Online ISSN: 3107 - 7676

IJMR 2025; 1(6): 04-10

2025 November - December

www.allmultiresearchjournal.com

Received: 10-09-2025

Accepted: 12-10-2025

Published: 04-11-2025

DOI: https://doi.org/10.54660/IJMR.2025.1.6.04-10

Software Testing Using Genetic Algorithms

Sumit Saxena [1*], Dr Qaim Mehdi Rizbi [2]

Department of Computer Science, Shri Krishna University Chhatarpur, Madhya Pradesh, India

Corresponding Author; **Sumit Saxena**

Abstract

The purpose of this work is to introduce a collection of techniques that utilize a genetic algorithm in order to generate test data automatically for software testing. Researchers have been proposing a number of various strategies for generating test data for a number of years now. Each of these methods has its own set of disadvantages. The purpose of this work is to introduce a number of test techniques that are based on Genetic Algorithms (GAs) and that will have different parameters that can be used to automate the generation of test data that is based on the internal structure of the program. In order to determine the most appropriate approach for testing, the factors that have been identified are utilized in the process of evaluating the fitness function of the genetic algorithm. The test populations are taken as input by these algorithms, which then proceed to evaluate the test cases for that particular program. The total effectiveness of the genetic algorithm in the realms of search space exploration and exploitation will be improved as a result of this integration, which will also bring about a higher convergence rate.

Keyword: Genetic Algorithm, Fitness Function, Test Data

Introduction

The process of software testing involves testing the amount and quality of software during runtime to the greatest extent possible. The most fundamental examination of software is conducted within the environment for which it was originally built. The process of checking whether or not it runs is carried out in order to ensure that it produces outputs that are both proper and efficient. In addition, software testing is carried out in foreign contexts in order to investigate the potential for scalability [8, 12].

Every single piece of software is put through its paces in a number of different environments that have been carefully planned out. The assumptions of some functions have been made, and the testing that has been carried out is expected to yield the appropriate results under these assumptions. However, it is not possible to identify every single problem at any given moment. Rather, it provides a comparison that examines the state and behavior of the product, which are the principles or methods by which users may be able to identify the issue. In general, a test case is made up of the data that serves as the input for the software testing process. The components that make up this system include a unique identification, references to requirements from a software specification, a sequence of steps that must be carried out, events, preconditions, input, output, the expected outcome, and the actual outcome. The number seventeen is written in brackets. It is often referred to as an anticipated outcome that will occur in the setting that is being evaluated. This can be as basic as stating, "for condition your derived result is b," but other test cases provided a full examination of the input circumstance and displayed findings that were in line with expectations [2, 9, 16].

The Genetic Algorithm (GA), which is a type of meta-heuristic search tool, is used by Evolutionary Testing in order to transform the process of generating test cases into an optimal problem [4, 13, 27]. The purpose of evolutionary testing is to identify the best combinations of test parameters that fulfill a predetermined test criterion. A "cost function" that is used to represent this test criterion quantifies the degree to which each of the optimization parameters that were generated automatically fulfills the test requirement that was specified [7, 22].

A study of several forms of genetic algorithms is conducted as part of our research, which is presented in our paper. Different algorithms have been tested on a variety of different tools, and their performance has been analyzed. All of these algorithms adhere to the same foundation of evolutionary testing, but they possess distinct cost functions. The observations that are made regarding the manner in which these functions respond are the result of running these cost functions on various tools [1, 3, 5, 11].

Introduction to Genetic Algorithm

In situations where there is a limited amount of information available, one of the most effective methods for resolving a group of problems is to utilize a genetic algorithm. Due to the fact that genetic algorithms are a very broad type of algorithm, they will perform effectively in any search space [1, 25, 30, 33]. All you have to know is what you want the solution to be able to accomplish in order to perform well, and a genetic algorithm will be able to develop a high-quality solution for you. In order to provide solutions for a variety of complicated issues, genetic algorithms utilize the concepts of selection and evolution.

