Information Processing Letters 114 (2014) 304–316

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Automatic generation of basis test paths using variable length genetic algorithm

Ahmed S. Ghiduk [a,b,*]

[a] College of Computers and Information Technology, Taif University, Taif, Saudi Arabia
[b] Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Beni-Suef, Egypt

ARTICLE INFO

Article history:
Received 10 July 2012
Received in revised form 21 January 2014
Accepted 23 January 2014
Available online 27 January 2014
Communicated by J.L. Fiadeiro

Keywords:
Software engineering
Genetic algorithm
Basis path testing
Test path generation

ABSTRACT

Path testing is the strongest coverage criterion in white box testing. Finding target paths is a key challenge in path testing. Genetic algorithms have been successfully used in many software testing activities such as generating test data, selecting test cases and test cases prioritization. In this paper, we introduce a new genetic algorithm for generating test paths. In this algorithm the length of the chromosome varies from iteration to another according to the change in the length of the path. Based on the proposed algorithm, we present a new technique for automatically generating a set of basis test paths which can be used as testing paths in any path testing method. The proposed technique uses a method to verify the independency of the generated paths to be included in the basis set of paths. In addition, this technique employs a method for checking the feasibility of the generated paths. We introduce new definitions for the key concepts of genetic algorithm such as chromosome representation, crossover, mutation, and fitness function to be compatible with path generation. In addition, we present a case study to show the efficiency of our technique. We conducted a set of experiments to evaluate the effectiveness of the proposed path generation technique. The results showed that the proposed technique causes substantial reduction in path generation effort, and that the proposed *GA* algorithm is effective in test path generation.

1. Introduction

Structural testing requires the execution of a set of test paths in the program under testing. Generating this set of paths is a critical task and can influence the efficacy and cost of testing activity. Therefore, the automation and reduction of this task are strongly desirable for simplifying the testing process.

Basis path testing is one of the very powerful structural testing criteria. Basis path testing requires number of test paths (basis paths) equals to the cyclomatic complexity of program [1] such that every path is an independent path and all edges in the control-flow graph (*CFG*) are covered by all paths in the basis set of paths. In addition, a path that is not contained in the basis set of paths can be constructed by a linear combination of the paths in this set.

Search-based optimization techniques such as simulated annealing, genetic algorithms, and genetic programming have been successfully applied to a wide number of software engineering activities, right across the life-cycle of the software from requirements engineering (e.g., [2]), project planning and cost estimation (e.g., [3,4]) through testing (e.g., [5–12]), to automated maintenance (e.g., [13]), service oriented software engineering (e.g., [14]), compiler optimization (e.g., [15]) and quality assessment (e.g., [16]). Recently, Harman et al. [17] introduced a comprehensive review and classification of literature on search-based software engineering. This work identifies research trends and relationships between the techniques applied and the applications to which they have been applied and highlights

* Correspondence to: Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Beni-Suef, Egypt.
E-mail address: asaghiduk@tu.edu.sa.

0020-0190/$ – see front matter
http://dx.doi.org/10.1016/j.ipl.2014.01.009

gaps in the literature and avenues for further research. However, as the list of application areas above indicates, search-based optimization techniques can be applied right across the spectrum of software engineering activity [17]. A wide range of different optimization and search techniques can and have been used. The most widely used are local search (e.g., [18]), simulated annealing (e.g., [19]), genetic algorithm (e.g., [8–12,20]) and genetic programming (e.g., [3]).

The major contributions of this paper are as follows. This paper introduces a new variable length genetic algorithm. In the proposed algorithm, the length of each chromosome varies from iteration to iteration according to the changing in the length of the path. Based on the proposed algorithm, the paper introduces a new technique for automatically generating a set of basis test paths. This paper presents new definitions for all key elements of the new genetic algorithm such as representation, crossover, and mutation to be compatible with path generation process. In addition, it introduces a new fitness function to evaluate the generated paths. Besides, it uses a method to verify the independency of the generated paths to append them to the basis set of paths. In addition, the proposed technique employs a method for checking the feasibility of the generated paths. The proposed technique overcomes the problem of loops in the CFG. In addition, the paper presents a case study to show the efficiency of our new technique. Some experiments to evaluate the effectiveness of the proposed approach and answer the two research questions: RQ1) How effective is our proposed technique in generating basis test paths? and RQ2) How effective is our proposed technique in generating feasible test paths? are conducted. The results showed that the proposed approach causes substantial reduction in the path generation effort, and that the proposed *GA* algorithm is effective in generating the basis test paths.

The rest of the paper is organized as follows: Section 2 discusses and compares related work. Section 3 introduces the problem formulation. Section 4 gives some definitions and basic concepts. Section 5 presents our proposed strategy. Section 6 introduces a case study of the proposed method. Section 7 describes empirical studies which are performed to evaluate our proposed strategy and tool. Section 8 concludes this paper.

2. Related research

Some path generation methods have been introduced so far. Bertolino and Marre [21] provided a path generation method by using a reduced *CFG*. Although all the statements and branches can be covered by the generated set of paths, it cannot be assured that the set of paths generated by this method is a basis set of paths.

Pool [22] discussed a basis paths generation method based on the depth first search in *CFG*. It uses a recursive search in the *CFG*. As Pool's method, the loop is not taken into account. In addition, this method didn't consider how to choose the successor of multiple-successors node in a *CFG* to build a basis path during the construction of the basis set of paths. The major limitation is that the method is unreliable on unstructured code. Useful test cases may be generated, but the number of paths in the basis set is no longer equal to the complexity measure of the control flow graph.

Guangmei et al. [23] presented an automatic generation method of basis set of paths which is built by searching the *CFG* using depth-first searching method. In order to avoid the infinite repetition of the algorithm, and for reducing the searching procedure, the sub-path from the multi-indegree nodes to the end node of a program and the sub-path that contains a loop are recorded during the construction of a basis path. This method did not handle infeasible paths.

Yan and Zhang [24] presented a method for generating a finite set F of feasible paths which satisfies the basis path coverage criterion. Then, they found a minimal subset S of set F such that S satisfies the test coverage criterion. The first step should check the feasibility of all paths and feasibility checking is quite time-consuming.

Zhonglin and Lingxia [25] and Qingfeng and Xiao [26] use cyclomatic complexity in generating a set of linearly independent paths. Many of the generated basis paths are infeasible because data dependences exist in variables involved in decision node. They combine the baseline method with the dependence relationship to avoid selecting infeasible paths. These methods didn't handle loops.

Genetic algorithms have been the most widely employed search-based technique in software testing activities such as finding test data [8–12,27,28]. Bint and Site [29] discussed a path generation method based on genetic algorithm. The drawback of this method is that the generated set of paths cannot cover all edges of the *CFG* because the loop operation is removed. In addition, this method cannot generate basis paths. To overcome the drawbacks of the previous methods, we introduce a new genetic algorithm based test path generation technique.

Why do we use genetic algorithms?

From the previous discussion, there is no doubt that basis paths generation is a demand task and genetic algorithm is one of the powerful search-based techniques which have been successfully used throughout the lifecycle of the software. For the following reasons we will utilize genetic algorithms to generate the basis test paths.

