Journal of AI and Data Mining
Vol 6, No 2, 2018, 375-385
DOI: 10.22044/JADM.2017.5015.1603

Optimizing Cost Function in Imperialist Competitive Algorithm for Path Coverage Problem in Software Testing

M. A. Saadatjoo and S. M. Babamir*

Department of Computer Engineering, University of Kashan, Kashan, Iran

Received 22 November 2016; Revised 10 April 2017; Accepted 27 July 2017
**Corresponding author: babamir@kashanu.ac.ir(S. M. Babamir).*

Abstract
The search-based optimization methods have been used for the software engineering activities such as software testing. In the field of software testing, search-based test data generation refers to the application of meta-heuristic optimization methods to generate the test data that cover the code space of a program. Automatic test data generation that can cover all the software paths is known as a major challenge.
This paper establishes a new cost function for automatic test data generation, which can traverse the non-iterative paths of the software control flow graphs (CFGs). This function is later compared with similar cost functions proposed in the other articles. The results obtained indicate the superior performance of the proposed function. Another innovation proposed in this paper is the application of the Imperialist Competitive Algorithm (ICA) in automatic test data generation along with the proposed cost function. Automatic test data generation is implemented through ICA as well as the genetic algorithm and particle swarm optimization algorithm for three software programs with different search space sizes. These algorithms are compared with each other in terms of the convergence speed, computational time, and local search. The test data generated by the proposed method achieved better results than the other algorithms in finding the number of non-iterative paths, convergence speed, and computational time with growing the searching space of the software CFG.

Keywords: *Software Testing, Imperialist Competitive Algorithm, Test Data Generation, Control Flow Graph, Program Complexity, Path Coverage.*

1. Introduction

One of the most important tasks in the process of software quality assurance is software testing, which is too expensive. According to the literature, nearly one-third of the software errors can be avoided by relying on the software testing methods [1]. Different methods are used for software testing. Among others, search-based testing is an effective method for testing a program if it is possible to cover the execution space of the program. It refers to the use of a meta-heuristic optimizing search method to automate test case generation [2]. In the search-based testing methods, a test dataset is provided as a vector of the required values to traverse different execution paths of a program so as to demonstrate the maximum software fault. A test dataset includes a vector with sufficient values to implement the corresponding software. A test data vector is ideal if 1) each of its elements is necessary (i.e. not redundant) and 2) its elements are sufficient (i.e. no other elements are required). These two conditions state that the vector should include the same number of elements as the program execution paths so that inputting each of its elements can lead to the execution of a different program path. If the fist condition is not met, the test will not be complete but if the second one is not met, unnecessary cost will be incurred.
The execution paths of a program, known as the search space, are demonstrated by a Control Flow Graph (CFG). This is a graph that is built according to the program code. The more branches the graph has, the more complex search space it has. Such a complexity has a direct

relationship with program testability (see Section 3).

In this article, a new cost function is introduced. This cost function is composed of the cost function introduced in [3] and the recommended parameter. The value of the cost function in the recommended method is obtained by the maximum path coverage, not by choosing the repeated paths of the Control Flow Graph (CFG). To evaluate the new function, the problem of search-based testing in [3] is solved by the recommended cost function. The results obtained show that the traverse of non-iterative paths is more than the one in [3].

Choosing the test data from a software search space is very complex. To overcome this problem, the automatic test data generation techniques can be of help using complete algorithms [4 and 5].

One of the recent evolutionary algorithms that has drawn the attention of researchers in search-based issues is the Imperialist Competitive Algorithm (ICA) [6]. This algorithm has led to better results in different applications in comparison with the Genetic Algorithm (GA) and the Particle Swarm Optimization (PSO) algorithm [7-9]. In [10], these three are compared with one another according to the local search parameters, convergence speed, and computational time. The results obtained showed that ICA is more efficient than the other algorithms. In [11], a hybrid meta-heuristic algorithm based on imperialist competition algorithm is introduced. Their method showed that ICA method has better results in finding global optimum and search speed. In [12], it was tried to solve a discrete Traveling Salesman problem using ICA. The results showed the high capability of this algorithm in solving discrete problems. Since the problem of automatic software test data generation by CFG is of a discrete nature and due to the priorities of ICA in [10], ICA was used in the present work. The aim of this work was to show the efficiency of ICA as an approach recommended versus GA and PSO algorithm in solving the problem of maximum path coverage of search-based testing.

Although ICA has been applied for software cost estimation aimed at software project management [13-15], it has not yet been used for the automatic test data generation software.

Using ICA, we generated the test data that covered more program paths than GA and PSO algorithm, and the generated data was closer to ideal. This work is the first attempt to use ICA through the recommended cost function so as to generate the automatic test data whose efficiency is determined against other related approaches such as GA and PSO algorithm. According to this experience, the efficiency and capability of the approach was determined based upon its nearness to the ideal data, local research, convergence speed, and computational time. In this work, the recommended algorithm was used to generate the test data for four programs with low, medium, and high degrees of complexity, and the results obtained were evaluated to determine the efficiency of the algorithm.

This paper is organized as what follows. The program CFG is introduced in Section 2. Section 3 deals with the related works. Section 4 addresses ICA (i.e. the proposed algorithm) for the automatic test data generation. The proposed cost function and its evaluation are discussed in Section 5. The use of ICA in software search-based testing is explained in Section 6. Section 7 draws conclusions and proposes directions for future research works.