There is a tendency for genetic algorithms to do well in environments that have a very big collection of candidate solutions and a search space that is not favorable and has numerous hills and valleys [12, 15, 16]. It is true that genetic algorithms are able to perform effectively in any context; nevertheless, it is possible that they could be surpassed by algorithms that are more situationally specific when it comes to search spaces that are less complicated. As a result, it is important to remember that when dealing with random events, genetic algorithms are not always the most appropriate option. They may occasionally need a significant amount of time to execute, which makes them unsuitable for usage in real-time situations. On the other hand, they are among the most effective techniques that may be used to develop solutions of a high standard rapidly in order to address a problem [4, 8, 21]. When it comes to constructing a genetic algorithm, there are a small number of fundamental methodologies and terminology that will be utilized, such as:

Individual – Possible solutions
Population - Set of all *individuals*
Search Space - All possible solutions to the specified problem
Chromosome – Blueprint for an *individual*
Trait - Possible aspect of an *individual entity.*
Allele - Possible settings for a *trait*
Locus - The position of a *gene* on the *chromosome*
Genome - Collection of all *chromosomes* for an *individual entity.*

Background

Genetic algorithms employ three fundamental operations on their population.

Selection: A systematic approach is employed to identify the criteria by which individuals are selected for reproduction from a population, grounded in their fitness levels. Fitness is characterized as an individual's ability and capacity to survive and reproduce within a given environment. The process of selection creates a new population derived from the previous one, thereby initiating a new generation. The fitness value of each chromosome in the current generation is assessed through a suitable evaluation process. Consequently, the fitness value serves as a criterion for selecting a subset of superior chromosomes from the population to carry forward into the next generation [5, 6].

Crossover: Following the selection process, the crossover operation is implemented on the chromosomes chosen from the population. Crossover entails the exchange of sequences of bits or genes within the strings of two individuals [8, 10]. This procedure of exchanging traits is executed and reiterated with various parent individuals until the subsequent generation showcases the most optimal individuals.

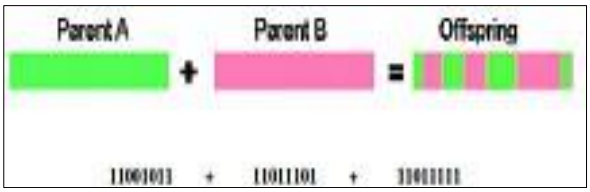


Fig 1: Uniform Crossover

Mutation: Following the crossover process, the mutation operation is implemented on a randomly chosen subset of the population. Mutation results in subtle changes to chromosomes, facilitating the introduction of beneficial traits. The primary objective of mutation is to introduce diversity within a population [1, 5].

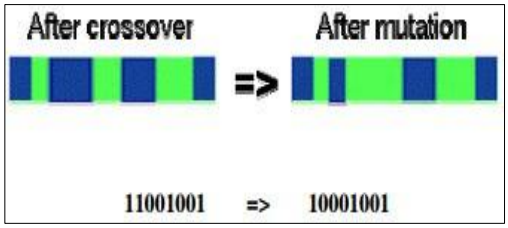


Fig 2: Mutation (Bit Inversion)

Factors essential in a fitness function are:

- Likelihood.
- Close to Boundary Value.
- Branch Coverage.

It has been proven that GAs required less CPU time in reaching a global solution in software testing **[13].**

Need for Genetic Algorithms in Software Testing:

1. Drawbacks of manual testing: [7, 12]
2. Speed of operation is limited as it is carried out by humans. High investment in terms of cost, time.
3. Limited availability of resources
4. Redundancy in test cases.
5. Inefficient and inaccurate test checking.

Pros of using genetic algorithms in software testing:
Parallelism is a important characteristic of genetic testing [11, 19].

Less likely to get stuck in extreme ends of a code during testing since it operates in a search space.
With the same encoding, only fitness function needs to be changed according to the problem.