- Search-based algorithms have been successfully applied to generate the paths in many software engineering activities. K. Derderian et al. [30], A. Kalaji et al. [31,32], and T. Yano et al. [33] have been successfully used genetic algorithms based techniques for generating feasible transition paths (FTPs) for model based testing. In addition, P. Srivastava et al. [34], A. Ghiduk [35], and S. Babu Lam [36] have been successfully used ant colony based techniques for finding test paths. J. Bint and R. Site used genetic algorithms and error prone path identification for optimizing testing efficiency [29]. Alba et al. [37] illustrated the efficiency genetic algorithms for detecting the shortest paths that lead to deadlocks in large concurrent java programs.
- Genetic algorithm outperforms depth-first search (DFS) in solving some search problems. M.M. Naoghare and V.M. Deshmukh [38] used genetic algorithm to

solve verbal arithmetic problems and they proved that genetic algorithm can find the solution more quickly than DFS with less memory.

- The search space in path generation problem is very large. Where, there are too large number of paths in the program and it may be infinite in case of loops. Then, the search space of generating basis paths is very large which is suitable for applying search-based algorithms. Therefore, our proposed approach uses *GA* and *dd-graph* to represent the program [21]. The *dd-graph* can be constructed by reducing *CFG* [21], interprocedural call graph [40] or class call graph [41]. Therefore, the proposed approach can be applied on the unit level, interprocedural level, and object oriented level. This means that our proposed method can be applied on small search space such as control flow graph of program unit as well as large search space such as interprocedural control flow graph of structured program [40] or class call graph [41] of object oriented program.
- The exhaustive searches based methods can stack in the local minima. In contrast, genetic algorithms are powerful tool for large-scale nonlinear optimization problems [39]. The additional advantages over conventional methods such as iterative least squares is that the sampling is global, rather than local, thereby reducing the tendency to become entrapped in local minima and avoiding a dependency on an assumed starting model. They also share a desirable characteristic of the local methods in that they assimilate and take advantage of information collected during the sampling of the model space, resulting in an extremely efficient and robust optimization technique [39].

3. Basis path testing

Structural testing generally requires the execution of a set Q of paths in the program under testing. Determining Q is a very important and critical task, and its automation is strongly desirable for easing the testing strategy. This task can influence the efficacy and the testing effort and costs.

Thomas McCabe [1] came up with the idea of using a vector space to carry out path testing. A vector space is a set of elements along with certain operations that can be performed upon these elements. What makes vector spaces an attractive proposition to testers is that they contain a basis. The basis of a vector space contains a set of vectors that are independent of one another, and have a spanning property; this means that everything within the vector space can be expressed in terms of the elements within the basis. What McCabe noticed was that if a basis could be provided for a program graph, this basis could be subjected to rigorous testing; if proven to be without fault, it could be assumed that those paths expressed in terms of that basis are also correct. The method devised by McCabe to carry out basis path testing has the following four steps:

- Compute the program graph.
- Calculate the cyclomatic complexity. In graph theory, the cyclomatic number is defined as $C(G) = m - n + q$, where m is number of edges in the graph G, n is number of nodes, and q is number of strongly connected components. For a program that has a single entry and exit point, $q = 1$ [24]. In basis path testing, the cyclomatic complexity should be the upper limit for the number of basis paths [42].
- Select a basis set of paths.
- Generate test cases for each of these paths.

Independent path: An independent path is a path of a program, where at least one edge of this path never appears in any other path in the control-flow graph (*CFG*).

Basis set of paths and basis path: A basis set of paths is a set of paths; every path in this set should satisfy the next three conditions:

- Every path should be an independent path.
- All edges in a *CFG* should be covered by all paths in the basis set.
- Every path not contained in this basis set of paths can be constructed by linear operations among paths in this set.

The path contained in a basis set is called a basis path. The problem of this work is defining a new genetic algorithm and using it for generating the set Q of basis paths.

4. Basic concepts

We introduce here some basic concepts that will be used throughout this work.

4.1. Genetic algorithms principles

The basic concepts of genetic algorithms (*GAs*) were developed by Holland [43]. The *GAs* start by creating an initial population of individuals, each represented by a randomly generated binary string called chromosome. The basic algorithm of *GAs*, where $P(n)$ is the population strings at generation number n, is as follows:

1. initialize $P(n)$;
2. evaluate $P(n)$;
3. ***while*** termination condition not satisfied ***do***
4. select $P(n+1)$ from $P(n)$;
5. recombine $P(n+1)$;
6. evaluate $P(n+1)$;
7. $n = n+1$;
8. ***end while***

In the evaluation step, the fitness of each individual is determined. The selection step is used to find pairs of individuals that will be combined in some way to contribute to the next generation. The process of crossover involves two chromosomes swapping chunks of data. Mutation introduces slight changes into a small proportion of population and is representative of an evolutionary step. The above algorithm will iterate until the population has evolved to

form a solution to the problem, or until a termination condition is satisfied.

4.2. Program representation

A program's structure is analyzed on the program flow-graph (i.e., an annotated directed graph which represents graphically the information needed to select the test cases).

A control-flow graph (*CFG*) is a directed graph $G = (N, E)$, with two distinguished nodes—a unique *entry* (n_0) and a unique *exit* (n_k). N is a set of nodes, where each node represents a statement, and E is a set of directed edges, where a directed edge $e = (n, m)$ is an ordered pair of adjacent nodes, called *tail* and *head* of e, respectively [21].

An inter-procedural call graph is directed graph in which nodes represent procedures in the program. An edge (p1, p2) exists if procedure p1 can call procedure p2 from some call site within p1 [40]. A **class call graph** is a directed graph in which nodes represent methods, and edges represent procedure calls between methods [41].

A dd-graph (*DDG*) is a directed graph $G = (N, E)$, where N is a set of nodes and E is a set of edges, with two distinguished edges e_0 and e_k (the unique *entry* edge and the unique *exit* edge, respectively), such that any other edge in E is reached by e_0 and reaches e_k, and such that for each node $n \in N$, $n \neq tail(e_0)$, $n \neq head(e_k)$, $(indegree(n) + outdegree(n)) > 2$, while $indegree(tail(e_0)) = 0$ and $outdegree(tail(e_0)) = 1$, $indegree(head(e_k)) = 1$ and $outdegree(head(e_k)) = 0$ [21].

An **edge *e*** in a dd-graph *DDG* is an ordered pair of adjacent nodes, called *tail* and *head* of e, respectively (i.e., $e = (tail(e), head(e))$).

A **path *p*** of length l in a dd-graph *DDG* is a sequence $p = e_0, e_1, e_2, \ldots, e_l$, where $tail(e_{i+1}) = head(e_i)$ for $i = 1, 2, \ldots, l-1$. A path p is simple if all its nodes, except possibly the first and last, are distinct. A complete path in a dd-graph *DDG* is a path from the *entry* node to the *exit* node of *DDG*. Given a path $p = e_0, e_1, e_2, \ldots, e_l$, then a path $p' = e_i, \ldots, e_j$ from e_i to e_j, with $1 \leqslant i \leqslant j \leqslant l$, is called a subpath of p.

A. Bertolino and M. Marre [21] provided a procedure to construct the dd-graph by reducing the control-flow graph of the program. Fig. 1(a) gives an example program, Fig. 1(b) shows its control-flow graph, and Fig. 1(c) provides its dd-graph.

In our proposed technique, we will represent the program under test as a dd-graph according to the above definitions.