2. Control flow graph (CFG)

A CFG is a graph in which each node contains one or more successive program statements. It has a start and an end node, and its edges denote the control flow between the program statements. In this graph, the branch points indicate conditional statements [16]. A path starts from a start node and ends with an end node. Figure 2 shows CFG for the program in figure 1 (the number of nodes in figure 2 indicates the number of statements in figure 1.

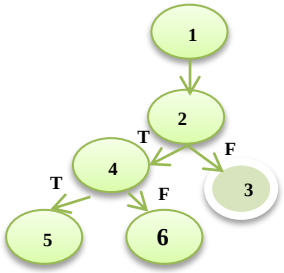

Figure 1. CFG of MAX program.

This graph has six nodes, each of which includes one or more program statements in figure 2, and seven edges each of which denotes the control between the statements. The basic paths are as what follow. The rest of the paths are constructed out of the basic paths, which are three. It should be noted that path 1 will not be evaluated by any test data. Paths such as this are called the infeasible software paths.

a) 1- 2- 3
b) 1-2-4-5-6-2-3
c) 1-2-4-6-4-3

Generation of an adequate test data for maximum CFG path coverage is our main concern in this

paper. As the CFG branches increase, the CFG structure becomes more complicated for searching because the number of paths increases too, and thus finding adequate test data that can traverse all the paths becomes difficult. In fact, the number of elements in the search space of source code paths is determined by the number of conditions. This is called program complexity or the McCabe number [17].

In order to automatically generate the test data, the search space for the program must be specified. This space, which shows the structure of the program, is obtained from CFG. The search in this space is carried out in the following three ways: 1) Searching the space for graph nodes in which each node represents an instruction for the program 2) Searching the space for graph edges in which each edge represents a branch in the program 3) Searching the graph paths in which each path is a set of nodes and edges and entails the beginning to the end of the graph. The test data selected as the answers are only those that are able to successfully traverse the maximum part of the search space or the paths of the program CFG.

```
1. void MAX (int x[],int n) {
2.     int max,i=1;
3.     max=x[0];
4. `   while(i<n){
5.     if(x[i]>max)
6.         max=x[i];
7.     i++;
8.       }
#9.    cout<<max;
#10.   }
```

Figure 2. MAX program.

For a software in which the whole program is run in one module, a CFG can be easily depicted so as to determine the number of software paths. However, in a software whose running process includes several intricate modules, depiction of CFG of the whole software is not an easy task. In such structures, first, a CFG is created for each module separately, and the number of paths is determined. Next, based upon the number of paths in each module, the paths in the whole software are estimated. This method was used in the present study to approximate the number of paths.

3. Related works

About 59% of the literature on software engineering is about software testing [2]. The main idea to use methods of evolutionary algorithms for search-based testing is to generate a set of test data that are partly ideal.

3.1. Application of PSO

The PSO algorithm was first introduced in [18] and inspired by a swarm of birds looking for food. In [19], the authors have tried to generate the test data for a program that merges two arrays using PSO. To do so, they generated six methods for traversing paths. However, their experience was of use for simple problems with low complexity, and no evaluation was provided for programs with medium and high degrees of complexity. In this paper, the studied issues were not so complex, and the results obtained were not compared with other approaches. However, the efficiency of the recommended approach was compared with that of the PSO algorithm approach.

3.2. Application of GA

GA can be applied to resolve the optimization issues [20]. The algorithm is also applied to software automatic test data generation for the purpose of path coverage. In generating the test data using this algorithm, a chromosome plays the role of the input vector of the test data, each of which functions as a gene.

In [21-24] have proposed principles and rules for using this algorithm to automatically generate the test data. They generated the test data according to the proposed algorithm for a couple of programs. This algorithm has been used in [22] to generate the test data based on a dynamic method. The proposed method was compared with GA for program branch coverage as the search space. Finding the ideal test data that is able to cover more space is one of the properties of this dynamic method.

Keyvanpour and Homayouni [3] have tried to use a combination of evolutionary and local search methods to generate the test data. Using the Memetic algorithm and a local search method called ‘hill climbing’, at each stage of test data generation, they tried to reduce the time spent for finding the suitable tests. The cost function that they used was a combination of three parameters including neighborhood, closeness to the border, and branch coverage. At each stage of their proposed algorithm, the generated data was inputted to a neural network for evaluation. They applied their method to a triangle program, and using the local search algorithm, they were able to reduce the number of algorithm iterations to an acceptable level, which made the convergence faster. However, this method was not implemented to solve problems with medium and high degrees of complexity.

Many researchers have used GA to generate the test data for different types of coverage including

branch coverage [22], condition-decision coverage [23], path coverage [24-26], and multiple-path coverage [27-29].

As for the present article, its focus has only been placed on path space coverage. In each of the above-mentioned evolutionary methods, a program with low complexity is initially introduced for generating the test data. In the second step, the evolutionary algorithm parameters are set according to the desired program. In the third step, the test data that can cover the search spaces of condition, path, branch, and instructions are generated.

Ahmed and Hermadi [27] have investigated their test data generation method to cover the path space in a program of low complexity. Their desired criterion was the amount of the generated data for traversing one path and then traversing all the paths of the program. Their proposed method failed to generate the least test data with the ability to traverse the most possible paths.

Using GA, Li and Yeh [29] have generated the test data to cover the path space through relational algebra. They examined the CFG paths when coverage of only one path was desired. However, they did not discuss the effectiveness of this method when it was aimed at traversing all the paths. In their method, the focus of each stage of implementing the proposed algorithm was only on traversing one path, and this prevented the generalization of their method to solve the whole problem space. Another drawback of this method is that they used a criterion for comparing the time regardless of the fitness value showing the extent of path coverage.

