Performance Analysis BGA, IGA & GA for Michael's Triangle Program with Mean Generations.

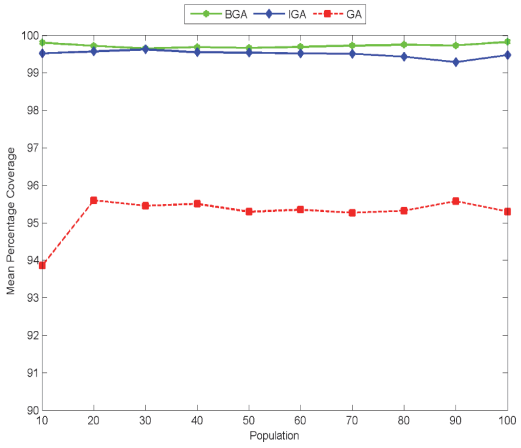


Fig. 17. Performance Analysis BGA, IGA & GA for Michael's Triangle Program with Mean Coverage.

**Table 6. ANOVA test evaluation of BGA, IGA & GA for Michael's triangle program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| TM | BGA | F Value | 43.42 | 14.93 | 6.15 | 2.33 | 0.00 | 0.09 | 0.04 | 0.26 | 1.71 | 0.08 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0319 | 0.0156 | 0.0202 | 0.0055 | 0.0010 | 0.0166 |
| | IGA | F Value | 63.67 | 20.92 | 7.76 | 2.03 | 0.00 | 0.17 | 0.08 | 0.52 | 1.10 | 0.15 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0638 | 0.0312 | 0.0404 | 0.0110 | 0.0018 | 0.0330 |
| | GA | F Value | 143.50 | 48.51 | 19.25 | 6.53 | 1.84 | 2.40 | 2.20 | 3.18 | 4.46 | 2.35 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.1390 | 0.0680 | 0.0880 | 0.0240 | 0.0040 | 0.0720 |

**Number of Days between two Dates (ND) –** The test data generation performance on ND program is illustrated in Figs. 18 and 19. Here the performance of BGA and IGA are same when there is an increase in population size, however the collective results shown that BGA takes 8 iterations to complete the automated test data generation process which is faster than IGA and GA approaches. And BGA achieves 99.65% of mean percentage coverage, which is closer to IGA, however the F-score from Table 7 shown that BGA has consistent performance than IGA.

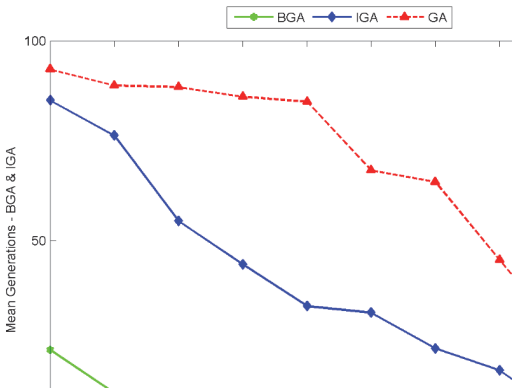


Fig. 18. Performance analysis BGA, IGA & GA for number of days program with mean generations.

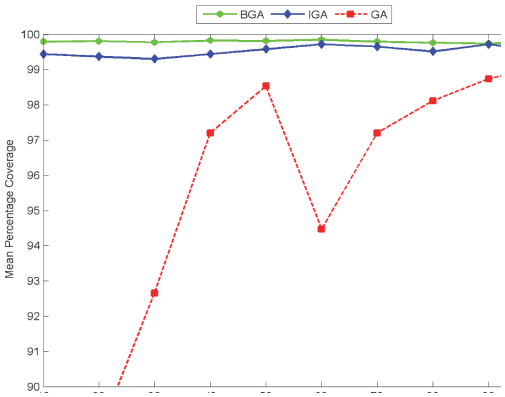


Fig. 19. Performance analysis BGA, IGA & GA for number of days program with mean coverage.