5. Our proposed technique

5.1. Our new genetic algorithm

In this section, we present our proposed *GA* for automatic generation of basis test paths for the tested software, which uses a new fitness function to evaluate the generated test path. This fitness function depends on the concepts of the number of the adjacent edges in the dd-graph of the software under test. The algorithm searches for test paths that satisfy the three conditions of the basis set of path (see Section 3). The major components of this *GA* are discussed below.

5.1.1. The search space of our algorithm

The search space is the set of all solutions among which the desired solution resides. Each point in the search space represents one possible solution. Suppose that *DDG* is a dd-graph of tested program. The search space of our new genetic algorithm is D, where

$$D = \{\forall e \mid e \in DDG \text{ and } e \text{ is reached by entry and reaches exit}\}$$

In other words, the input domain of our algorithm is the set of all edges in the dd-graph of the program under test. For example, the search space of the example program in Fig. 1 is the set of edges $D = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$.

5.1.2. Encoding

Encoding is a process of representing individual genes. The process can be performed using bits, numbers, trees, arrays, lists or any other objects. The encoding depends mainly on solving the problem. For example, one can encode directly real or integer numbers. On the other hand, it is necessary to develop new genetic operator's specific to the problem.

The proposed *GA* uses a vector of integer numbers as a chromosome to represent the edges in the dd-graph of the program under test. The length of the vector (chromosome) depends on the length of the required basis path. Each edge in the dd-graph is represented by its index in the chromosome. In addition, each cell (value) in the vector (chromosome) is mapped into its corresponding edge using the map:

$$M : i \in \text{chromosome} \rightarrow e_i \in \text{dd-graph}.$$

For example, consider the program in Fig. 1 and suppose an example chromosome: $(0, 1, 3, 5, 9)$. By using M, the given chromosome is mapped into the path $p = \{e_0, e_1, e_3, e_5, e_9\}$.

5.1.3. Initial population

As mentioned above, each chromosome is represented by a vector of integer numbers. We randomly generate *PS* integer vectors (chromosomes) of length 2 to represent the initial population, where *PS* is the population size. The appropriate value of *PS* is experimentally determined. Each individual in the initial population contains two edges the *entry* and the *exit* of the dd-graph of the program under test. For the example in Fig. 1, all individuals have the form $(0, 9)$ corresponds to the path $\{e_0, e_9\}$. The minimum length of the chromosome is two and the maximum length equals the number of edges in the dd-graph.

5.1.4. Evaluation function

For defining the fitness function, the proposed algorithm uses the definition of paths. The fitness function depends on the probability of the adjacent edges in the

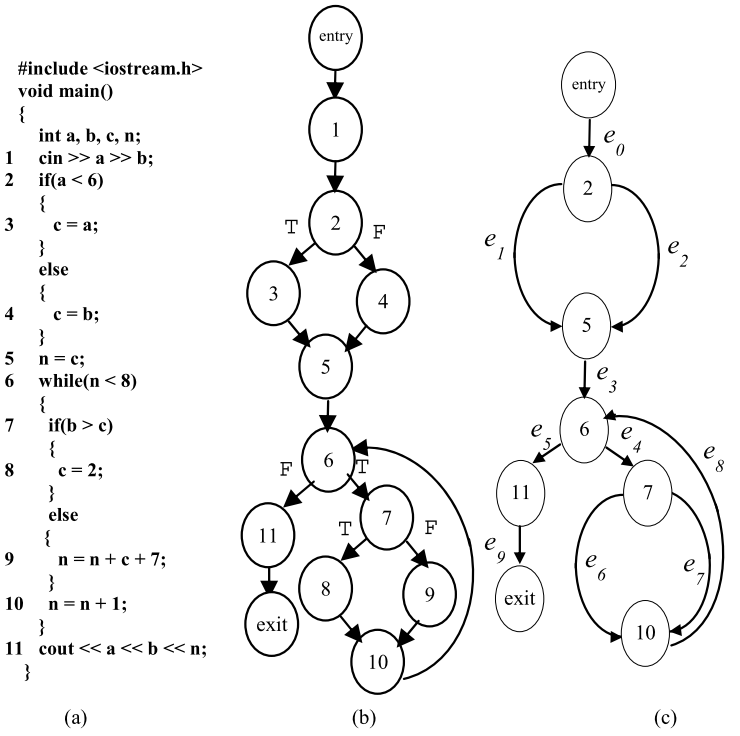

Fig. 1. (a) An example program, (b) its control-flow graph, and (c) its dd-graph.

path. The algorithm uses this fitness function to evaluate each generated path.

The fitness value $ft(v_i)$ for each chromosome v_i ($i = 1, \ldots, PS$) is calculated as follows:

$$ft(v_i) = \sum_{j=1}^{d(v_i)} w(e_j) \tag{1}$$

where, $d(v_i)$ is the number of adjacent edges in the chromosome v_i, $w(e_j)$ is the weight (probability) of edge e_j in the chromosome v_i, $w(e_j) = \frac{1}{L(v_i)}$, where $L(v_i)$ the length of the chromosome v_i (i.e., the total number of edges in the chromosome v_i). The fitness function can be written as the number of adjacent edges in the chromosome divided by the total number of edges in the same chromosome.

The fitness value is the only feedback from the problem for the *GA*. A test case that is represented by the chromosome v_i is optimal if its fitness value $ft(v_i) = 1$.

Consider Fig. 1(c) and suppose that $v_1 = (0, 1, 9)$, $v_2 = (0, 2, 3, 9)$, $v_3 = (0, 1, 3, 5, 9)$, and $v_4 = (0, 2, 3, 4, 6, 8, 9)$ are set of chromosomes of different lengths. Chromosomes v_1, v_2, v_3, and v_4 contain 3, 4, 5, and 7 genes (edges), respectively. Chromosomes v_1, v_2, v_3, and v_4 contain 2, 3, 5, and 5 adjacent edges, respectively. Therefore, $L(v_1) = 3$, $L(v_2) = 4$, $L(v_3) = 5$, and $L(v_4) = 7$ and $d(v_1) = 2$, $d(v_2) = 3$, $d(v_3) = 5$, and $d(v_4) = 5$.

The probability of each gene in $v_1 = 1/3$, the probability of each gene in $v_2 = 1/4$, the probability of each gene in $v_3 = 1/5$ and the probability of each gene in $v_4 = 1/7$.

According to Eq. (1), $ft(v_1) = 2/3$, $ft(v_2) = 3/4$, $ft(v_3) = 5/5 = 1$ and $ft(v_4) = 5/7$.

In this case, the fourth chromosome has the highest fitness. Then, the path $p_3 = \{e_0, e_1, e_3, e_5, e_9\}$, which corresponds to the chromosome v_3, is the optimal path. After that, we check the independency of this path, if this path is independent path, then we add this path to the basis set of paths.

We have to note that all chromosomes of a population in any iteration of the algorithm have the same length but these lengths vary from iteration to iteration. This means that the algorithm favors smaller paths because the lengths of chromosomes in a population are smaller than their lengths in the next population.

5.1.5. Selection

After computing the fitness of each test path in the current population, the algorithm selects test paths from all the members of the current population that will be parents of the new population. In the selection process, the proposed *GA* uses the roulette wheel method [44]. The selection process is based on spinning the roulette wheel *PS* times; each time we select a single chromosome for a new population.

5.1.6. Reproduction

In the Reproduction phase, we use three operators, crossover, mutation and breeding (a new operator we have proposed it), which are the key to the power of *GAs*. These operators create new individuals from the selected parents to form a new population.