Zhang and Gong [31 and 32] have compared the speed of implementing their proposed algorithm in generating the test data on two programs (array search and bubble sort) with low complexity. In the current work, this parameter was used to display and evaluate the implementation speed of the proposed method together with the quality of the generated test data (which is dependent upon a fitness function).

Harman and McMinn [33] have evaluated some (local and general) search methods in program tests, and concluded that a method that combines the local and general search can greatly contribute to the optimal generation of the test data, which can properly cover the search space. Their evaluation was based upon the complexity of sample programs, all of which were simple or with low complexity. It was not determined in their work whether their method could account for the time when the problem space expands.

Mansour and Salame [34] have used GA for automatic generation of the test data in problems that work with accurate and real data. Their method was limited to certain data types, and no comparison was made for the amount of path coverage. However, the test data type had no effect on our proposed method since it was determined with respect to the desired problem.

In addition to GA and PSO algorithm, the Artificial Bee Colony (ABC) algorithm is a search-based method used for automatic generation of the test data in order to cover the path space [35]. However, its effectiveness has not been proved for path coverage of programs with high complexity and large search space.

4. Imperialist competitive algorithm (ICA)

In ICA, countries are introduced as problem results (i.e. test data). The characteristics of each country are the desired test data to cover the search space. In this algorithm, in terms of ideal test data selection, powerful countries are opted through a cost function as used in this algorithm. Some countries are selected as imperialists and some others as colonies. Like many other evolutionary algorithms, at the beginning of the algorithm implementation, countries (test data) are randomly selected in the whole problem space. Based upon the power of each country, some are randomly selected as imperialists (i.e. test data that traverses new and more paths compared with the other data in the search space) and some others as colonies. The imperialists and colonies are selected independently. An imperialist and its colonies are called an *empire*. Imperialist competition means the attempt of an imperialist to attract maximum colonies from other empires.

There are two conditions for this algorithm including 1) the algorithm continues based on the described cases until one or more empires remain with regard to the problem (2) the algorithm runs to a certain number of rounds. One may shortly refer to these two as the final condition and convergence. In the case of automatic generation of the test data in this work, each empire will be considered as a test data vector. Each vector represents the test data that traverse at least one new path. The ultimate solution will be the most powerful imperialist, indicating the best set of test data vectors that traverse a maximal number of program paths.

The first step in forming an empire is to establish countries that are randomly placed throughout the space. The structures of the countries vary according to the software under test. In general, a

problem with a test vector having n elements is shown as (1):

$$Country = [num_0, num_1, ..., num_n] \quad (1)$$

Another parameter that plays a crucial role in attracting the countries and developing an empire is the power of each imperialist within the empire or simply the power of each empire. Using the power of an empire, it can be specified which country is shared at the time of competition. To calculate the power of each empire, as mentioned in [3], the ultimate power of that empire is taken into account.

The following section outlines the proposed cost function and its application in ICA for automatic test data generation.

5. Proposed cost function

The first challenge in solving problems by evolutionary algorithms is to design a proper cost function. This aims at generating a minimum test data to traverse maximum paths of a software CFG. Thus a proper cost function should be designed.

The recommended cost function in this paper is composed of the cost function introduced in [3] and a new parameter. For a better understanding of the recommended cost function, all parameters will be discussed.

5. 1. Path traverse probability

A test vector consists of some test data, each of which traverses a path of the graph. The possibility of traversing this path by test data t_i from vector T is:

$$L(Path(t_i)) = \prod_{j=1}^{k} P_j(\text{for all conditions in path}(t_i)) \quad (2)$$

In (2), k is equal to the conditions in which test data t_i is traversed through its path.

Generally, by taking the following steps, it is possible to obtain the probable parameter of traversing a path by a test vector.

- Obtaining a path that traverses the test vector.
- Using CFG of the software to determine the set of conditions that the test data traverse through and writing an algebraic expression among its conditions.
- Normalizing the algebraic equation by moving one side of this equation to the other side, for example, changing a < b to a-b < 0.
- Calculating the probability of generating a normal algebra.
- Calculating the probable parameter of the path by the data of a T test vector, as follows:

$$P(T)=1-\left(\left(1-L\left(Path\left(t_1\right)\right)\right)*\left(1-L\left(Path\left(t_2\right)\right)\right)*...\right) \quad (3)$$

$L(P(t_1))$ is the probability of selecting a path that is traversed by the test data t_i. If each path has more than one condition, the probability of the test data not to select that path is equal to 1-$L(P(t_1))$. In addition, a test vector is composed of a combination of several test data. Thus this amount is calculated for each test data of one test vector.

5.2. Closeness to boundary values

Experience has shown that test cases close to boundary values are more likely to find program error 4 [4]. The closeness to the boundary is shown by $N(T)$.

Close to the boundary values for a test vector T. This parameter for a test vector is calculated by Eq. 4:

$$N(T) = N(t_1) \times N(t_2) \times ... \quad (4)$$

where, t_i is the i^{th} test data of test vector T, and $N(t_i)$ is close to the boundary value for t_i that is calculated as follows:

- Obtaining a path that traverses the test vector.
- Using CFG of the software to determine the set of conditions that the test data traverse through and writing an algebraic expression among its conditions.
- In each algebraic expression, all comparison operations {<, >, ≤, ≥, =} change to {=}.
- Moving the right side of each algebraic expression to the left side to find the boundary value of that expression.
- Calculating the parameter of closeness to the boundary values for each input test data, as follows:

$$N(t_i) = 1 - \left|(t_i)/(Max\_Value\_Size)\right| \quad (5)$$

where, t_i is the test data input, and Max_Value_Size is the domain of the test data. The closeness to the boundary values is calculated for each existing data on the test vector, and then each value is multiplied by the others to obtain the probability of closeness to the boundary borders of the test vector.

