Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 5 (3): 91-101, 2011
ISSN 1819-4311 / DOI: 10.3923/jse.2011.91.101

Automatic Software Test Case Generation

[1]M.R. Keyvanpour, [1]H. Homayouni and [2]Hasein Shirazee
[1]Department of Computer Engineering, Alzahra University of Tehran, Vanak St., Tehran, Iran
[2]Department of Computer Engineering, Islamic Azad University, Qazvin Branch, Qazvin, Iran

Corresponding Author: M.R. Keyvanpour, Department of Computer Engineering, Alzahra University of Tehran, Vanak St., P.O. Box 1993893973, Tehran, Iran Tel: (+98)9133163550, (+98-21)88044040 Fax: (+98-21)88035187

ABSTRACT

Test case generation is one of the most important and costly steps in software testing. the techniques for automatic generation of test cases try to efficiently find a small set of cases that allow an adequacy criterion to be fulfilled, thus, reducing the cost of software testing and resulting in more efficient testing of software products. In this study, we analyze the application of different machine learning methods for automatic test case generation task. We describe in this study how these algorithms can help in white-box testing. Different algorithms, consists of random, GA, MA and a proposed hybrid method called GA-NN, are then considered and studied in order to understand when and why a learning algorithm is effective for a testing problem. For the experiments we use a benchmark program, called Triangle classifier. Finally, the evaluations and some open research questions are given.

Key words: White-box testing, test case generation, learning algorithms

INTRODUCTION

Software testing is an important activity of the Software Development Life Cycle (SDLC). Software testing helps in building the confidence of a developer that a program does what it is intended to do so. In other words, software testing is the process of executing a program with intends to find errors (Biswal *et al*., 2010). A software test consists of a set of test cases, each of which is made up of the input of the program, called test data and the output that must be obtained (Alba and Chicano, 2008). In structural testing the test case generator uses the structural information of the program to guide the search of new input data (for this reason it is also called white-box testing). Usually, this structural information gathered from the control flow graph of the program (Singh and Kumar, 2010).

Genetic Algorithm as a Metaheuristic that uses global search, is easy to apply to a wide range of optimization problems. Software testing is also an optimization problem with the objective that the effort consumed should be minimized and the number of faults detected should be maximized (Singh, 2004).

In this study, we analyze the application of Genetic Algorithm for automatic test case generation problem. We also apply Memetic Algorithm, using a local search technique called Hill Climbing in each generation of Genetic algorithm. For the fitness function, we use the average of three factors, including likelihood, close to boundary and branch coverage as proposed in (Arcuri and Yao, 2008) and analyze the results. Then we propose a technique that uses the features of efficient test cases to generate population of next generation in genetic algorithm. For

this purpose, we use neural network as an estimator for the fitness of test suits. We called this hybrid method GA-NN. We implemented all mentioned algorithms and analyze the results on a benchmark problem, called Triangle classifier.

Several approaches have been used for the automation of test cases generation for program based structural testing. Among these approaches, metaheuristic search techniques (Genetic Algorithms, Genetic Programming, Simulated Annealing, Tabu Search, Scatter Search, etc.) are mostly used. The application of metaheuristic algorithms to solve problems in Software Engineering was proposed by the SEMINAL network (Software Engineering using Metaheuristic Innovative Algorithms) and is widely explained by Clarke *et al.* (2003). One of these applications is software testing, in which the testing problem is treated as a search or optimization problem, as is shown in several surveys (Mantere and Alander, 2005; McMinn, 2004). Besides, this problem is ideal for the application of metaheurístic techniques (Harman and Jones, 2001). The most widely used metaheuristic technique in this yield is Genetic Algorithms (Goldberg, 1989). This technique is based on the principles of genetics and Darwin's theory of evolution. Jones *et al.* (1998), Miller *et al.* (2006), Pargas *et al.* (1999) and Sthamer (1996) used Genetic Algorithms to obtain branch coverage, Michael *et al.* (2001) to achieve condition-decision coverage, Ahmed and Hermadi (2008), Bueno and Jino (2002), Lin and Yeh (2001) and Watkins and Hufnagel (2006) to reach path coverage, Wegener *et al.* (2001) to obtain several coverage criteria, Girgis (2005) to obtain def-use coverage and Gong *et al.* (2011) to obtain Multiple Paths Coverage.