Crossover: The proposed *GA* uses the uniform crossover to exchange information at a random position in the

selected two chromosomes to produce new two chromosomes [44]. The probability of crossover *PX* gives us the expected number of chromosomes, which undergo the crossover operation.

For each pair of coupled chromosomes we generate a random integer number *pos* from the range $[2 \dots L-1]$ (L is the number of edges in a chromosome). The number *pos* indicates the position of the crossing point. Two chromosomes $(b_1 \cdots b_{pos} b_{pos+1} \cdots b_n)$ and $(c_1 \cdots c_{pos} \cdot c_{pos+1} \cdots c_n)$ are replaced by a pair of their offspring $(b_1 \cdots b_{pos} c_{pos+1} \cdots c_n)$ and $(c_1 \cdots c_{pos} b_{pos+1} \cdots b_n)$.

Suppose that $C_1 = (\mathbf{0}, \mathbf{1}, \mathbf{3}, \mathbf{5}, \mathbf{9})$ and $C_2 = (0, 2, 3, 4, 9)$ are two chromosomes and $pos = 3 \in [2 \dots 4]$, where $L = 5$. After applying the crossover operator the new chromosomes are $B_1 = (\mathbf{0}, \mathbf{1}, \mathbf{3}, 4, 9)$ and $B_2 = (0, 2, 3, \mathbf{5}, \mathbf{9})$.

Mutation: Mutation operates after the crossover operator and changes each cell with the pre-determined probability. The probability of mutation *PM*, gives us the expected number of mutated cells [44]. Our proposed *GA* performs the following procedure for each chromosome in the current population and for each cell within the chromosome:

- Generate a random number r from the range $[0 \dots 1]$;
- If $r < MP$ then mutate the cell by replacing the edge with another edge of its siblings (edges with the same parent are called siblings such as e_1 and e_2).

Suppose that $C_1 = (0, 1, 3, 5, 9)$ is a chromosome. The second cell which has the value 1 will mutate to the value 2. Therefore the new chromosome will be $B_1 = (0, \mathbf{2}, 3, 5, 9)$.

Breeding: We have developed this new operator to extend the chromosomes to represent complete paths. The main contribution of this operator in the processes of the proposed genetic algorithm is inserting new genes or edges into the chromosomes until they compose complete paths. It is performed on a cell level after the mutation operator. This operator is applied periodically in all iterations of the proposed genetic algorithm.

Breeding operator is performed in the following way:
For each chromosome in the current population:

- Generate a random integer number *pos* from the range $[2 \dots L-1]$, L is the length of the chromosome. The number *pos* indicates the position of the breeding point.
- Identify the edge at the position *pos* and randomly select one edge of its successors edges.
- Then, insert the successor edge at the position $pos+1$ and increase the length of the chromosome by one.

Suppose that $v_1 = (0, 1, 3, 4, 9)$ is a chromosome and $pos = 4$. The edge at position 4 is e_4. The successors of e_4 are e_6 and e_7. We randomly select one of these successors (e.g., e_6) and insert it at position 5. The new chromosome is $v'_1 = (0, 1, 3, 4, \mathbf{6}, 9)$. This chromosome will be extended many times in the next cycles of the proposed genetic algorithm via the breeding operator to obtain a chromosome such as $(0, 1, 3, 4, 6, 8, 5, 9)$ which represents the complete path $\{e_0, e_1, e_3, e_4, e_6, e_8, e_5, e_9\}$.

5.1.7. Elitist

The elitist function enhances the current population by storing one copy of the best member of the previous population. If the best member of the current population is worse than the best member of the previous population it exchanges them, and the best member of the current population would replace the worst member of the current population. After that, it stores the best member of the current population.

5.1.8. The stop conditions

In the traditional *GA* the population would evolve until one individual from the whole set which represents the solution is found. In our case, this condition would correspond to finding group of paths achieving the basis test paths conditions. The evolution stops when a set of individuals has satisfied the required conditions in Section 3. The solution is this set.

The algorithm will stop and the search will end in two cases. The first case when the generated test paths satisfy the conditions of the basis set of paths. The second case when the number of generations reaches the maximum number of generations.

5.2. The overall algorithm of our technique

Our proposed *GA*-based technique accepts as input the program to be tested, the control-flow graph (*CFG*) and the dd-graph (*DDG*) of the program, the *entry* (e_0) and the *exit* (e_k) edges of the dd-graph (*DDG*) and the set (S_i) of successors of each edge e_i. Also, it accepts the *GA* parameters such as population size (*PS*), maximum number of generations (*MG*), and probabilities of the crossover (*PX*) and mutation (*PM*). The algorithm produces a set of basis test paths.

The algorithm generates one basis test path at a time and repeats until the required paths are obtained or the maximum number of generations is exceeded. The overall algorithm is presented in Fig. 2.

5.3. A basis test paths generation tool

We proposed a basis test paths generation tool based on our proposed technique. This tool consists of four main modules:

1) The Analysis Module.
2) Path Generation Module.
3) Feasibility Checking Module.
4) Independency Checking Module.

Fig. 3 shows the overall diagram of our proposed tool. We give more details of these four modules of our tool in the following subsections.

5.3.1. The analysis module

The analysis module has been built to perform the following tasks:

```
/* A GA algorithm to automatically generate set of basis test paths for a given program */
Input:
    The program to be tested P;
    The control-flow graph (CFG) and the dd-graph (DDG) of P.
    The entry e0 and the exit ek edges of the dd-graph (DDG).
    The set Si of successors of each edge ei.
    Population size (PS);
    Maximum no. of generations (MG);
    Probability of crossover (PX);
    Probability of mutation (PM);
Output:
    Set of basis test paths (BTP) for P.
Begin
    Step 1: Initialization
        for i = 1 to PS
            Initialize each path pi ← φ;
        Initialize the set of basis test paths BTP ← φ;
        nRun ←0;
    Step 2: Generate basis test paths
        While (the set BTP is not a basis set)
            Begin
              nRun ← nRun + 1;
              for i = 1 to PS // Create Initial_Population;
                  Put each path pi ← {e0, ek};
              Current_population ← Initial_Population;
              No_Of_Generations ← 0;
              For each individual of the current population do
                Begin
                    Convert the current chromosome to the corresponding path;
                    Evaluate the current path using equation (1);
                    If (the current path is independent path) then
                        Add the current path to the set BTP;
                        nPaths ← nPaths + 1;
                    End If
              End For;
              Keep the best individual of the current population;
              While (the best individual is not independent path and No_Of_Generations ≤ MG) do
                  Begin
                    Select set of parents of new population from members of current population using
                          roulette wheel method;
                    Create New_Population using crossover and mutation operators;
                    Current_Population ← New_Population;
                    For each individual of Current_Population do
                    Begin
                      Convert current chromosome to the corresponding path;
                      Evaluate the current path using equation (1);
                      If (the current path is independent path) then
                        Add the current path to the set BTP;
                        nPaths ← nPaths + 1;
                      End If
                    End For;
                    Apply Elitist function;
                    Enhance the Current_Population by applying the breeding operator;
                    For each individual of Current_Population do
                    Begin
                      Convert current chromosome to the corresponding path;
                      Evaluate the current path using equation (1);
                      If (the current path is independent path) then
                        Add the current path to the set BTP;
                        nPaths ← nPaths + 1;
                      End If
                    End For;
                    No_Of_Generations ← No_Of_Generations + 1;
                  End While;
            End While;
    Step 3: Produce output
        Return set of basis test paths for P, and set of edges covered by each test path;
        Report on uncovered edges, if any;
End.
```

Fig. 2. The overall algorithm.