5.3. Edge coverage

Edge coverage means the percentage of edges in CFG covered by the test data vector. Edge coverage is computed using (6), where e is the number of edges traversed by the test vector T, and E is the total number of CFG edges:

$$D(T) = e / E \quad (6)$$

The aim of this paper was to generate an ideal test data that could cover the maximum paths of a graph. To gain this goal, the parameter of non-iterative path coverage was selected.

5.4. Non-iterative path coverage

To compute the number of paths traversed by a test vector, it is required to determine whether the traversed paths are non-iterative since the test data that do not generate a new path are practically useless. For this purpose, we introduced the concept of Evaluator, and, using this, we generated vectors that created non-iterative paths.

5.4.1. Evaluator

Obtaining the non-iterative traversed paths in CFG by a test data vector was an important part of the present study. An evaluation matrix is used to determine the non-iteratively traversed paths by the test data. Assume that the conditions of a CFG are C_p. Therefore, there are $2C_p$ corresponding algebraic expressions (one corresponding algebraic expression based upon a true condition and one corresponding algebraic expression based upon a false condition). Each Evaluator is an array of paths at the length of $2C_p$ in which each element is a corresponding algebra expression. For every test data, there is an Evaluator. From the beginning, this array is initialized with a zero, i.e. the test data does not traverse any path. When an edge is traversed by the test data, an algebraic expression is selected from $2C_p$ of them, whose corresponding value in the Evaluation array has the value of 1.

After running the program by a test vector, all the Evaluators are calculated based on decimals. If the decimal numbers generated by the Evaluators are unique, non-iterative paths will be traversed. In other words, iteration of each number indicates the traverse of an iterative path. Thus the Evaluator obtained is unique in every path of CFG.

If the number of existing test data is equal to that of vector V_p, then we will have the matrix $Eval_{V_p \times 2C_p}$ in which every row is an array corresponding to the test data.

In graphs with loops, it is possible to make the following equation after generating an Evaluation matrix:

$$Eval[k] = Eval[i] \vee Eval[j] \quad (7)$$

where, $Eval[i]$ is the i^{th} row of the Evaluation matrix. Here, the k^{th} row of the Evaluation matrix is obtained from the OR logical actions between the i^{th} and j^{th} rows of the matrix. It should be noted that the test data that generate these Evaluators can traverse a given path more than one time.

The number of lines in the Evaluator obtained from other lines through the Union logical operation is displayed by cnt_Eval. This variable indicates the number of non-iterative paths traversed by the test data. The non-iterative path coverage parameter is computed by cnt_Eval, here referred to as $W(T)$ (Equation 8). The parameter is presented in this paper for the first time:

$$W(T) = \frac{cnt\_Eval}{N_p} \quad (8)$$

where, N_p represents the number of total estimated paths in the graph.

Simply, the computational steps of $W(T)$ are as follow:

- Using CFG to determine the set of conditions that the test data can traverse.
- Generating the $Eval_{V_p \times 2C_p}$ Evaluation matrix in which V_p is the number of test data of a test vector, and C_p is the number of conditions in CFG.
- Setting the Evaluation matrix according to the input test vector.
- Calculating the decimal number of each row.
- Determining the number of non-iterative paths using *cnt_Eval*.

According to the above criteria, the proposed cost function is:

$$F(T) = (P(T) + N(T) + D(T) + W(T))/4 \quad (9)$$

5.5. Evaluating proposed cost function

To generate a test data with high quality, the three parameters path traverse probability, closeness to boundary values, and edge coverage were linearly integrated. As in [3], a cost function was used with the non-iterative path coverage parameter for the purpose of high-quality test data generation. To determine the effect of the non-iterative path coverage parameter, a triangle program was used as the base example to show the automatic test data generation.

Two empirical tests were carried out to evaluate the proposed cost function. In the first one, according to [3], 25 test vectors, each with five test data and three elements (all integer numbers between -32678 and 32677), were involved in the test. The maximum path coverage of CFG

depends on the number of test data in a test vector.

In the second test, 90 test vectors, each with 18 test data and three elements, were selected. Since the triangle program had 18 paths and the aim of the article was to traverse the maximum paths of CFG of a program, every test vector had 18 test data. These two tests were carried out with the cost function in [3], the cost function proposed in the present work, and using GA in equal situations. The results of averaging 30 independent tests are shown in table 2.

As shown in this table, using the non-iterative path parameter in the proposed cost function, GA can find all the paths with five test data. This is while, by using the cost function [3], only three paths are traversed. In addition, when the test data increase, the proposed cost function is more successful in finding more paths than the cost function is in [3]. Using the proposed parameter in the cost function prevents generating iterative test data and leads to finding the data that traverses new paths. This is in contrast to the cost function of [3], where only ideal data are important, and iterative or non-iterative data is not important. Finding more paths leads to finding more errors in the software. The aim of this work was to find an ideal test data to traverse the CFG paths through introducing a parameter as explained above.

Owing to its efficacy, the proposed cost function was used to solve a search-based path-coverage problem with ICA.

6. Using ICA in search-based testing

This section addresses the automatic test data generation for path coverage in three software programs with moderate and high degrees of complexity. Figure 3 shows the pseudo-code of generating the test vectors to cover the search space of the software paths using ICA. According to figure 3, after executing the proposed method, countries will compete with each other to gain ideal test data. After the power of each test vector is calculated, ICA comes to be of use.