GA-based test case generation: Genetic algorithms (Arcuri and Yao, 2008) represent a class of adaptive search techniques and procedures based on the process of natural genetics and Darwin's principle of the survival of the fittest. There is a randomized exchange of structured information among a population of artificial chromosomes. GAs is a computer model of biological evolution. When GAs is used to solve optimization problems, good results are obtained surprisingly quickly. Genetic algorithm searching mechanism starts with a set of solution called a population. One solution in the population is called a chromosome. The search is guided by a survival of the fittest principle. The search proceeds for a number of generations, for each generation the fitter solutions (based on the fitness function) will be selected to form a new population. During the cycle, there are three main operators namely reproduction, crossover and mutation. The cycle will repeat for a number of generations until certain termination criteria are met.

It is impossible to test software exhaustively (i.e., for each and every possible value). There for, to test the software, tester uses the selected cases only. The selection criteria of the earlier automated test tools are mainly branch coverage, statement coverage, etc but there are problem with these tools when complicated programs are there. They may leave very important test cases that are must to be tested. So, a new technique is proposed by Singh (2004) so that all the important test cases can be selected. Selection is done basis of various factors of a good test suit (set of test cases). Here, the factors that contribute to selection of test cases are as:

- Likelihood
- Close to boundary value
- Branch coverage

These factors can be used as the fitness function of the GA to find the optimal solution i.e., the best set of test cases.

Likelihood: The paths which are more likely to be executed than others should be given higher priority for testing. Likelihood of any test suit is higher than the likelihood of any other test suit, if the test cases of the test suit follow the paths that are more likely to be executed. Mathematically, the likelihood, Λ of a test suit, T is evaluated as follows:

$$\Lambda(T) = 1-(1-\Lambda(\Pi(\tau_1)))*(1-\Lambda(\Pi(\tau_2)))*... \quad (1)$$

where, $\Pi(\tau_1)$ is the path followed when the test case τ_1 is executed, $\Lambda(\pi)$ is the likelihood of the path π which is evaluated as follows:

- Using symbolic evaluation method (Singh, 2004), evaluate the boolean expression of that path
- From the complete input domain find the probability of that boolean expression to be true
- The probability above found gives the likelihood of that path

Close to boundry value: As the chances of bugs are mostly at the boundary values, so the test cases close to boundary values must be given higher priority than the other ones for testing. Close to boundary value is the factor that represents how much the test case values are closer to the boundaries. Mathematically, close to boundary value factor, B of a test suit, T is evaluated as follows:

$$B(T) = B(\tau_1)* B(\tau_2)*... \quad (2)$$

where, $B(\tau)$ is close to boundary value factor of a test case τ which is evaluated as:

- For the test case, evaluate the path that is followed, when the test case is executed
- Using symbolic evaluation method, evaluate the boolean expression of that path
- From the boolean expression evaluate the boundary value expressions by converting any comparison operators $\{<,>,\leq,\geq,=\}$ to '=' comparison operator
- Change the boundary value expression so that right side of the boundary value expression becomes zero
- Now the close to boundary value factor of a test case is evaluated as:

$$B(\tau) = B(\beta_1)* B(\beta_2)*... \quad (3)$$

where, $B(\beta_1)$ is the factor denoting 'how much the test case values are closer to the boundary value expression, β_1 (which is of the form [expression] = 0)'.