- Read the program under test.
- Classify program statements and reformats them to facilitate the construction of the program *CFG*.
- Construct the control-flow graph of the reformatted-version of the program (see Fig. 1(b)).
- Construct the dd-graph by reducing the control-flow graph using the REDUCE algorithm [21]. Fig. 1(c) gives an example dd-graph after applying the REDUCE algorithm on the control-flow graph.
- Find the set of successors for each edge in the *DDG*.
- Pass the control-flow graph, dd-graph, and table of successors to the test path generation module.

Table 1 shows the set of successors of each edge in the dd-graph of the example program which is given in Fig. 1.

5.3.2. Path generation module

The path generation module uses the suggested genetic algorithm to generate set of basis test paths. This module

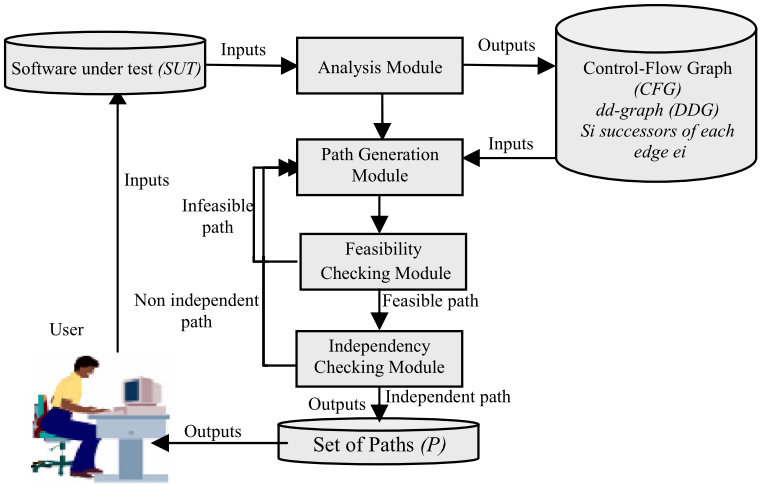

Fig. 3. The block diagram of the proposed tool.

Table 1
The set of successors of each edge in the dd-graph.

Edge	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
Successors	e_1, e_2	e_3	e_3	e_4, e_5	e_6, e_7	e_9	e_8	e_8	e_4, e_5	NA

starts by initializing all paths by the *entry* and *exit* edges of the dd-graph of the program under test. Then it will increment each path by adding new edges until getting a complete path. For example $\{e_0, e_9\}$ is the initialization of any path then the algorithm will add new edges according to the operators of the genetic algorithm until getting a complete path such as $\{e_0, e_2, e_4, e_6, e_9\}$. In case of loop our technique reiterates the loop until covering all edges in the loop such that it selects the edges which are not covered yet. The path generation module passes the generated path to the feasibility checking module to check the feasibility of the generated path.

5.3.3. Feasibility checking module

The feasibility checker collects all the constraints of the generated path. Then it calls a constraints solver such as Sugar [49] to check the consistency of the constraints. If the path is feasible (i.e., constraints are consistent), this module passes the path to the independency checking module, otherwise it passes the path to the path generation module to generate another path. The feasibility checking module can use any feasibility checking method such as the method of Ngo and Tan [50] or the method of Yan and Zhang [51].

5.3.4. Independency checking module

The approach checks the independency of the generated complete path using the independency checking module; if the generated path is an independent path, the algorithm adds this path to the basis set of paths. Otherwise the algorithm ignores this path and starts new cycle to find a different complete path. The algorithm generates test paths one at a time. Therefore, the algorithm will repeat this process until generating the set of basis test paths or satisfying the stop condition.

Table 2
The chromosomes of the initial population.

Chromosome #	Chromosome	Corresponding path
C1	0,9	e_0, e_9
C2	0,9	e_0, e_9
C3	0,9	e_0, e_9
C4	0,9	e_0, e_9

6. A case study

In this section, we introduce a case study to show how the proposed technique can find a basis set of test paths for the example program in Fig. 1. In the following, we will show all the steps of our proposed genetic algorithm. Suppose that the population size (*PS*) is 4.

6.1. Initial population

The initial population is four individuals, each one contains the entry and the exit edges only. Table 2 shows the four chromosomes of the initial populations.

6.2. Evaluation of the current population

Suppose that the current population has the following four chromosomes: $C_1 = (0, 1, 3, 5, 9)$, $C_2 = (0, 2, 3, 4, 9)$, $C_3 = (0, 2, 3, 5, 9)$ and $C_4 = (0, 1, 3, 4, 9)$. To find the fitness value of each chromosome, we convert each chromosome into the corresponding path. Then, we use Eq. (1) to find the fitness value for each chromosome. In each chromosome there are five edges in its corresponding path

Table 3
The fitness values of the current population.

Chromosome #	Corresponding path	Fitness value
C1	e_0, e_1, e_3, e_5, e_9	0.80
C2	e_0, e_2, e_3, e_4, e_9	0.80
C3	e_0, e_2, e_3, e_5, e_9	0.80
C4	e_0, e_1, e_3, e_4, e_9	0.80

Table 4
The roulette wheel.

C #	Fitness value	Relative fitness	Cumulative fitness	r	Parents
C1	0.80	0.25	0.25	0.70	C2
C2	0.80	0.25	0.50	0.20	C1
C3	0.80	0.25	0.75	0.80	C3
C4	0.80	0.25	1.0	0.15	C1

Table 5
The selected parents for crossover.

Parents	R	The selected parents	New individual $pos = 3$
$Pa1 = C2$	0.65	e_0, e_1, e_3, e_4, e_9	e_0, e_1, e_3, e_4, e_9
$Pa2 = C1$	0.82	–	–
$Pa3 = C3$	0.87	–	–
$Pa4 = C1$	0.40	e_0, e_2, e_3, e_4, e_9	e_0, e_2, e_3, e_4, e_9

Table 6
The mutation operation.

Current population	R	New population
e_0, e_1, e_3, e_4, e_9	0.5, 0.1, 0.1, 0.2, 0.1	e_0, e_2, e_3, e_4, e_9
e_0, e_1, e_3, e_4, e_9	0.1, 0.6, 0.2, 0.1, 0.1	e_0, e_1, e_3, e_5, e_9
e_0, e_2, e_3, e_4, e_9	0.1, 0.1, 0.4, 0.4, 0.1	e_0, e_1, e_3, e_4, e_9
e_0, e_2, e_3, e_4, e_9	0.1, 0.2, 0.1, 0.1, 0.1	e_0, e_2, e_3, e_5, e_9

($L = 5$), and four adjacent edges ($d = 4$). Then, $w_i = 1/5$, $i = 1, \ldots, 5$. Therefore, the fitness value of $C_1 = ft(C_1) = w_1 + w_2 + w_3 + w_4 = 4/5 = 0.8$. Table 3 shows the fitness values of the current population.

6.3. Selection

We use the roulette wheel method to select the parents of the next population. The total fitness $F = ft(C_1) + ft(C_2) + ft(C_3) + ft(C_4) = 3.2$. Table 4 shows the computations of roulette wheel.