To test the software comprehensively, a test vector including all the aspects is required. The number of test data in a test vector is proportional to the paths of the software under test.

6.1. Setting-up parameters of ICA

In this section, the parameters required for running ICA are discussed.

6.1.1. Empire formation

As stated in Section 5, an Empire is a set of test vectors. Empire formation means to generate a set of test vectors in all the spaces of a CFG. In the automatic test data generation of a software program and after obtaining the space of the software CFG and determining the structure of each vector as a member of an Empire, we have to determine the role of the test vectors as an empire or a colony. Vectors with a higher cost function are an Empire, and the other vectors (countries) are selected as colonies.

1- Define Fitness
- Identify condition nodes
- Calculate edge coverage probability
- Calculate closeness to boundary
- Calculate branch coverage
- Calculate path coverage
- Calculate finesse value

2- Define IP according to paths

3- Tune ICA parameters
- Define test-data pattern
- Define assimilation policy

4- Generate test data
- Run program (p) for each test data
- Calculate Empire's power
- Colony absorption
- Eliminate powerless Empire
- Check stop conditions

Figure 3. Pseudo-code of generating test data using ICA.

Table 2. Evaluation of proposed cost function in Triangle program using GA.

Cost function	Test number	Test data number	Traversed paths
Cost function introduced in [3]	First	5	3
	Second	18	6
Proposed cost function	First	5	5
	Second	18	14

6.1.2. Assimilation policy

In order to achieve a powerful empire, an imperialist tries to assimilate colonies using a specific assimilation policy. To understand the process, this policy has to be explained first. According to the presented cost function, after calculating the power of each empire and before entering the competition, each imperialist tries to assimilate more colonies. At each step of the algorithm, the imperialist, quite randomly, substitutes 10% of its arrayed elements, as part of the test vector, with identical colony elements. Through this method, the speed and power to reach an optimal answer significantly increases.

What follows is an example of replacing the first element of an imperialist with that of its colony in a triangle problem. In this example, the first epoch of assimilation policy is illustrated. The first element of the empire is replaced with the first element of the colony. This replacement continues to the last element. It means that after n epochs of ICA, all the remaining colonies in the empire look completely like the imperialists. The selected assimilation policy for all the tested programs is as (10):

$$\begin{aligned} &Before\ assimilation: \\ &Country \quad = num_1\ num_2\ num_3 ... num_n \\ &Imperialist = C_1 \quad C_4 \quad C_2\ ...\ C_5 \\ &After\ assimilation: \\ &Country \quad = num_1\ num_2\ num_3 ... num_n \\ &Colony \quad = C_2 \quad C_3 \quad C_4\ ...\ C_1 \\ &\qquad \Downarrow \\ &Colony \quad = C_1 \quad C_3 \quad C_4\ ...\ C_1 \end{aligned} \tag{10}$$

6.1.3. Final conditions and answers

Like other evolutionary approaches, termination conditions should be defined. In ICA, the goal is to have a single empire as the final condition. In this situation, the test data in the imperialist vector is the final answer. The convergence condition means that the algorithm repeats for a pre-determined number of times, and the cost function value does not change in any iteration; therefore, ICA will end. If the algorithm is not terminated after specific iterations, the Empire that has a higher cost function than the others is the answer. Before showing the efficiency of ICA in search-based testing, it is necessary to introduce the set of tested software and define the structure of the tested vectors (i.e. countries) that are required for traversing the paths.

6.2. Case studies

Four programs (Table 3) were used to evaluate the effectiveness of the proposed method. Each program was selected according to the complexity of the condition structure and the search. Triangle program, as one of the common software programs of generating automatic test data, proved to be proper with which to introduce the proposed method. Two programs from the Siemens database were selected to evaluate the proposed method in big industrial spaces. In order to automatically generate the test data for these applications using the proposed method, the structure of the test vector was initially identified for each of these programs, and CFG was created for each program. Then based on that, the corresponding Evaluation matrix was created. Finally, a brief description was provided for each software program and for the structure of the test vectors.

6.2.1. Triangle program

Triangle program is one of the common programs that researches use a basic benchmark program.

Table 3. Properties of selected programs.

Program	Task	#paths	Type of complexity	Line of Code (LOC)
Triangle	Determining triangle type	18	moderate	30
Schedule	Scheduling tokens by users	173	high	410
Print-token	Get and print a token out of queue	158	high	563

The code of the triangle software used is in [3], and its CFG is generated. This program has six conditions and 12 algebraic expressions. If a test vector is assumed to consist of four tests, i.e. T={(1,2,5),(5,5,5),(5,4,5),(25,2,7)},the Evaluation matrix will be:

$$Eval = \begin{bmatrix} 0\,1\,0\,1\,0\,1\,1\,0\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,0\,0 \\ 1\,0\,0\,1\,0\,1\,0\,1\,0\,1\,1\,0 \\ 1\,0\,0\,1\,1\,0\,1\,0\,0\,0\,0\,0 \end{bmatrix}$$

According to the evaluation matrix:

- The 1^{st} test data will not evaluate the condition related to lines 2, 3, and 4 but it can evaluate the condition of line 5 that is in the scalene triangle group.
- The 2^{nd} test data will not evaluate the condition related to lines 2, 3, 4, and 5 but it can evaluate the condition of line 9 that is in the equilateral triangle group.
- The 3^{rd} test data will not evaluate the condition related to lines 3, 4, 5, and 9 but it can evaluate the condition of lines 2 and 12 that are in the isosceles triangle group.
- The 4^{th} test data will not evaluate the condition related to line 3 but it can evaluate the condition of lines 2, 4, and 5 that are in the non-triangle group.