Branch coverage: Branch coverage means the percentage of no of edges/branches of the control flow graph covered by the test suit. Control flow graph is graphical notation of the program that shows the flow of control of that program. The branch coverage factor for any test suit, T here is denoted as P (T). To evaluate this, firstly control flow graph of the program is created. Then, for each test case of a test suit, evaluate the set of branches that has been covered. Take the union of all the evaluated set of branches. Count the number of elements of the resultant set (let it be ν). At the last, Branch coverage is evaluated as:

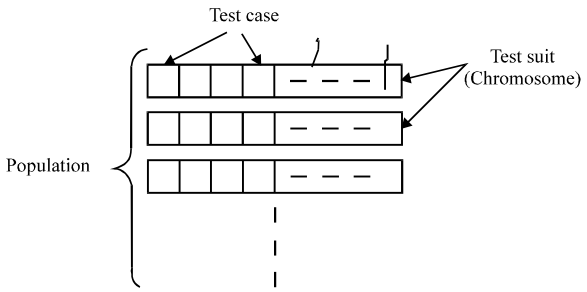

Fig. 1: Model used by Singh (2004)

$$P(T) = \nu/\varepsilon \tag{4}$$

where, ε is the total number of edges of control flow graph.

The general objective of the GA used here is to select best test suit for the testing purpose. Here, a test suit is taken as an individual (chromosome). A population is the set of test suits. Figure 1 shows the complete structure. The general factors to be considered in genetic algorithms are:

- Population Initialization
- The fitness function
- Selection/operations used
- Termination criteria

Population initialization: As at the start, there is no information about the test cases which are good and which are bad, that is why test cases are generated randomly i.e. population initialized randomly.

The fitness function: The fitness function used for every individual is the average of three factors discussed above of test suit (Singh, 2004 i.e., fitness function Φ of test suit, T is:

$$\Phi(T) = \Lambda(T) + B(T) + P(T))/3 \tag{5}$$

The priority of the test suit to be selected is directly proportional to the fitness value.

Selection: The selection of the parent chromosomes is done base on the fitness value. Test suits with highest fitness values are selected as parents for both crossover and mutation operators, i.e., the "Best" selection operator.

Uniform crossover: In uniform crossover used here, any two test suits are selected based on their fitness, Interchange randomly some of the test cases of parents ignoring test cases that are common to both the test suits. The result is the two child test suits.

Mutation: In mutation a test suit is selected based on its fitness, then randomly select some test cases of the parent test suit and replace them with the new test cases generated randomly that are not present in the parent test suit earlier and the new child test suit gets created.

Termination criteria: The search could terminate after a fixed number of generations, after a chromosome with a certain high fitness value is located or after a certain simulation time.

MA-based test case generation: The Memetic Algorithms (MAs) are metaheuristics that uses both global and local search (e.g., a GA with a HC). It is inspired by the cultural evolution. A meme is a unit of imitation in cultural transmission. The idea is to mimic the process of the evolution of these memes. From an optimization point of view, we can approximately describe a MA as a population based metaheuristic in which, whenever an offspring is generated, a local search is applied to it until it reaches a local optimum (Arcuri and Yao, 2008).

The MA we used in this study is built on GA and the only difference is that at each generation on each individual an Hill Climbing is applied until a local optimum is reached.

Hill climbing is a local search algorithm which needs a neighborhood N of the current solution Ti. The search will move to a new solution TjϵN only if Tj is better. If there are no better solutions in N, a local optimum has been reached. We need to define N. Its size has a big impact on the performance of the algorithm (Yao, 1991).

In each generation of GA we perform the mentioned local search until we get to a local optima, to impose a balance between generality and problem specificity. the flowchart of MA is shown in Fig. 2.