**Table 7. ANOVA test evaluation of BGA, IGA & GA for number of days program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| ND | BGA | F Value | 7.99 | 6.40 | 4.88 | 2.07 | 6.68 | 8.52 | 7.07 | 7.78 | 5.92 | 4.22 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0010 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | IGA | F Value | 10.52 | 8.13 | 5.85 | 1.64 | 8.56 | 11.31 | 9.13 | 10.20 | 7.41 | 4.87 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0018 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | GA | F Value | 25.39 | 20.08 | 15.01 | 5.65 | 21.03 | 27.16 | 22.31 | 24.69 | 18.48 | 12.83 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0040 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

**Sthamer's Triangle Classifier (ST) –** based on the performance comparisons as shown in Figs. 20 and 21, the BGA wins the race once again by reaching the solution with the minimum of 202 generations and achieves 99.97% of coverage. Here the IGA based approach is 6 times costlier than BGA method, and the F-score has been reduces 60% by BGA when compare to IGA as shown in Table 8.

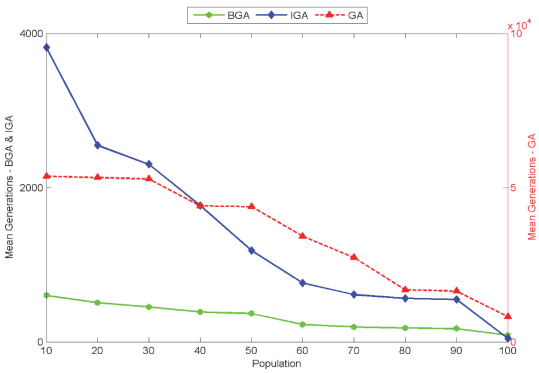


Fig. 20. Performance analysis BGA, IGA & GA for Sthamer's triangle program with mean generations.

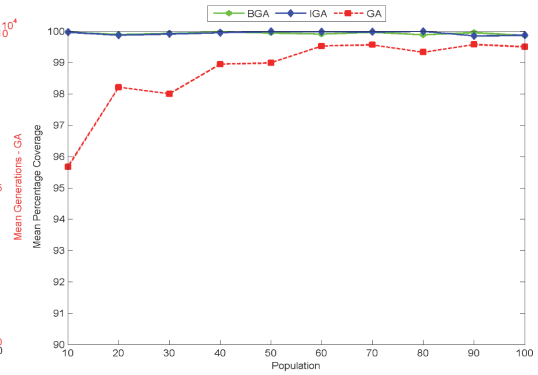


Fig. 21. Performance analysis BGA, IGA & GA for Sthamer's triangle program with mean coverage.

**Table 8. ANOVA test evaluation of BGA, IGA & GA for Sthamer's triangle program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| ST | BGA | F Value | 6.60 | 4.24 | 2.70 | 5.10 | 0.10 | 0.43 | 0.37 | 0.00 | 0.23 | 0.00 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0140 | 0.0021 | 0.0030 | 0.0647 | 0.0069 | 0.0585 |
| | IGA | F Value | 8.43 | 4.89 | 2.58 | 6.18 | 0.21 | 0.85 | 0.74 | 0.00 | 0.45 | 0.00 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0280 | 0.0041 | 0.0060 | 0.1294 | 0.0138 | 0.1170 |
| | GA | F Value | 20.74 | 12.88 | 7.75 | 15.75 | 2.48 | 3.91 | 3.66 | 1.27 | 3.02 | 1.36 |
| | | P Value | 0.0 | 0.0 | 0.0 | 0.0 | 0.0610 | 0.0090 | 0.0130 | 0.2820 | 0.0300 | 0.2550 |

**Wegener's Triangle Classifier (WT) –** the investigation study with WT program once again justifies that the BGA approach outperforms IGA and GA based methods. As shown in Figs. 22 and 23 the BGA based approach achieves the feasible test data with the minimum of 71 generations and also reached the coverage of 99.85%, which is similar to IGA. But the robustness of BGA is proved in Table 9 through decent reduce in F-score comparatively.

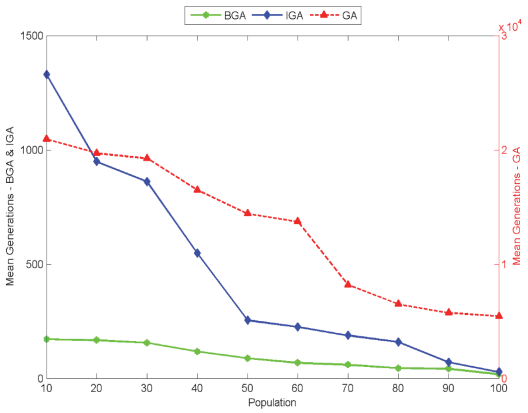


Fig. 22. Performance analysis BGA, IGA & GA for Wegener's triangle program with mean generations.