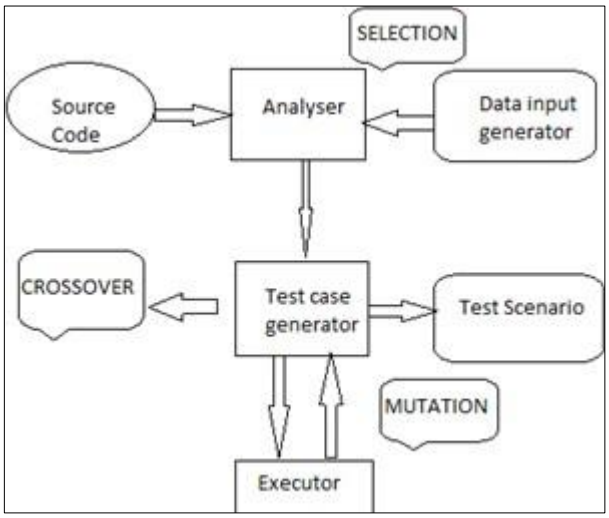


Fig 3: Test Case Generation in Software Testing Using GA

Genetic Algorithm Working

The genetic algorithm represents an evolutionary method in computing, capable of identifying suitable approximate solutions for optimization challenges. The fundamental approach utilized by Genetic algorithms generally consists of generating an initial collection of random solutions (population) and assessing them [2, 5, 9, 12]. After a selection process, the most effective solutions are identified as the parents, which are then utilized to produce new solutions, referred to as children. These values may serve as substitutes for other less significant members of the population. The new population is subsequently reevaluated, and the process of generating new values persists, producing new generations until a final solution is identified or an alternative criterion for result determination is achieved.
The Genetic Algorithm draws its terminology from the realm of biology. For instance, the genetic algorithm employs various representations for potential solutions known as chromosomes. The operators utilized to produce new offspring solutions, such as crossover and mutation, are inspired by natural processes. In their most fundamental and straightforward form, Genetic Algorithms were primarily utilized for single objective search and optimization tasks. Most Genetic algorithms typically incorporate a chromosome, genetic operators, a selection mechanism, and an evaluation mechanism [23, 27].
In this scenario, our task is to identify the necessary parameters to construct an effective function and encapsulate them within a binary string, which will serve as the definition of our chromosome. Consequently, every chromosome will comprehensively characterize a Function.
A typical method when utilizing Genetic Algorithms involves initiating a 'population' of random chromosomes (Test Variables), often around 100. As previously mentioned, we can assess each individual and assign a score, or more accurately, 'evaluate its fitness' (function).
This calculated fitness function will assist in assessing the efficiency of the employed method [11, 27, 29]. To enhance the efficiency of the test results, we can modify the input parameters and derive values for varying population sizes.

Applications of Genetic Algorithm

Genetic algorithms have been applied to address complex issues, including NP-complete and NP-hard problems, as well as in machine learning, and are also utilized for the evolution of basic test programs. They serve as a highly efficient method for swiftly identifying a viable solution to a multifaceted issue.
Genetic algorithms demonstrate their highest efficiency and effectiveness in search spaces where knowledge is limited. Conversely, genetic algorithms may generate solutions that function well within a test environment but tend to diverge when applied in real-world scenarios [17, 24].
In simple terms, a genetic algorithm can be employed to develop solutions for problems that are complex and challenging to compute and analyze.