6.4. Crossover

Suppose that the probability of crossover $PX = 0.80$. Therefore, the expected number of chromosomes is 4, where $PX \times PS = 0.8 \times 5 = 4$. Table 5 shows the new individuals after applying the crossover operator on the selected individuals.

6.5. Mutation

Suppose that the probability of mutation $PM = 0.15$. Therefore, the expected number of mutated cells is 3, where $PM \times L \times PS = 0.15 \times 5 \times 4 = 3$. Table 6 shows the new population after applying the mutation operator.

6.6. Breeding

Suppose that the random number pos = number of adjacent edges in the chromosome = 4. Then, the proposed *GA* inserts the successor edge at position 5. Table 7 shows the results of applying the breeding operator. After computing the fitness of the new population, we get two independent paths $p_1 = e_0, e_1, e_3, e_5, e_9$ and $p_2 = e_0, e_2, e_3, e_5, e_9$. The algorithm will repeat the steps from 3 into 6 to get other two independent paths $p_3 = e_0, e_1, e_3, e_4, e_6, e_8, e_5, e_9$ and $p_4 = e_0, e_1, e_3, e_4, e_7, e_8, e_5, e_9$. We can see that p_1 is a sub-path from p_3 and p_4 as well. Where the cyclomatic complexity of the dd-graph in Fig. 1 $C(DDG) = 10 - 8 + 1 = 3$. Therefore, the set of basis test paths consists of the three paths p_2, p_3, and p_4.

From the case study, the proposed genetic algorithm is effective to generate the basis paths. The set of basis test paths may include some infeasible paths which do not have alternative feasible paths. Except the infeasible paths all the generated basis paths are executable paths and can be run independently of the rest of the code. We can use our proposed genetic algorithm based approach in [45] to find some input data for executing each path in the basis set of paths.

7. Empirical studies

In this section, we describe the empirical studies which we performed to evaluate our proposed technique and to answer the following research questions:

RQ1: How effective is our proposed technique in generating basis test paths?
RQ2: How effective is our proposed technique in generating feasible test paths?

7.1. Empirical setup

7.1.1. Tool implementation

Fig. 3 gives the architecture of the tool of our proposed technique, which consists of four modules: the analysis module, the path generation module, the feasibility checking module and independency checking module. We will use this tool to conduct our proposed empirical studies.

7.1.2. Subject programs

For the empirical studies, we selected a set of common programs of the literature [45–48,52,53]. Some of these programs are usually used in software testing researches such as “triangle classifier” (P#1), “middle value” (P#2), “power x^y” (P#3), and “remainder” (P#4) programs. Programs (P#5–P#9) are synthetic programs of complex structure such as nested ifs, nested loops, and nested loops and ifs. We selected other two complex programs. The first program (P#10) is the “line rectangle classifier” program which determines the position of a line in relation to a rectangle. It has eight real input variables; four of these represent the coordinates of a rectangle and the other four input variables represent the coordinates of a line [52]. The second program (P#11) is the ‘number of days’ program

Table 7
The breeding operation.

Current population	Successors	New population	Fitness value
$e_0, e_2, e_3, \mathbf{e_4}, e_9$	e_6, e_7	$e_0, e_2, e_3, e_4, e_7, e_9$	0.83
$e_0, e_1, e_3, \mathbf{e_5}, e_9$	e_9	$\mathbf{e_0}, \mathbf{e_1}, \mathbf{e_3}, \mathbf{e_5}, \mathbf{e_9}, \mathbf{e_9}$	1.0
$e_0, e_1, e_3, \mathbf{e_4}, e_9$	e_6, e_7	$e_0, e_1, e_3, e_4, e_6, e_9$	0.83
$e_0, e_2, e_3, \mathbf{e_5}, e_9$	e_9	$\mathbf{e_0}, \mathbf{e_2}, \mathbf{e_3}, \mathbf{e_5}, \mathbf{e_9}, \mathbf{e_9}$	1.0

Table 8
Subject programs for the empirical studies.

P#	LOC	NoN	NoE	NoBP	Description
P#1	42	44	49	6	Finds type of triangle (if any)
P#2	37	39	43	5	Determines the middle value
P#3	27	29	31	3	Determines the value of x^y
P#4	38	40	44	5	Determines remainder of x/y
P#5	34	36	39	4	Synthetic of do-while, & ifs
P#6	36	38	42	5	Synthetic of while, for, and ifs
P#7	40	42	44	3	Synthetic of nested do-while and ifs
P#8	21	23	25	3	Sorts and finds smallest and largest in array
P#9	20	22	24	3	Sorts and finds even numbers in array
P#10	180	53	80	25	Finds the position of a line in relation to a rectangle
P#11	253	122	170	100	Finds number of days that are between two dates

which calculates the number of days that are there between two input dates of the current century. It has six integer input variables: three correspond to the initial date and the other three to the final date [52,53].

Table 8 shows details of the programs: The first column, P#, gives program's number; the second column, *LOC*, shows the number of lines of code in the program; the third column, *NoN*, gives the number of nodes in *CFG* of the program; the fourth column, *NoE*, gives the number of edges in *CFG* of the program; the fifth column, *NoBP*, gives the number of basis paths in the program; the sixth column, *Description*, provides a description of the program under test.

7.1.3. Procedure

We conducted the empirical studies as follows.

1. Run the analysis module of our prototype to find the control-flow graph, dd-graph, and table of successors of each node in the dd-graph.
2. Adapt the *GA* parameters to be $MG = 200$, $PS = 10$, $PX = 0.80$ and $PM = 0.15$.
3. Run the path generation module which uses the suggested genetic algorithm to generate one complete path.
4. Run the feasibility checking module to verify the feasibility of the generated path. If it is feasible path, then we pass it to the independency checking module. Otherwise, run the path generation module.
5. Run the independency checking module to verify the independency of the generated complete path. If the generated path is independent path, then we add it to the basis set of paths.
6. Iterate steps 3, 4 and 5 from this procedure to find the remainder of basis paths.

We repeated steps from 2 through 6 of the above procedure 10 times for each subject program. In each run, we re-adjust the parameters of the proposed *GA*. We considered the average of the generated paths in the 10 runs as the number of generated paths for each program (see Table 9).

7.1.4. Threats to validity

There are two main external threats to validity, which are conditions that limit the ability to generalize the results of the studies to a larger population of subjects. First, the subject programs are not large, and we cannot claim that these subjects represent a random selection over the population of programs as a whole. Second, the number of selected paths (population size) is constant. This means that selecting infeasible paths will affect the accuracy of the results. The main internal threats to validity, which can affect the dependent variables is the insufficient information about using genetic algorithm for generating test paths which lead to inadequate evaluation of our proposed technique.

7.2. Results and discussion

Table 9 shows the number of generated paths for each program in each run.

Table 10 shows the actual number of basis paths in each subject programs and the number of generated basis paths by the proposed technique in each subject program. The proposed technique could find the total number of basis paths in six programs of the total eleven programs. For

Table 9
Generated paths for each program through 10 runs.