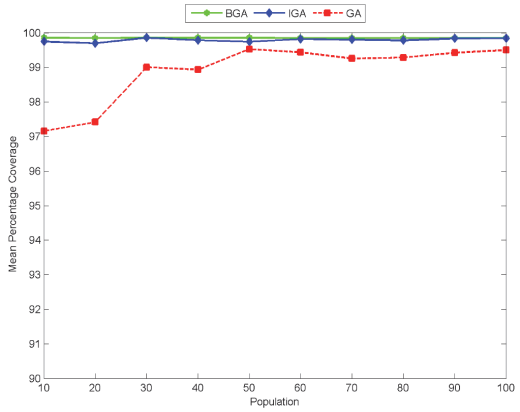


Fig. 23. Performance analysis BGA, IGA & GA for Wegener's triangle program with mean coverage.

**Table 9. ANOVA test evaluation of BGA, IGA & GA for Wegener's triangle program.**

| Program | Technique | ANOVA | Population Size | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| WT | BGA | F Value | 13.39 | 3.31 | 3.86 | 1.95 | 0.40 | 0.00 | 0.00 | 0.25 | 0.00 | 0.00 |
| | | P Value | 0.0 | 0.0005 | 0.0003 | 0.0053 | 0.0120 | 0.0629 | 0.1058 | 0.0182 | 0.0028 | 0.0014 |
| | IGA | F Value | 18.62 | 3.50 | 4.33 | 1.45 | 0.81 | 0.00 | 0.00 | 0.50 | 0.00 | 0.00 |
| | | P Value | 0.0 | 0.0009 | 0.0005 | 0.0105 | 0.0239 | 0.1257 | 0.2115 | 0.0362 | 0.0055 | 0.0028 |
| | GA | F Value | 43.39 | 9.80 | 11.64 | 5.24 | 3.82 | 1.20 | 0.55 | 3.12 | 1.05 | 1.00 |
| | | P Value | 0.0 | 0.0020 | 0.0010 | 0.0230 | 0.0520 | 0.2740 | 0.4610 | 0.0790 | 0.0120 | 0.0060 |

In the overall, the BGA outperforms the other GA based approaches. The study on experimental results clearly indicates that an extra Buffer spaced Genetic Algorithm (BGA) has significant effect on GA's performance upgrade, especially for the application of automated test data generation. Fig. 24 summarizes the overall performance comparison between BGA and IGA based test data generation on benchmark programs.

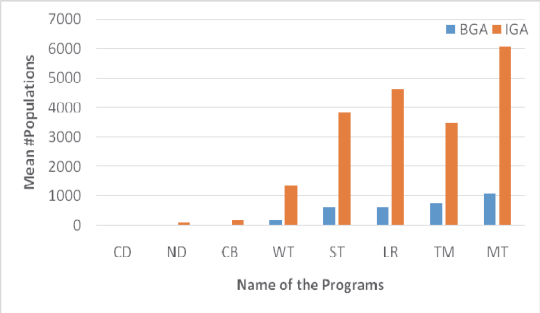


Fig. 24. Performance comparison between BGA and IGA.

## 7. CONCLUSIONS

A novel Genetic Algorithm (GA) based test data generator for branch coverage criterion is proposed in this paper. An extra buffer space is provided for GA for maintaining the list of covered target branches and to store the test data. When GA searches test data set for a specific target, the current population may contain successful test data for other target branch which is to be covered in future. In such situation, those test data are store in the buffer space and the corresponding branch is marked as covered. Hence the test data for non-specific targ*et al.*so get stored, this reduces reasonable amount of population generation and thus saves the software testing time. The proposed Buffered Genetic Algorithm (BGA) based automated test data generation is evaluated with eight benchmark programs and the performance is compared with the existing GA based approaches. The quantified results indicate the superior performance of the proposed BGA based test data generation approach.