Proposed method: The proposed idea here is to use the features of efficient test suits and give a high fitness value to generated test suits which their features are nearer to efficient ones. For this purpose, we first extract the features of each test suit included in randomly initialized population, give a target value 1 to the test suits with high fitness values and 0 to the ones with low fitness values (the fitness value is calculated as described in previous sections). With these data (test suits

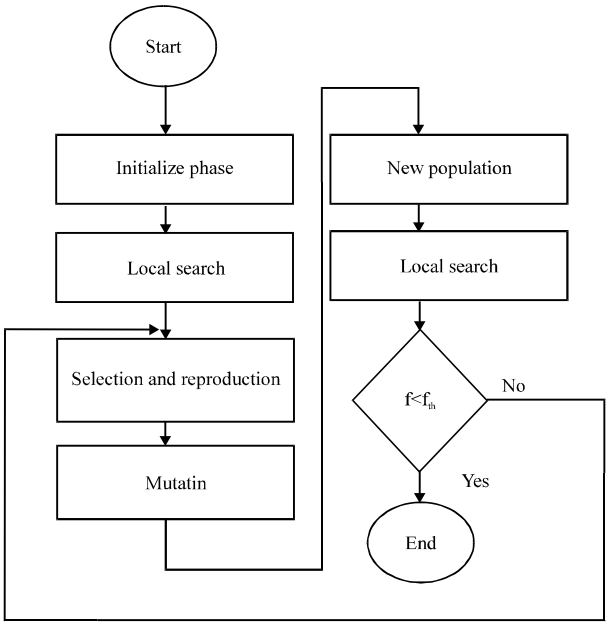

Fig. 2: Schema of the memetic algorithm. f_{th} denotes a fixed threshold value used for the stopping criterion

```
int tri_type(int a, int b, int c)
{
int type;
if (a > b) {int t = a; a = b; b = t; }
if (a > c) {int t = a; a = c; c = t; }
if (b > c) {int t = b; b = c; c = t; }
if (a + b <= c) {
type = NOT_A_TRIANGLE;
}else {
type = SCALENE;
if (a == b && b == c) {
type = EQUILATERAL;
}else if (a == b || b == c) {
type = ISOSCELES;
}
}
return type;
}
```

Fig. 3: Triangle classifier procedure

features vectors as input and their level of efficiency as target values) a two layers neural network is trained using error backpropagation with unipolar sigmoid units. Once trained, the network predicts the efficiency of new test suits. For GA operations like crossover and mutation, the most efficient test cases are selected as parents and the new children features are extracted and given to the trained neural network to predict their efficiency and this cycle will repeat for a number of generations until certain termination criteria are met. So, in this technique the NN works as a fitness function estimator for the Genetic algorithm. In fact, the network is trained in each generation of GA with new population. This technique seems to be more efficient than the previous ones because of considering the test case metrics in generating an efficient set of test cases for testing purpose.

Experiments: For testing and analyzing mentioned algorithms, we used a benchmark program called Triangle classifier program shown in Fig. 3.

Denoting the above two codes, Consider its any test suit T which is the set of test cases τ_1, τ_2, τ_3,.... Mathematically:

$$T = \{\tau_1, \tau_2, \tau_3, ...\} \tag{6}$$

A test case is a set of values of all input variables. In the above segment there are three input variables, namely, a, b and c declared as integer. So, every test case of the above segment is a set of three integer values. Mathematically:

$$\tau = \{l_1, l_2, l_3, ...\} \tag{7}$$

where, l is the value of the variable in (ν = 1, 2, 3,...) but within its range. For example, in the above codes all the three variables are of type integer (2 bytes) i.e., their values range from -32768 to 32767. So, its any test case is the set of three values, each ranging from -32768 to 32767. We describe how we apply different algorithms to these two programs, as follows:

Table 1: Test suit metrics for triangle classifier and greatest programs

No.	Test suit feature
1	Consisting at least one test case with all values equall to zero
2	Consisting at least one test case which all values are negative
3	Consisting at least one test case which all values are positive
4	Consisting at least one test case which the values are in decreasing order
5	Consisting at least one test case which the values are in increasing order
6	Consisting at least one test case which has at least one zero in its values
7	Consisting at least one test case which has at least one negative value in its values
8	Consisting at least one test case with all values equall

Random algorithm: In this approach, we generated 25 test suits randomly, each test suit contains 5 test cases which the three integer values are generated randomly considering this fact that integers ranges from -32768 to 32767. We calculate the fitness value for all of these test suits as described in section 3 and finally chose the test suit with highest fitness value as the problem solution.