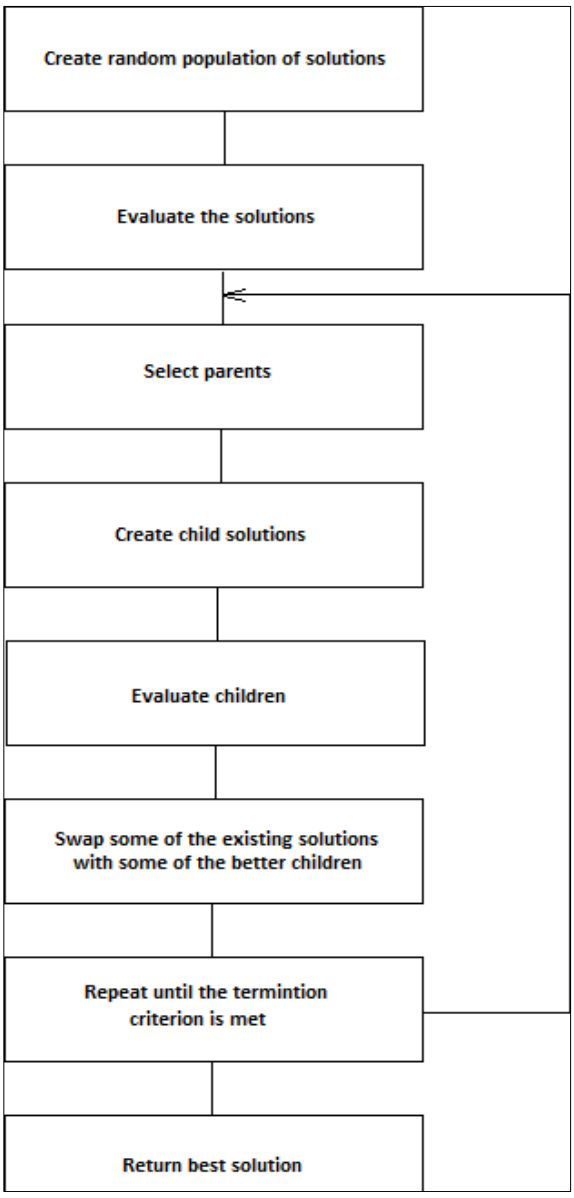


Fig 4: Flow chart of the workflow of Genetic Algorithm used for Test Case Generation n Software Testing.

Implementation of GA in Software Testing

Test case generation using GA in Ruby

Algorithm: Start with randomly generated test cases from the population.
Calculate the fitness f(x) of each pair of test cases (chromosome x) in the population. Repeat the following steps until a n child test cases have been generated.

1. Select a pair of parent test cases from the current population where the probability of selection is an increasing function of fitness. Selection is done "with replacement," meaning that the same pair of test case can be selected more than once to become a parent. i.e. (Selection process is carried out)
2. With the crossover probability Pc, cross over the pair at a randomly chosen point to form two child cases or off springs. If no crossover takes place, form two test cases that are exact copies of their respective parent cases.
3. Mutate the two child cases with mutation probability Pm, and place the resulting pair of test cases in the new population. If n is odd, one new population member can be discarded at random. Replace the current test cases with the new test cases.

Population size = 50
Number of generations = 500 Crossover rate=0.7
Mutation rate = 0.001

Ruby Implementation with First Case:

Table 1: Average fitness value with mutation rate = 0.001

Generation	Average fitness	Max fitness
1	31.88	33
2	32.88	34
3	32.67	34
4	32.75	35
5	32.96	34
10	34.17	38
25	37.29	42
50	41.21	44
100	39.67	47
150	44.33	49
200	46.33	47
250	47.63	49
500	50.23	50
750	51.79	51
1000	52	51

6.1.2
Population size = 50
Number of generations = 500 Crossover rate=0.7
Mutation rate = 0.01

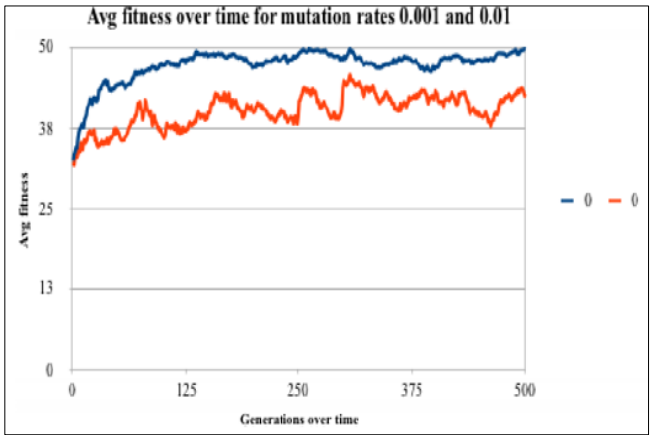
Ruby Implementation with Second Case:

Table 2: Average fitness value with mutation rate = 0.01

Generation	Average	Max
1	32.25	32
2	32.13	32
3	33.92	33
4	32.42	33
5	33.79	33
10	34.71	34
25	38.42	36
50	39.08	37
100	37.42	36
150	35.54	40
200	38.79	40
250	41.83	39
500	42.18	43

Mutation rate has a great impact on the average on the average fitness of genetic algorithms during testing. Smaller the rate, better the fitness function value will be.