## 8. FUTURE ENHANCEMENTS

In future, we plan to use BGA for test case generation for other data types, and also we can use granular computing techniques to develop new methods for automated branch coverage in software testing [51, 56, 75] A. Skowron, 2016; D. Dubois, 2016; Y. Yao, 2016; D. Ciucci, 2016; H. Liu, 2016; S. S. S. Ahmad, 2017; G. Wang, 2017; H. Liu, 2017.

## REFERENCES

1. W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for

non-functional system properties," *Information and Software Technology*, Vol. 51, 2009, pp. 957-976.
2. K. Agarwal and G. Srivastava, "Towards software test data generation using discrete quantum particle swarm optimization," in *Proceedings of the 3rd ACM India Software Engineering Conference*, 2010, pp. 65-68.
3. M. A. Ahmed and I. Hermadi, "GA-based multiple paths test data generator," *Computers & Operations Research*, Vol. 35, 2008, pp. 3107-3124.
4. J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," *Artificial Neural Nets and Genetic Algorithms*, 1998, pp. 325-328.
5. E. Alba and F. Chicano, "Observations in using parallel and sequential evolutionary algorithms for automatic software testing," *Computers & Operations Research*, Vol. 35, 2008, pp. 3161-3183.
6. S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, pp. 742-762.
7. A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *Proceedings of the 3rd IEEE International Conference on Software Testing*, *Verification and Validation*, 2010, pp. 205-214.
8. A. Arcuri and X. Yao, "A memetic algorithm for test data generation of object-oriented software," in *Proceedings of IEEE Congress on Evolutionary Computation*, 2007, pp. 2048-2055.
9. T. Bäck, D. B. Fogel, and Z. Michalewicz, eds., *Evolutionary Computation 1: Basic Algorithms and Operators*, Vol. 1, CRC Press, NY, 2000.
10. A. Baresel, H. Pohlheim, and S. Sadeghipour, "Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms," in *Genetic and Evolutionary Computation*, Springer Berlin/Heidelberg, 2003, pp. 215-215.
11. A. Baresel, H. Sthamer, and M. Schmidt, "Fitness function design to improve evolutionary structural testing," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, 2002, pp. 1329-1336.
12. B. Beizer, *Software Testing Techniques*, Dreamtech Press, 2003.
13. D. J. Berndt and A. Watkins, "High volume software testing using genetic algorithms," in *Proceedings of the 38th IEEE Annual Hawaii International Conference on System Sciences*, 2005, p. 318b.
14. D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, and A. Watkins, "Breeding software test cases with genetic algorithms," in *Proceedings of the 36th IEEE Annual Hawaii International Conference on System Sciences*, 2003, pp. 10-20.
15. S. Binitha and S. Sathya, "A survey of bio inspired optimization algorithms," *International Journal of Soft Computing and Engineering*, Vol. 2, 2012, pp. 137-151.
16. R. Blanco, J. Tuya, and B. Adenso-Díaz, "Automated test data generation using a scatter search approach," *Information and Software Technology*, Vol. 51, 2009, pp. 708-720.
17. R. Blanco, J. Tuya, E. Diaz, and B. A. Diaz, "A scatter search approach for automated branch coverage in software testing," *Engineering Intelligent Systems for Electrical Engineering and Communications*, Vol. 15, 2007, pp. 135-141.
18. A. Bouchachia, "An immune genetic algorithm for software test data generation," in

*Proceedings of the 7th IEEE International Conference on Hybrid Intelligent Systems*, 2007, pp. 84-89.