In ICA, each test vector is considered as a country, and defined as (11).

$$\begin{aligned} &Country = \{(num_{11}, num_{12}, num_{13}), \\ &(num_{21}, num_{22}, num_{23}), ..., \\ &(num_{K1}, num_{K2}, num_{K3})\} \end{aligned} \tag{11}$$

In this equation, each country has some test data that are the inputs of the problem. In the triangle software, each test data has three integers that are the lengths of the triangle lines. According to the triangle software paths, the maximum path that a vector can traverse is 18. Thus K = 18, where K is the number of test data used in the test vector.

6.2.2. Schedule program

This program can be taken from the Siemens database as an industrial program with the aim of scheduling the sent items in a network. Moreover, the switch-case control condition makes this software different from the other software programs. The features of this software are presented in table 3, and the program code is available in [36]. CFG is designed for each module of this software, and its paths are extracted. The maximum number of the paths of this software is 173 with respect to CFG of its modules. A country (i.e. test vector) is defined as in (12), where the number of elements in the test data is 23, and the number of paths equals $K = 173$.

$$Country = \{(num_{1,1},...,num_{1,23}), (num_{2,1},...,num_{2,23}),..., (num_{K,1},...,num_{K,23})\} \quad (12)$$

6.2.3. Print-token program

This program is extracted from the Siemens database. CFG is made for this software, and its paths are extracted. With respect to CFG of each module, the maximum number of paths in this software is 158. In this work, selection of this software is due to its having more functions and nested call nodes. For this software, a country (or a test vector) is defined as (13). This software has 32 inputs, so the number of elements in each test data is 32, and the paths are calculated as k = 158.

$$Country = \{(num_{1,1},...,num_{1,32}), (num_{2,1},...,num_{2,32}),..., (num_{K,1},...,num_{K,32})\} \quad (13)$$

6.3. Evaluating efficiency of ICA for path coverage in search-based testing

To evaluate the efficiency of ICA over GA and PSO Algorithm, the cost function proposed in Section (5) and the parameter introduced in [10] are used. The corresponding criteria are presented in the following:

a. Convergence speed: this criterion is the number of repetitions of algorithms to get a final condition and find a final answer. In this paper, the final answer is a test vector that has a higher cost function.

b. Computational time: this criterion is the time of executing an algorithm at specified intervals.

c. Local search: the maximum non-iterative path in a search space is introduced using this criterion.

The different nature of the softwares introduced in part (7) can be of use to investigate the ability of the selected algorithms in meeting the above criteria. For GA and PSO Algorithm, each gene and particle is a country introduced in part (7). The selected generations and the final conditions are the same in the three evolutionary algorithms.

Table 4 shows the general settings of algorithms for running each of the programs introduced in table 3. To have a fair comparison between the evolutionary algorithms, a standard implementation should be ensured.

Table 4. General settings of algorithms.

Program	Triangle	Schedule	Print-token
Test data length	3	23	32
Path number	18	173	158
Population	2000	5000	5000
Maximum repetition	200	500	500

The parameters ICA, GA, and PSO are presented in table 5 based on [10], [37], and [38], respectively. Random initializations are selected in generations. The proposed cost function is used in the implementation of evolutionary algorithms shown in table 6. This means that the results of ICA are obtained using the proposed cost function.

The average results of 30 sequential executions for four programs are shown in table 6. In this table, the second column shows the paths traversed by the ideal test vector. The third column shows the steps of executing the algorithm to reach the final conditions.

The fourth column shows the execution time of the algorithm to reach the optimal answer. Among all the available paths in CFG, the fifth column shows the traverse percentage of the algorithm. The quality of the generated test data through the cost function in each algorithm is shown in the last column of the table. The following results are obtained from table 6:

a. In problems with bigger search spaces, ICA is repeated less than in the other two methods in the search space of CFG.

b. Computational time of the PSO algorithm is shorter than that of GA and ICA due to fewer operators.

c. In bigger search spaces, the searching power of ICA is more. The higher rate of path coverage in big problems is a good proof.
d. Although cost function structures are similar, it is obvious that the test data selected by ICA has a higher quality than those selected by the other two methods (i.e. when the cost function value is higher, selection chances are better).

Table 5. Basic algorithm parameters.

Algorithms	**ICA**			**GA**				**PSO**	
Parameters	Coefficient power	Imperialist number	Revolution rate	Mutation	Cross-over	Probability	Selection method	Inertia weight	Acceleration
Values	0.05	10% of total countries	0.2	0.01	Two Point	1	Binary tournament	0.09	2

Table 6. Average results of simulation for 30 runs.