Below is a graph which represents the average fitness overtime for different mutation rates.



Genetic Algorithm Implementation in C++
Pseudo-code for genetic algorithm:

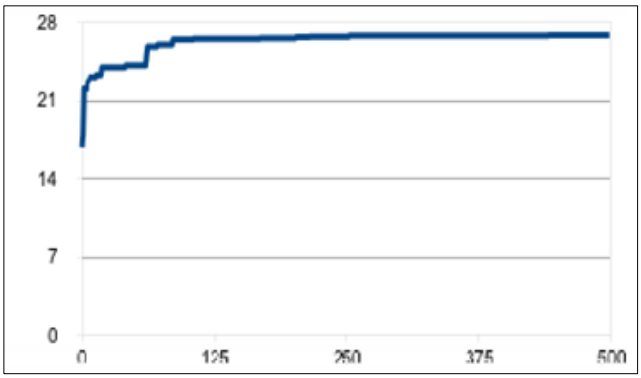
1. choose initial_population:
2. evaluate individual_fitness function determine population's_average
3. fitness_function
4. Repeat
5. select best_case individuals to reproduce;
6. mate_pairs at random; apply crossover_operator; apply mutation_operator; evaluate Individual fitness;
7. determine population's average fitness;

The second step involves generating data through the outer loop, which will produce the remaining possible test cases. To address the potential for impractical test requirements, encompassing branches and statement values, the loop will generate iterations until it meets the test results for the specified population values. The algorithm generates values that will be utilized for the crossover and mutation operator. Subsequently, the fitness function for each individual value is generated, and the average fitness function of the population is calculated.
In the concluding phase, the algorithm will allocate the aggregated values of the test cases and identify at least one individual desired fitness function value until a sufficient number of test generations have been completed.

6.2.1
Population size = 50
Number of generations = 250 Crossover rate=0.7
Mutation rate = 0.001

Table 3: Best fitness value generation

Iteration	Best fitness	Average fitness	Standard Deviation
1	16.77	11.59	5.54
2	20.39	15.09	3.26
3	20.39	15.71	3.27
4	22.99	15.84	3.79
5	22.99	16.03	3.89
10	23.24	17.65	3.46
25	25.12	20.87	3.61
50	25.89	21.42	4.32
100	26.27	20.88	5.33
150	26.77	23.28	3.77
200	26.77	20.38	6.91
250	26.78	22.41	4.42
500	26.98	22.68	5.54