19. P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, 2000, pp. 209-218.
20. Y. Cao, C. Hu, and L. Li, "An approach to generate software test data for a specific path automatically with genetic algorithm," in *Proceedings of the 8th IEEE International Conference on Reliability*, *Maintainability and Safety*, 2009, pp. 888-892.
21. Y. Chen, Y. Zhong, T. Shi, and J. Liu, "Comparison of two fitness functions for GA-based path-oriented test data generation," in *Proceedings of the 5th IEEE International Conference on Natural Computation*, Vol. 4, 2009, pp. 177-181.
22. I. Chung and J. M. Bieman, "Generating input data structures for automated program testing," *Software Testing*, *Verification and Reliability*, Vol. 19, 2009, pp. 3-36.
23. L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, Vol. 3, 1976, pp. 215-222.
24. L. N. de Castro and F. J. von Zuben, "Learning and optimization using the clonal selection principle," *IEEE Transactions on Evolutionary Computation*, Vol. 6, 2002, pp. 239-251.
25. E. Díaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on tabu search," in *Proceedings the 18th IEEE International Conference on Automated Software Engineering*, 2003, pp. 310-313.
26. E. Díaz, J. Tuya, R. Blanco, and J. J. Dolado, "A tabu search algorithm for structural software testing," *Computers & Operations Research*, Vol. 35, 2008, pp. 3052-3072.
27. R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology*, Vol. 5, 1996, pp. 63-86.
28. J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software: Practice and Experience*, Vol. 42, 2012, pp. 1331-1362.
29. G. Fraser and F. Wotawa, "Using model-checkers to generate and analyze property relevant test-cases," *Software Quality Journal*, Vol. 16, 2008, pp. 161-183.
30. D. E. Goldberg, *Genetic Algorithms in Search*, *Optimization*, *and Machine Learning*, Addison-Wesley, 1989, Reading, MA.
31. A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques," *ACM SIGSOFT Software Engineering Notes*, Vol. 23, 1998, pp. 53-62.
32. A. Gotlieb, B. Botella, and M. Rueher, "A CLP framework for computing structural test data," *Computational Logic*, 2000, pp. 399-413.
33. G. Peters and R. Weber, "DCC: A framework for dynamic granular clustering," *Granular Computing*, Vol. 1, 2016, pp. 1-11.
34. H. Gross, P. M. Kruse, J. Wegener, and T. Vos, "Evolutionary white-box software test with the evotest framework: A progress report," in *Proceedings of IEEE International Conference on Software Testing*, *Verification and Validation Workshops*, 2009, pp. 111-120.

35. M. Harman and A. Mansouri, "Search based software engineering: Introduction," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, p. 737.
36. M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, pp. 226-247.
37. M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, Vol. 30, 2004, pp. 3-16.
38. M. Harman, K. Lakhotia, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th ACM Annual Conference on Genetic and Evolutionary Computation*, 2007, pp. 1098-1105.
39. M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Technical Report TR-09-03, Department of Computer Science, King's College London, 2009.
40. I. Hermadi and M. A. Ahmed, "Genetic algorithm based test data generator," in *Proceedings of IEEE Congress on Evolutionary Computation*, Vol. 1, 2003. pp. 85-91.
41. I. Hermadi, C. Lokan, and R. Sarker, "Genetic algorithm based path testing: challenges and key parameters," in *Proceedings of the 2nd IEEE World Congress on Software Engineering*, Vol. 2, 2010, pp. 241-244.
42. J. H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, MA, 1992.
43. I. Hooda and R. Chhillar, "A review: Study of test case generation techniques," *International Journal of Computer Applications*, Vol. 107, 2014, p. 16.
44. W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Transactions on Software Engineering*, Vol. 4, 1977, pp. 266-278.
45. B. F. Jones, D. E. Eyres, and H. H. Sthamer, "A strategy for using genetic algorithms to automate branch and fault-based testing," *The Computer Journal*, Vol. 41, 1998, pp. 98-107.
46. B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, Vol. 11, 1996, pp. 299-306.
47. S. Kansomkeat, J. Offutt, A. Abdurazik, and A. Baldini, "A comparative evaluation of tests generated from different UML diagrams," in *Proceedings of the 9th IEEE International Conference on Software Engineering*, *Artificial Intelligence*, *Networking*, *and Parallel/Distributed Computing*, 2008, pp. 867-872.
48. B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, Vol. 16, 1990, pp. 870-879.
49. B. Korel, "Automated test data generation for programs with procedures," *ACM SIGSOFT Software Engineering Notes*, Vol. 21, 1996, pp. 209-215.
50. F. C. Kuo, T. Y. Chen, H. Liu, and W. K. Chan, "Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters," *Software Quality Journal*, Vol. 16, 2008, pp. 303-327.
51. L. Livi and A. Sadeghian, "Granular computing, computational intelligence, and the analysis of non-geometric input spaces," *Granular Computing*, Vol. 1, 2016, pp. 13-20.