P#	P#1	P#2	P#3	P#4	P#5	P#6	P#7	P#8	P#9	P# 10	P#11
Run#1	5	4	3	4	4	4	3	3	3	19	71
Run#2	4	3	3	4	4	5	3	3	3	20	78
Run#3	3	4	3	3	4	5	3	3	2	18	68
Run#4	4	3	3	3	4	5	3	3	3	21	76
Run#5	3	4	3	4	3	4	3	2	3	22	70
Run#6	4	4	3	4	3	3	3	2	2	24	82
Run#7	3	4	3	4	3	4	3	3	3	18	73
Run#8	5	4	2	4	4	5	3	3	3	22	70
Run#9	3	3	3	3	4	5	3	3	2	19	76
Run#10	4	3	3	3	4	4	3	3	3	21	80
Average	3.8	3.6	2.9	3.6	3.7	4.4	3	2.8	2.7	20.4	74.4

Table 10
Ratio of generated paths to actual number of paths.

P#	No. of actual basis paths	No. of generated basis paths	Ratio
P#1	6	4	67%
P#2	5	4	80%
P#3	3	3	100%
P#4	5	4	80%
P#5	4	4	100%
P#6	5	5	100%
P#7	3	3	100%
P#8	3	3	100%
P#9	3	3	100%
P#10	25	21	84%
P#11	100	75	75%
Total	162	129	80%

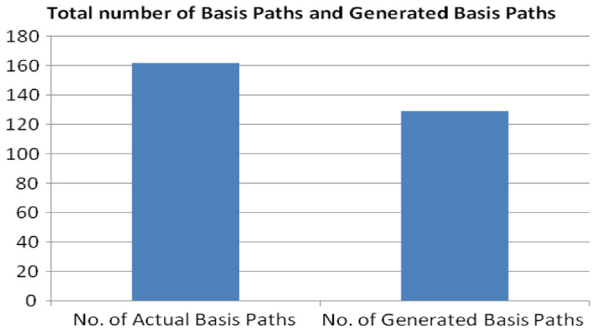

Fig. 4. Actual and generated basis paths.

all programs, the proposed technique could find 80% of the total number of basis paths in all subject programs.

Fig. 4 gives the total number of basis paths in all subject programs which is 162 paths and the number of generated basis paths which is 129 paths. In addition, Fig. 4 shows the efficient of the proposed technique in generating basis test paths and answers the first research question.

Fig. 5 gives the total number of generated basis test paths in all subject programs which is 129 paths and the number of feasible paths which is 118 paths. From Fig. 5, we can see that 91.5% of the total number of generated basis paths in all subject programs is feasible paths. In addition, Fig. 5 shows the efficient of the proposed technique in generating feasible basis test paths and answers the second research question.

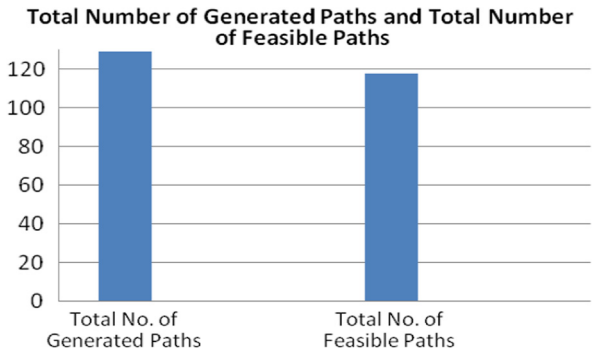

Fig. 5. Total number of generated basis paths and total number of feasible paths.

8. Conclusion

We introduced a new variable length genetic algorithm for automatically generating set of basis test paths which can be used as testing paths in any path testing technique. The length of each chromosome varies from iteration to iteration in the proposed genetic algorithm according to the change in the length of the path. Based on the proposed algorithm, we introduced a new technique for automatically generating set of basis test paths. We presented new definitions for all key elements of the new genetic algorithm such as representation, crossover, and mutation to be compatible with path generation process. In addition, we introduced a new fitness function to evaluate the generated paths. This technique used a method for checking the feasibility of the generated paths. Besides, we suggested a method to verify the independency of the generated paths to append them to the basis set of paths. The proposed technique overcomes the problem of loop in the *CFG*. In addition, we presented a case study to show the efficiency of our new technique. Some experiments to evaluate the effectiveness of the proposed approach are conducted. The results showed that the proposed approach causes a substantial reduction in the path generation effort, and that the proposed *GA* based technique is effective in path generation for path testing. In addition, the proposed technique has the ability to generate feasible paths.

Our future work is directed towards carrying out more complex experiments using some real programs to measure the efficiency of our strategy. We will use our proposed approach for generating the required paths to cover

other control-flow and data-flow criteria. Then, we will use *GA* for generating test data for cover all the basis test paths. In addition, our future work will direct to apply another search-based technique such as Practical Swarm Optimization Algorithms (PSO) to generate test paths for some control-flow and data-flow criteria and compare the efficiency of *GA* with the efficiency of PSO in generating test paths. Also, our future work includes enhancing our approach by including the test prioritization technique to arrange the generated test paths such that the higher priority tests can be executed earlier than lower priority tests. In addition, we will conduct an empirical comparison of genetic algorithm with depth first search algorithm for generating test paths and test data to cover a test coverage criterion.

References

[1] T.J. McCabe, Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-99, NIST, Washington, D.C., 1982.

[2] A. Bagnall, V. Rayward-Smith, I. Whittley, The next release problem, Inf. Softw. Technol. 43 (14) (2001) 883–890.

[3] C.J. Burgess, M. Lefley, Can genetic programming improve software effort estimation? A comparative evaluation, Inf. Softw. Technol. 43 (14) (2001) 863–873.

[4] G. Antoniol, M.D. Penta, M. Harman, Search-based techniques applied to optimization of project planning for a massive maintenance project, in: 21st IEEE International Conference on Software Maintenance, 2005, pp. 240–249.

[5] Z. Li, M. Harman, R. Hierons, Meta-heuristic search algorithms for regression test case prioritization, IEEE Trans. Softw. Eng. 33 (4) (2007) 225–237.

[6] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, in: Special Issue on Software Engineering Using Metaheuristic Innovative Algorithms, Inf. Softw. Technol. 43 (14) (2001) 841–854.

[7] P. McMinn, M. Harman, D. Binkley, P. Tonella, The species per path approach to search-based test data generation, in: International Symposium on Software Testing and Analysis (ISSTA 06), 2006, pp. 13–24.

[8] Moheb R. Girgis, Ahmed S. Ghiduk, Eman H. Abd-Elkawy, An approach for enhancing regression testing using genetic algorithm and data flow analysis, Int. J. Intell. Comput. Inf. Sci. 13 (2) (2013) 115–132.

[9] Ahmed S. Ghiduk, Moheb R. Girgis, Using genetic algorithms and dominance concepts for generating reduced test data, Informatica 34 (3) (2010) 377–385.

[10] Ahmed S. Ghiduk, Automatic generation of object-oriented tests with a multistage-based genetic algorithm, J. Comput. 5 (10) (2010) 1560–1569.

[11] Ahmed S. Ghiduk, Mary Jean Harrold, Moheb R. Girgis, Using genetic algorithms to aid test-data generation for data flow coverage, in: Proceeding of 14th Asia-Pacific Software Engineering Conference (APSEC 2007), December 5–7, 2007, pp. 41–48.

[12] Abdelaziz M. Khamis, Moheb R. Girgis, Ahmed S. Ghiduk, Automatic software test data generation for spanning sets coverage using genetic algorithms, J. Comput. Informatics 26 (4) (2007) 383–401.

[13] B.S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Trans. Softw. Eng. 32 (3) (2006) 193–208.