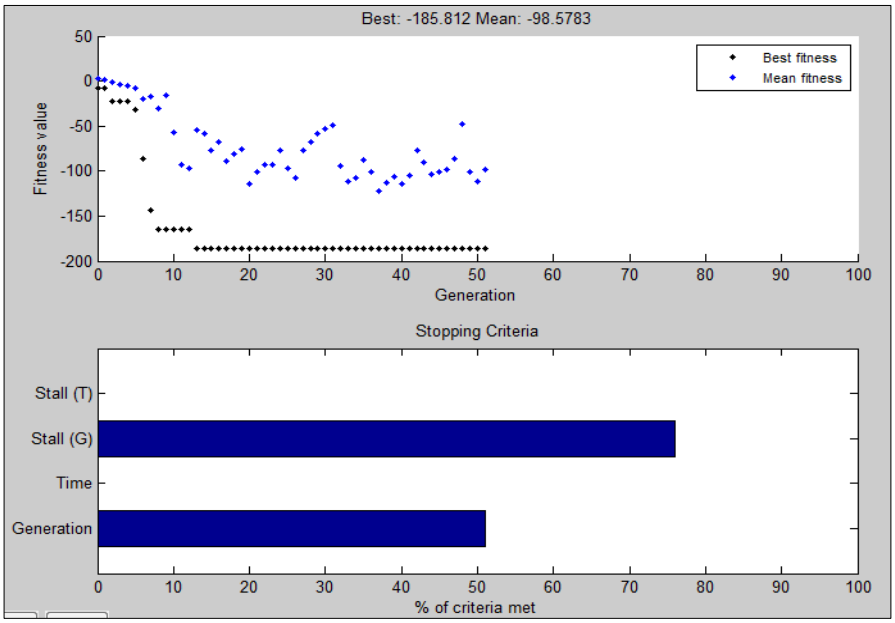


Graph 1: Number of generations (x-axis) vs. average fitness (y-axis)

Genetic Algorithm Implementation using Matlab

In this work both the genetic algorithm and the random testing method were compared and detailed analysis of the best fitness has been evaluated. In order to compare Genetic algorithm and a pure random method, 150 test cases were generated and tested by both methods [22]. From this we can see that the average response time of test cases created by genetic algorithm is much efficient than that of the random method.

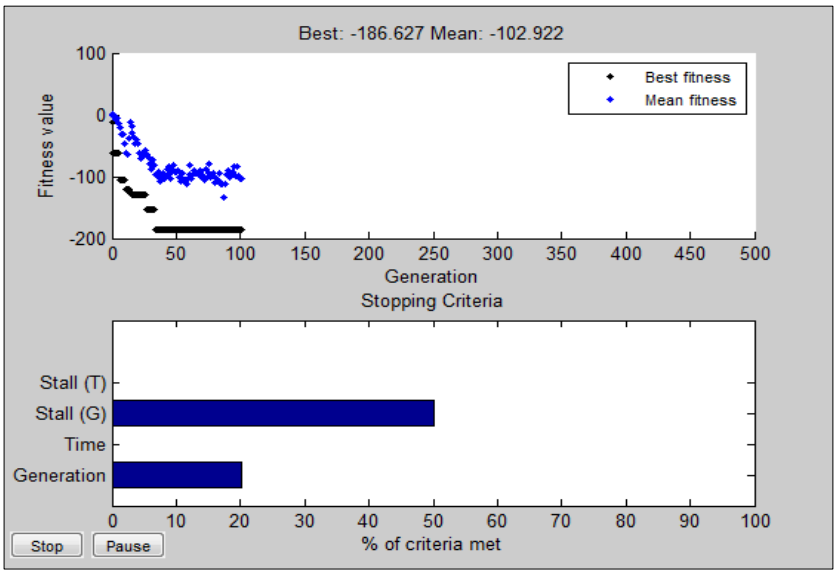
Case 1 shows the plot of the best and mean score of the population at every generation. The second plot function is stopping criteria, which plots the percentage of stopping criteria satisfied.



Graph 2: Shows the plot of the best and mean score of the population at every generation

Results generated for Case 1:
The number of generations: 124
The number of function evaluations: 6250 The best function value found: -186.

Case 2 shows a better detailed analysis of the best and the mean fitness function by changing the test cases to 20 instead of 10. This iteration generates better results than the previous iteration and by further iterating the test cases the result is obtained.



Graph 3: Better detailed analysis of the best and the mean fitness function by changing the test cases to 20 instead of 10

Genetic algorithm for test case generation using Mat lab was implemented. It can be found out that for a population of 50 and maximum generations 500, the best fitness function is carried out until maximum generation is reached or until a certain value after which no significant change occurs. The iterative generation is stopped itself after such a condition hence providing us with a further optimization and avoid redundant solutions.