52. H. Li and C. P. Lam, "Software test data generation using ant colony optimization," in *Proceedings of International Conference on Computational Intelligence*, 2004, pp. 1-4.
53. K. Liaskos and M. Roper, "Hybridizing evolutionary testing with artificial immune systems and local search," in *Proceedings of IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, pp. 211-220.
54. J. C. Lin and P. L. Yeh, "Using genetic algorithms for test case generation in path testing," in *Proceedings of IEEE 9th Asian* Test Symposium, 2000, pp. 241-246.
55. S. Mairhofer, R. Feldt, and R. Torkar, "Search-based software testing and test data generation for a dynamic programming language," in *Proceedings of the 13th ACM Annual Conference on Genetic and Evolutionary Computation*, 2011, pp. 1859-1866.
56. M. Antonelli, P. Ducange, B. Lazzerini, and F. Marcelloni, "Multi-objective evolutionary design of granular rule-based classifiers," *Granular Computing*, Vol. 1, 2016, pp. 37-58.
57. T. Manikumar, A. J. S. Kumar, and R. Maruthamuthu, "Automated test data generation for branch testing using incremental genetic algorithm," *Sādhanā*, Vol. 41, 2016, pp. 959-976.
58. P. McMinn, "Search-based software test data generation: a survey," *Software Testing*, *Verification and Reliability*, Vol. 14, 2004, pp. 105-156.
59. P. McMinn, "Search-based software testing: Past, present and future," in *Proceedings of the 4th IEEE International Conference on Software Testing*, *Verification and Validation Workshops*, 2011, pp. 153-163.
60. P. McMinn and M. Holcombe, "Evolutionary testing using an extended chaining approach," *Evolutionary Computation*, Vol. 14, 2006, pp. 41-64.
61. P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," *ACM Transactions on Software Engineering and Methodology*, Vol. 18, 2009, p. 11.
62. P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to search-based test data generation," in *Proceedings of ACM International Symposium on Software Testing and Analysis*, 2006, pp. 13-24.
63. A. Mehrmand, "A factorial experiment on scalability of search-based software testing," Master's Thesis, Thesis No. MSE-2009:20, Blekinge Institute of Technology, Sweden, 2009.
64. C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, 1997. pp. 307-308.
65. C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 1085-1110.
66. J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using genetic algorithm and program dependence graphs," *Information and Software Technology*, Vol. 48, 2006, pp. 586-605.
67. H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom software engineering," *IEEE Software*, Vol. 4, 1987, p. 19.
68. G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, John Wiley & Sons, 2011.

69. P. B. Nirpal and K. V. Kale, "Comparison of software test data for automatic path coverage using genetic algorithm," *International Journal of Computer Science & Engineering Technology*, Vol. 1, 2011, pp. 12-16.
70. A. J. Offutt and J. H. Hayes, "A semantic model of program faults," in *Proceedings of ACM SIGSOFT Software Engineering Notes*, Vol. 21, 1996, pp. 195-200.
71. J. Oh, M. Harman, and S. Yoo, "Transition coverage testing for simulink/stateflow models using messy genetic algorithms," in *Proceedings of the 13th ACM Annual Conference on Genetic and Evolutionary Computation*, 2011, pp. 1851-1858.
72. A. Pachauri and G. Srivastava, "Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism," *Journal of Systems and Software*, Vol. 86, 2013, pp. 1191-1208.
73. R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing Verification and Reliability*, Vol. 9, 1999, pp. 263-282.
74. M. Pei, E. D. Goodman, Z. Gao, and K. Zhong, "Automated software test data generation using a genetic algorithm," Technical Report No. 1, Department of Computer Science, Michigan State University, 1994, pp. 1-15.
75. P. Lingras, F. Haider, and M. Triff, "Granular meta-clustering based on hierarchical, network, and temporal connections," *Granular Computing*, Vol. 1, 2016, pp. 71-92.
76. P. W. Tsai, J. S. Pan, S. M. Chen, B. Y. Liao, and S. P. Hao, "Parallel cat swarm optimization," in *Proceedings of International Conference on Machine Learning and Cybernetics*, Vol. 6, 2008, pp. 3328-3333.
77. P. W. Tsai, J. S. Pan, S. M. Chen, and B. Y. Liao, "Enhanced parallel cat swarm optimization based on the Taguchi method," *Expert Systems with Applications*, Vol. 39, 2012, pp. 6309-6319.
78. H. H. Sthamer, "The automatic generation of software test data using genetic algorithms," Doctoral dissertation, Department of Computer Studies, University of Glamorgan, 1995.
79. S. M. Chen and N. Y. Chung, "Forecasting enrollments using high-order fuzzy time series and genetic algorithms," *International Journal of Intelligent Systems*, Vol. 21, 2006, pp. 485-501.
80. S. M. Chen and T. H. Chang, "Finding multiple possible critical paths using fuzzy PERT," *IEEE Transactions on Systems*, *Man*, *and Cybernetics – Part B: Cybernetics*, Vol. 31, 2001, pp. 930-937.
81. S. M. Chen and C. Y. Chien, "Parallelized genetic colony systems for solving the traveling salesman problem," *Expert Systems with Applications*, Vol. 38, 2011, pp. 3873-3883.
82. X. B. Tan, C. Longxin, and X. Xiumei, "Test data generation using annealing immune genetic algorithm," in *Proceedings of the 5th IEEE International Joint Conference on INC*, *IMS and IDC*, 2009, pp. 344-348.
83. P. Thevenod-Fosse and H. Waeselynck, "STATEMATE applied to statistical software testing," *ACM SIGSOFT Software Engineering Notes*, Vol. 18, 1993, pp. 99-109.
84. P. Tonella, "Evolutionary testing of classes," *ACM SIGSOFT Software Engineering Notes*, Vol. 29, 2004, pp. 119-128.