Algorithm	**Program**	**Local search (No. of paths)**	**Convergence speed (No. of algorithm operations)**	**Computational time (minute) (runtime of algorithm)**	**Path coverage percentage**	**Cost function value**
Imperialist Competitive Algorithm	Triangle	15.5	23.4	0.21	0.86	0.75
	Schedule	148.3	132.6	30.55	0.85	0.85
	Print-token	137.4	178.4	35.53	0.86	0.88
GA	Triangle	15.5	49.3	0.94	0.86	0.71
	Schedule	101.4	177.88	140.13	0.58	0.63
	Print-token	92.2	470.37	186.99	0.58	0.68
PSO Algorithm	Triangle	16.5	86.6	0.19	0.91	0.77
	Schedule	61.1	66.2	4.93	0.35	0.41
	Print -token	53.6	46.23	5.15	0.34	0.38

7. Conclusions and perspective for future research works

In the present work, we applied ICA for the optimal generation of the test data. In order to enhance the optimization process in this regard, the cost function algorithm was re-designed and a new parameter, namely non-iterative path coverage, was added to it. For automatic test data generation by evolutionary algorithms, a new cost function was proposed. This cost function serves to compute the parameter of "non-iterative path coverage". In this situation, only the set of test data that have traversed the non-iterative paths of CFG are selected by this cost function. To this end, an Evaluation matrix was used for detecting these paths.
The cost function was used to detect the non-iterative paths of three software programs in a search-based testing problem. In this work, we used ICA, for the first time, for the automatic test data generation, which could have maximum path coverage of CFG. Three criteria including local search, computation time, and convergence speed were involved in the evaluation of ICA, as compared to GA and PSO algorithm.
The evaluation results showed that:

- As the search space for the problem increased, ICA gained a higher convergence speed.
- It was also found that a higher path coverage speed in bigger problems signified the ICA power in a local search.
- As another finding, a higher cost function rate in ICA served as an indication of the value of ideal data, as compared with the other algorithms.
- Finally, the computing time in PSO turned out to be shorter than that in the other algorithms, which could be

- attributed to the smaller number of computational operators in the structure of PSO.

To evaluate the capability of the proposed method in detecting faults, mutation testing will be used in our future research work.

Acknowledgment

The authors wish to thank University of Kashan for supporting this research work with grant No. 577242.

References

[1] Aggarwal, K. & Singh, Y. (2007). Software Engineering (3rd Ed.). New Age International Publishers

[2] Peeze, M. & Young, M. (2007). Software Testing and Analysis: Process, Principles and Techniques, John Wiley, Sons

[3] Keyvanpour, M. R., Homayouni, H. & Shirazee, H. (2011). Automatic Software Test Case Generation. Software Engineering, vol. 5, pp. 91-101.

[4] Singh, H. (2004). Automatic generation of software test cases using genetic algorithms. A thesis in Thapar University Patiala may.

[5] Shimin, L. & Zhangang, W. (2011). Genetic Algorithm and its Application in the path-oriented test data automatic generation. Procedia Engineering, vol. 15, PP. 1186 – 1190.

[6] Atashpaz-Gargari, E. & Lucas, C. (2007). Imperialist Competitive Algorithm: An Algorithm for Optimization Inspired by Imperialistic Competition. IEEE Congress on Evolutionary Computation, Singapore, pp. 4661-4667.

[7] Lucas, C., Nasiri-Gheidari, Z. & Tootoonchian, F. (2010). Application of an imperialist competitive algorithm to the design of a linear induction motor. Energy Conversion and Management, vol. 51, pp. 1407-141.

[8] Bahrami, H., Faez. K. & Abdechiri, M. (2010). Imperialist competitive algorithm using chaos theory for optimization. Computer Modelling and Simulation (UKSim), 12th International Conference on IEEE, pp. 98-103.

[9] Wang, G., Zhang, J. B. & Chen, J. W. (2011). A novel algorithm to solve the vehicle routing problem with time windows: Imperialist competitive algorithm. Advances in Information Sciences and Service Sciences, vol. 3, no. 5, pp. 108-116.

[10] Hosseini, S., Al Khaled, A. (2014). A survey on the Imperialist Competitive Algorithm metaheuristic: Implementation in engineering domain and directions for future research. Applied Soft Computing, vol. 24, pp. 1078-1094.

[11] Roustaei, R. & Yousefi Fakhr, F. (2016). A hybrid meta-heuristic algorithm based on imperialist competition algorithm. Journal of AI & Data Mining, In Press.

[12] Yousefikhoshbakht, M. & Sedighpour, M. (2013). New Imperialist Competitive Algorithm to solve the travelling salesman problem. Computer Mathematics, vol. 90, pp. 1495-1505.

[13] Gharehchopogh, F. S. & Maroufi, A. (2014). Approach of software cost estimation with hybrid of imperialist competitive and artificial neural network algorithms, Journal of Scientific Research and Development, vol. 1, pp. 50-57.

[14] Pourali, A. & Sangar, A. B. (2015). A new approach in software cost estimation with hybrid of imperialist competitive algorithm and ant colony algorithm. Academiea Royale Des Sciences D Outre-Mer Bulletin Des Seances, vol. 4, pp. 106-113.

[15] Sadeghi, B., Khatibi, V., Esfandiari, M. & Hosseinzadeh, F. (2015). A Novel ICA-based Estimator for Software Cost Estimation, Advances in Computer Engineering and Technology, vol. 1, pp.15-24.

[16] Hutcheson, M. L. (2003). Software Testing Fundamentals: Methods and Metrics. John Wiley, Sons.

[17] Qingfeng, D. & Xiao, D. (2011). An improved algorithm for basis path testing. Management and Electronic Information (BMEI), International Conference on, vol. 3, pp. 175-178.

[18] Kennedy, J. & Eberhart, R. (1995). Particle swarm optimization. Proceedings of IEEE international conference on neural networks, vol. 4, no. 2, pp. 1942-1948.

[19] Andalib, A. & Babamir, S. M. (2014). A New Approach for Test Case Generation by Discrete Particle Swarm Optimization Algorithm. 22nd Iranian Conference on Electrical Engineering, pp. 1180–1185.

[20] Sivanandam, S. N & Deepa. S. N. (2008). Genetic Algorithm Optimization Problems. Springer Berlin Heidelberg.