Genetic algorithm: In this approach, we start with a population size of 25 individuals (test suits). We used uniform crossover and one mutation for every ten crossovers, was applied. For the fitness function we used the average of three factors described in section 3. The selection operation is the "selecting the Best" strategy and it led to better result than the "random" strategy. The algorithm stops after defined number of iterations.

Memetic algorithm: The MA we used in this study is built on GA and the only difference is that at each generation on each individual an Hill Climbing is applied until a local optimum is reached. In our case study, we can consider the following operation to define the neighbourhood N.

For a given test suit Ti as the current solution, we choose one of its test cases randomly and then add ±100 to all its three values to make N.

GA-NN algorithm: The hybrid (GA-NN) technique we proposed in this study is also built on GA and the only difference is that we applied a neural network to predict the individual fitness. The cost of applying NN to GA is high, the total number of generations is lower than in the GA (5 versus 100).

The training set for the network includes 25 test suits generated randomly. for the test suits with fitness value more than 0.7, the target value set to 1 and the for the lower ones, the target value set to 0.

We trained our two layers neural net using error backpropagation in 3000 epochs. The network learned to predict test suit efficiency. We used 10 input nodes including test suit metrics and one output node for the network. Table 1 shows some metrics for the test suit for our case study. Before training, all input and output vectors in the data set were normalized using a linear scale. We used *logsig* function as the transition function for both hidden and output layers. The Learning rate and Momentum constant parameters are calculated experimentally to 0.3 and 0.6, respectively to achieve the best Root Mean Square (RMS) error during train.

RESULTS

The results of applying mentioned algorithms to the triangle classifier procedure in 8 runs are shown in Table 2 and in Fig. 4-7. For implementation we used visual studio 2010, c# language and

Table 2: results of applying random, GA, MA and GA-NN techniques to the triangle classifier problem in 8 runs

	Results in 8 runs			
Algorithm	Number of iterations	Population size	Fitness value (mean-max)	Run time (s) (mean)
Random	1	25	0.72-0.8	13
GA	100	25	0.77-0.82	113
MA	100	25	0.82-0.86	240
GA-NN	5	25	0.79-0.85	278

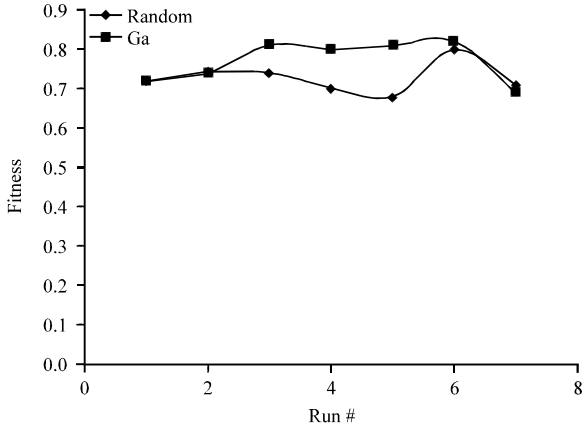

Fig. 4: Results of applying random and GA algorithms to theTriangle classifier program in 8 runs