85. N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," *ACM SIGSOFT Software Engineering Notes*, Vol. 23, 1998, pp. 73-81.
86. N. Tracey, J. Clark, K. Mander, and J. McDermid, "Automated test-data generation for exception conditions," *Software-Practice and Experience*, Vol. 30, 2000, pp. 61-79.
87. J. Voas, L. Morell, and K. Miller, "Predicting where faults can hide from testing," *IEEE Software*, Vol. 8, 1991, pp. 41-48.
88. Y. Wang, Z. Bai, M. Zhang, W. Du, Y. Qin, and X. Liu, "Fitness calculation approach for the switch-case construct in evolutionary testing." in *Proceedings of the 10th ACM Annual Conference on Genetic and Evolutionary Computation*, 2008, pp. 1767-1774.
89. J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, Vol. 43, 2001, pp. 841-854.
90. J. Wegener, A. Baresel, and H. Sthamer, "Suitability of evolutionary algorithms for evolutionary testing," in *Proceedings of the 26th IEEE Annual International Computer Software and Applications Conference*, 2002, pp. 287-289.
91. J. Wegener, K. Buhr, and H. Pohlheim, "Automatic test data generation for structural testing of embedded software systems by evolutionary testing," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, 2002, pp. 1233-1240.
92. L. D. Whitley, "The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best," *ICGA Journal*, Vol. 89, 1989, pp. 116-123.
93. A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *Proceedings of the 9th ACM Annual Conference on Genetic and Evolutionary Computation*, 2007, pp. 1121-1128.
94. S. Xanthakis, C. Ellis, C. Skourlas, A. le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *Proceedings of the 5th International Conference on Software Engineering and Applications*, 1992, pp. 625-636.
95. M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques," *Empirical Software Engineering*, Vol. 12, 2007, pp. 183-239.
96. H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, Vol. 29, 1997, pp. 366-427.

**T. Manikuma**r received his Bachelor's degree in Science and his Master's degree in Computer Applications from Madurai Kamaraj University in 2004 and 2008, respectively. He obtained his Ph.D. degree in Computer Applications from Anna University in 2018. Currently he is working as an Assistant Professor in MCA Department at RVS College of Engineering, Dindigul, Tamilnadu, India. His current research interests include software engineering, search based software testing, optimization techniques.

**A. John Sanjeev Kumar** received the Ph.D. degree in Computer Applications from Anna University Chennai, India in 2010. Currently he is working as an Assistant Professor in MCA Department, Thiagarajar College of Engineering, Madurai, India. His research interests include spatial databases, image processing, cloud computing and software engineering.