[21] Papadakis, M. & Malevris, N. (2012). Mutation based test case generation via a path selection strategy. J.Information and Software Technology, vol. 54, pp. 915-932.

[22] Pachauri, A. & Srivastava, G. (2013). Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism. Systems and Software, vol. 86, pp. 1191-1208.

[23] Miller, J., Reformat, M. & Zhangm, H. (2006). Automatic test data generation using genetic algorithm and program dependence graphs. Information and Software Technology, vol. 48, pp. 586-605.

[24] Babamir, S. M. & Babamir, F. S. (2009). Test-Data Generation for Program Path Coverage Using Genetic Algorithm. 4th annual International CSI Computer Conference.

[25] Sthamer, H. (1995). The Automatic Generation of Software Test data using genetic algorithms. Ph.D. Thesis, University of Glamorgan, UK.

[26] Michael, C., McGraw, G. & Schatz, M. (2001). Generating Software Test Data by Evolution. IEEE Transactions. Software Engineering, vol. 27, pp. 1085-1110.

[27] Ahmed, M. A. & hermadi, I. (2008). GA-based multiple paths test data generator. Computers & Operations Research, vol. 35, pp. 3107-3124.

[28] Bueno, P. M. S. & Jino, M. (2002). Automatic test data generation for program paths using genetic algorithms. Software Engineering. Knowledge Engineering, vol. 12, pp. 691-709.

[29] Lin, J. C. & Yeh, P. L. (2001). Automatic test data generation for path testing using GAs. Information Sciences, pp. 47-64.

[30] Watkins, A. & Hufnagel, E. M. (2006). Evolutionary test data generation: A comparison of fitness functions. Software: Practice and Experience, vol. 36, pp. 95-116.

[31] Gong, D., Zhang, W. & Zhang, Y. (2011). Evolutionary generation of test data for multiple paths coverage. Chinese Journal of Electronics, vol. 19, pp. 233-237.

[32] Zhang, W., Gong, D., Yao, X. & Zhang, Y. (2010). Evolutionary generation of test data for many paths coverage. Control and Decision Conference (CCDC), Chinese, pp. 230-235. IEEE.

[33] Harman, M. & McMinn, P. (2010). A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. IEEE Transactions on Software Engineering, vol. 36, pp. 226-247.

[34] Mansour, N. & Salame, M. (2004). Data generation for path testing. Software Quality Control, vol. 12, pp. 121–136.

[35] Suri, B. & Snehlata, B. (2011). Review of Artificial Bee Colony Algorithm to Software Testing. Research and Reviews in Computer Science (IJRRCS), vol. 2, pp. 706-711.

[36] Rothermel, G., Elbaum, S., Kinneer, A & Do, H. Software artifact infrastructure repository. Available: http://www.cse.unl.edu/~galileo/sir.

[37] Boyabatli, O. & Sabuncuoglu, I. (2004). Parameter selection in genetic algorithms. Systemics, Cybernetics and Informatics, vol. 4, pp. 78.

[38] Rezaee, J. A. & Jasni, J. (2013). Parameter selection in particle swarm optimization: a survey. Experimental & Theoretical Artificial Intelligence, pp. 527-542.

نشریه هوش مصنوعی و داده کاوی

بهینه‌سازی تابع هزینه در الگوریتم رقابت استعماری برای مسئله پوشش مسیر آزمون نرم افزار

محمدعلی سعادت جو و سید مرتضی بابامیر*

گروه مهندسی کامپیوتر، دانشگاه کاشان، کاشان، ایران.

ارسال ۲۰۱۶/۱۱/۲۲؛ بازنگری ۲۰۱۷/۰۴/۱۰؛ پذیرش ۲۰۱۷/۰۷/۲۷

چکیده:

یکی از روش‌هایی که در آزمون نرم افزار مورد استفاده قرار می‌گیرد بهینه‌سازی آزمون مبتنی بر جستجو می‌باشد. جستجو در فضای مسیرهای اجرایی کد یک نرم‌افزار برای یافتن خطا، یک چالش اساسی در نرم‌افزارهای بزرگ است. زیرا پوشش این فضا نیازمند تعداد لازم و کافی از داده‌های آزمون مناسب است. روش‌هایی که بتوانند به صوررت خودکار و با تولید داده‌های لازم و کافی فضای جستجوی حداکثری در کد نرم‌افزار را پوشش دهند، مورد استقبال واقع شده است. در این مقاله یک تابع هزینه جدید جهت تولید خودکار داده آزمونی که بتواند مسیرهای غیرتکراری گراف کنترل جریان نرم-افزار را بپیماید و با تابع هزینه مشابه در سایر مقالات مقایسه شده است که نتایج بیانگر عملکرد بهتر تابع هزینه پیشنهادی است. نوآوری دیگر مقاله، بکاریری الگوریتم رقابت استعماری در تولید خودکار داده آزمون با تابع هزینه پیشنهادی است. تولید خودکار داده آزمون توسط الگوریتم‌های رقابت استعماری، بهینه‌سازی ذرات و ژنتیک برای سه نرم‌افزار با فضای جستجوی متفاوت انجام شده است. هر کدام از الگوریتم‌ها از نظر مقدار تابع هزینه، تعداد مراحل اجرا و میزان پوشش مسیر با یکدیگر مقایسه شده‌اند. نتایج این ارزیابی نشان دهنده برتری الگوریتم پیشنهادی است.

کلمات کلیدی: آزمون نرم‌افزار، الگوریتم رقابت استعماری، تولید داده آزمون، گراف کنترل جریان، درجه پیچیدگی نرم‌افزار.