[14] G. Canfora, M.D. Penta, R. Esposito, M.L. Villani, An approach for QoS-aware service composition based on genetic algorithms, in: Conference Genetic and Evolutionary Computation (GECCO, 2005), 2005, pp. 1069–1075.

[15] M. Cohen, S.B. Kooi, W. Srisa-an, Clustering the heap in multithreaded applications for improved garbage collection, in: 8th Annual Conference on Genetic and Evolutionary Computation, vol. 2, 2006, pp. 1901–1908.

[16] T.M. Khoshgoftaar, L. Yi, N. Seliya, A multiobjective module-order model for software quality enhancement, IEEE Trans. Evol. Comput. 8 (6) (2004) 593–608.

[17] M. Harman, S. Afshin Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Comput. Surv. 45 (1) (November 2012), article no. 11.

[18] K. Mahdavi, M. Harman, R.M. Hierons, A multiple hill climbing approach to software module clustering, in: IEEE International Conference on Software Maintenance, 2003, pp. 315–324.

[19] M. Harman, K. Steinhöfel, A. Skaliotis, Search based approaches to component selection and prioritization for the next release problem, in: 22nd International Conference on Software Maintenance (ICSM 06), 2006, pp. 1063–6773.

[20] M. Harman, R. Hierons, M. Proctor, A new representation and crossover operator for search-based optimization of software modularization, in: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, 2002, pp. 1351–1358.

[21] A. Bertolino, M. Marre, Automatic generation of path covers based on the control flow analysis of computer programs, IEEE Trans. Softw. Eng. 20 (12) (1994) 885–899.

[22] J. Poole, A Method to Determine a Basis Set of Paths to Perform Program Testing (NISTIR 5737), Technical report, U.S. Department of Commerce, National Institute of Standards and Technology, Nov. 1995.

[23] Z. Guangmei, C. Rui, L. Xiaowei, H. Congying, The automatic generation of basis set of path for path testing, in: Proceedings of the 14th Asian Test Symposium (ATS '05), 2005, pp. 46–51.

[24] J. Yan, J. Zhang, An efficient method to generate feasible paths for basis path testing, Inf. Process. Lett. 107 (3–4) (2008) 87–92.

[25] Z. Zhonglin, M. Lingxia, An improved method of acquiring basis path for software testing, in: Proceedings of 5th International Conference on Computer Science & Education, China, 2010, pp. 1891–1894.

[26] D. Qingfeng, D. Xiao, An improved algorithm for basis path testing, in: Proceedings of the International Conference on Business Management and Electronic Information (BMEI), 2011, pp. 175–178.

[27] D. Gonga, W. Zhanga, X. Yaob, Evolutionary generation of test data for many paths coverage based on grouping, J. Syst. Softw. 84 (12) (2011) 2222–2233.

[28] P.M.S. Bueno, M. Jino, W.E. Wong, Diversity oriented test data generation using metaheuristic search techniques, J. Inf. Sci. 259 (20 February 2014) 490–509.

[29] J.R. Bint, Renate Site, Optimizing testing efficiency with error prone path identification and genetic algorithms, in: Proceedings 2004 Australian Software Engineering Conference (ASWEC'04), Australia, 2004, pp. 106–115.

[30] K. Derderian, R.M. Hierons, M. Harman, Q. Guo, Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms, in: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, ACM, Washington, D.C., USA, 2005, pp. 1081–1082.

[31] A.S. Kalaji, R.M. Hierons, S. Swift, Generating feasible transition paths for testing from an extended finite state machine (EFSM), in: 2nd IEEE International Conference on Software Testing, Verification, and Validation (ICST), 2009, pp. 230–239.

[32] A.S. Kalaji, R.M. Hierons, S. Swift, Generating feasible transition paths for testing from an extended finite state machine (EFSM) with the counter problem, in: 3th IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 232–235.

[33] T. Yano, E. Martins, F.L. de Sousa, Generating feasible test paths from an executable model using a multi-objective approach, in: 3th IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 236–239.

[34] P.R. Srivastava, K. Baby, G. Raghurama, An approach of optimal path generation using ant colony optimization, in: Proceedings of IEEE Region 10 Conference on TENCON, 2009, pp. 1–6.

[35] A.S. Ghiduk, A new software data-flow testing approach via ant colony algorithms, Univ. J. Comput. Sci. Eng. Technol. 1 (1) (2010) 54–62.

[36] S.S. Babu Lam, M.L. Hari Prasad Raju, M. Uday Kiran, Ch. Swaraj, Praveen Ranjan Srivastav, Automated generation of independent paths and test suite optimization using artificial bee colony, Proc. Eng. 30 (2012) 191–200.

[37] E. Alba, F. Chicano, M. Ferreira, J.A. Gómez-Pulido, Finding deadlocks in large concurrent java programs using genetic algorithms, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO '08), 2008, pp. 1735–1742.

[38] M.M. Naoghare, V.M. Deshmukh, Comparison of parallel genetic algorithm with depth first search algorithm for solving verbal arithmetic problems, in: Proceedings of the International Conference & Workshop on Emerging Trends in Technology (ICWET '11), 2011, pp. 324–329.
[39] K. Gallagher, M. Sambridge, Genetic algorithms: A powerful tool for large-scale nonlinear optimization problems, J. Comput. Geosci. 20 (7–8) (1994) 1229–1236.
[40] B.G. Ryder, Constructing the call graph of a program, IEEE Trans. Softw. Eng. SE-5 (3) (1979) 216–226.
[41] M.J. Harrold, G. Rothermel, Performing data flow testing on classes, in: 2nd ACM-SIGSOFT Symposium on the Foundations of Software Engineering, 1994, pp. 154–163.
[42] S. Haiying, Method and Practice of Software Testing, China Railway Publishing House, 2009.
[43] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, ISBN 0472084607, 1975.
[44] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd edition, Springer, 1999.
[45] Ahmed S. Ghiduk, Moheb R. Girgis, Using genetic algorithms and dominance concepts for generating reduced test data, Informatica 34 (3) (2010) 377–385.
[46] C.C. Michael, G.E. McGraw, M.A. Schatz, Generating software test data by evolution, IEEE Trans. Softw. Eng. 27 (12) (2001) 1085–1110.
[47] R.P. Pargas, M.J. Harrold, R.R. Peck, Test data generation using genetic algorithms, J. Softw. Test. Verif. Reliabil. 9 (1999) 263–282.
[48] M.R. Girgis, Automatic test data generation for data flow testing using a genetic algorithm, J. Univers. Comput. Sci. 11 (5) (2005) 898–915.
[49] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, Mutsunori Banbara, Compiling finite linear CSP into SAT, Constraints 14 (2) (June 2009) 254–272.
[50] Minh Ngoc Ngo, Hee Beng Kuan Tan, Heuristics-based infeasible path detection for dynamic test data generation, J. Inf. Softw. Technol. 50 (2008) 641–655.
[51] Jun Yan, Jian Zhang, An efficient method to generate feasible paths for basis path testing, Inf. Process. Lett. 107 (2008) 87–92.
[52] E. Diza, J. Tuyaa, R. Blancoa, J. Doladob, A tabu search algorithm for structural software testing, J. Comput. Oper. Res. 35 (2008) 3052–3072.
[53] Calculate the number of days in-between two dates, http://www.dreamincode.net/forums/topic/177600-calculate-the-number-of-days-in-between-two-dates/, last visit 18-1-2014.