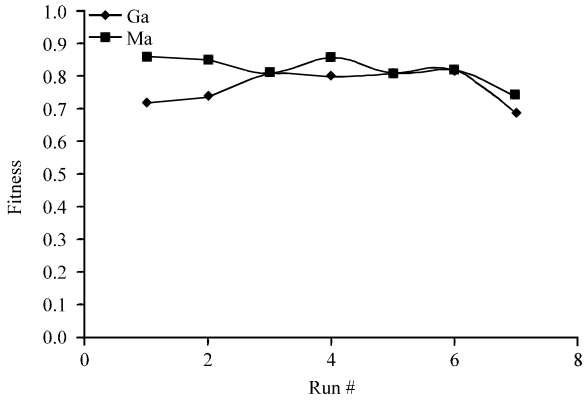

Fig. 5: Results of applying GA and MA algorithms to theTriangle classifier program in 8 runs

Matlab 2010a. The initialized population for the GA and MA is generated randomly and is the same as GA-NN initialized training set. The cost of applying NN is high, hence the total number of generations is lower than in the GA and MA. We calculate the fitness function for the resulted solution of GA-NN algorithm in the same way we did for GA and MA, for better comparison. The total generations in GA-NN algorithm is 5 because of high execution time but in GA and MA it is 100 generations.

Figure 4 shows the result of applying random and genetic algorithms to the triangle classifier program in a sequence of 8 runs. Figure 5 and 6 shows these results for GA versus MA and MA versus GA-NN, respectively. Figure 7 compares GA with GA-NN approach.

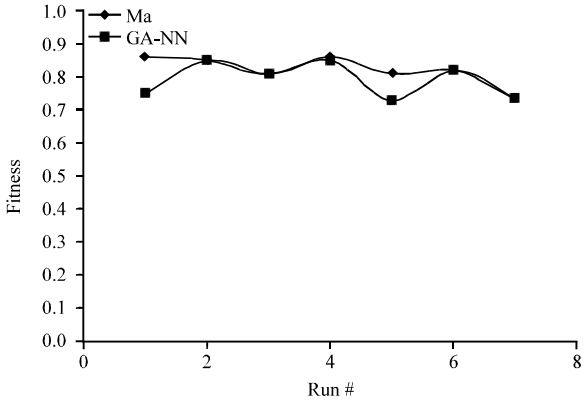

Fig. 6: Results of applying MA and GA-NN algorithms to theTriangle classifier program in 8 runs

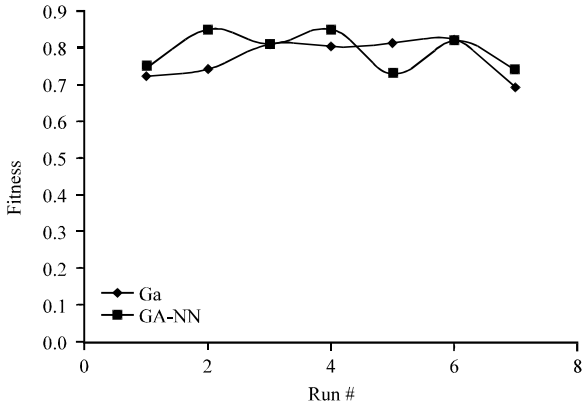

Fig. 7: Results of applying random and GA algorithms to theTriangle classifier program in 8 runs

As shown in figures, all algorithms could outperform random test case generation. The fitness value obtained in all algorithms depends on the initialized random data set. The run time for applying the hybrid method GA-NN is more than other methods, although having less number of generations. The results also show that Memetic algorithm outperforms other algorithms but the cost of applying local search to this algorithm is high (5).

CONCLUSION

We presented some test case generation techniques for white-box testing of a benchmark program. Random, Genetic Algorithms and Memetic Algorithms (MAs) have been applied. The MA we used in this study is built on GA and the only difference is that at each generation on each individual an Hill Climbing is applied until a local optimum is reached. As fitness function we used an average of three factors composed of likelihood, close to boundary value and branch coverage. The results show that Applying MA leads to better solutions for our case study, because it balances well between generality and problem specificity. A novel hybrid technique, uses neural network as a test suit fitness estimator for genetic algorithm, was also presented and called GA-NN. The cost of this new approach is higher than pure GA, so we considered lower number of generation to apply it. This approach could not outperform GA in some cases, it may be because of small training set, or few number of generations.

Our future study contains using larger training sets to get better results for our proposed approach and also use other benchmark programs for comparing the results.

ACKNOWLEDGMENT

This study is supported by Alzahra university of Tehran, computer engineering department. The authors grateful to anonymous referees of this study for their constructive comments.

REFERENCES

Ahmed, M.A. and I. Hermadi, 2008. GA-based multiple paths test data generator. Comput. Oper. Res., 35: 3107-3124.

Alba, E. and F. Chicano, 2008. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. Comput. Oper. Res., 35: 3161-3183.

Arcuri, A. and X. Yao, 2008. Search based software testing of object-oriented containers. Inform. Sci., 178: 3075-3095.

Biswal, N., S.S. Barpanda and D.P. Mohapatra, 2010. A novel approach for optimized test case generation using activity and collaboration diagram. Int. J. Comput. Appl., 1: 63-67.

Bueno, P.M.S. and M. Jino, 2002. Automatic test data generation for program paths using genetic algorithms. Int. J. Software Eng. Knowledge Eng., 12: 691-709.

Clarke, J., J.J. Dolado, M. Harman, R.M Hierons and B. Jones *et al.*, 2003. Reformulating software engineering as a search problem. IEEE Proc. Software, 150: 161-175.

Girgis, M.R., 2005. Automatic test data generation for data flow testing using a genetic algorithm. J. Universal Comput. Sci., 11: 898-915.

Goldberg, D.E., 1989. Genetic Algorithms in Search Optimization and Machine Learning. Addison Wesley Publishing Co., Reading, Massachutes.

Gong, D., W. Zhang and Y. Zhang, 2011. Evolutionary generation of test data for multiple paths coverage. Chin. J. Electron., 19: 233-237.

Harman, M. and B.F. Jones, 2001. Search-based software engineering. Inform. Software Technol., 43: 833-839.

Jones, B.F., D.E. Eyres and H.H. Sthamer, 1998. A strategy for using Genetic Algorithms to automate branch and fault-based testing. Comput. J., 41: 98-107.

Lin, J.C. and P.L. Yeh, 2001. Automatic test data generation for path testing using gas. Inform. Sci., 131: 47-64.

Mantere, T. and J.T. Alander, 2005. Evolutionary software engineering: A review. Applied Soft Comput., 5: 315-331.

McMinn, P., 2004. Search-based software test data generation: A survey. Software Test. Verification Reliab., 14: 105-156.

Michael, C.C., G. McGraw and M. Schatz, 2001. Generating software test data by evolution. IEEE Trans. Software Eng., 27: 1085-1110.

Miller, J., M. Reformat and H. Zhang, 2006. Automatic test data generation using genetic algorithm and program dependence graphs. Inform. Software Technol., 48: 586-605.

Pargas, R.P., M.J. Harrold and R.R. Peck, 1999. Test-data generation using genetic algorithms. Software Testing Verification Reliability, 9: 263-282.

Singh, H., 2004. Automatic generation of software test cases using genetic algorithms. Master Thesis, Computer Science and Engineering Department, Thapar Institute of Engineering and technology, Patiala, Punjab, India.

Singh, K. and R. Kumar, 2010. Optimization of functional testing using genetic algorithms. Int. J. Innovation Manage. Technol., 1: 43-46.

Sthamer, H., 1996. The automatic generation of software test data using genetic algorithms. Ph.D. Thesis, University of Glamorgan, UK.

Watkins, A. and E.M. Hufnagel, 2006. Evolutionary test data generation: A comparison of fitness functions. Software: Pract. Experience, 36: 95-116.

Wegener, J., A. Baresel and H. Sthamer, 2001. Evolutionary test environment for automatic structural testing. Inform. Software Technol., 43: 841-854.

Yao, X., 1991. Simulated annealing with extended neighborhood. Int. J. Comput. Math., 40: 169-189.