Conclusions

In this paper, we analyzed how evolutionary techniques such as GAs helped in software testing. The findings demonstrate the efficiency of software testing through the application of Genetic Algorithms, even as the number of test cases rises. In Random Testing Methods, the lack of temporal dependence among data points leads to inefficiencies as the complexity of the code increases. To enhance the efficiency and reduce the process time of software testing, Genetic Algorithms are employed, offering a method for automatic test case generation. The evolutionary generation of test cases demonstrates greater efficiency and cost-effectiveness compared to Random Testing.

Acknowledgement

We would like to acknowledge the immense support of our Professor Dr …….. without whom it was not possible to write the paper. Her guidance formed the base of the paper and her expertise comments greatly enhanced the manuscript. We would also like thank our university (Shri KJrishana University Chhaterpur MP) which has always encouraged the students to go in the field of research and develop an interest towards exploration and innovation.

References

1. Goldberg DE. Genetic algorithms in search, optimization and machine learning. Massachusetts: Addison-Wesley; 1989.
2. Horgan J, London S, Lyu M. Achieving software quality with testing coverage measures. IEEE Computer. 1994;27(9):60–9.
3. Berndt DJ, Fisher J, Johnson L, Pinglikar J, Watkins A. Breeding software test cases with genetic algorithms. In: Proceedings of the Thirty-Sixth Hawaii International Conference on System Sciences (HICSS-36); c2003 Jan; Hawaii.
4. Last M, Eyal S, Kandel A. Effective black-box testing with genetic algorithms. In: IBM Conference Proceedings.
5. Lin JC, Yeh PL. Using genetic algorithms for test case generation in path testing. In: Proceedings of the 9th Asian Test Symposium (ATS'00); c2000 Dec 4–6; Taipei, Taiwan.
6. Baresel A, Sthamer H, Schmidt M. Fitness function design to improve evolutionary structural testing. In: Proceedings of the Genetic and Evolutionary Computation Conference; c2002.
7. Michael CC, McGraw GE, Schatz MA, Walton CC. Genetic algorithms for dynamic test data generation. In: Proceedings of the 1997 International Conference on Automated Software Engineering (ASE'97). IEEE; 1997.
8. Somerville I. Software engineering. 7th ed. Addison-Wesley; c2004.
9. Mathur AP. Foundations of software testing. 1st ed. Pearson Education; c2008.
10. Alander JT, Mantere T, Turunen P. Genetic algorithm based software testing [Internet]. 1997 [cited 2025 Nov 13]. Available from: http://citeseer.ist.psu.edu/40769.html
11. Mansour N, Salame M. Data generation for path testing. Software Quality Journal. 2004;12:121–36.
12. Srivastava PR, *et al*. Generation of test data using meta heuristic approach. In: IEEE TENCON; c2008 Nov 19–21; India.
13. Wegener J, Baresel A, Sthamer H. Suitability of evolutionary algorithms for evolutionary testing. In: Proceedings of the 26th Annual International Computer Software and Applications Conference; c2002 Aug 26–29; Oxford, England.
14. Berndt DJ, Watkins A. Investigating the performance of genetic algorithm-based software test case generation. In: Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE'04); c2004 Mar 25–26; University of South Florida. p. 261–2.
15. Korel B. Automated software test data generation. IEEE Transactions on Software Engineering. 1990;16(8).
16. Zhang B, Wang C. Automatic generation of test data for path testing by adaptive genetic simulated annealing algorithm. In: IEEE Proceedings; c2011. p. 38–42.
17. Doungsa C, *et al*. An automatic test data generation from UML state diagram using genetic algorithm [Internet]. Available from: http://eastwest.inf.brad.ac.uk/document/publication/DoungsaardSKIMA.pdf
18. Berndt DJ, Watkins A. High volume software testing using genetic algorithms. In: Proceedings of the 38th International Conference on System Sciences (HICSS-38); c2005. IEEE. p. 1–9.
19. Emanuelle F, *et al*. Using genetic algorithms for test plans for functional testing. In: 44th ACM SE Proceedings; c2006. p. 140–5.
20. Goldberg DE. Genetic algorithms in search, optimization and machine learning. Massachusetts: Addison-Wesley; 1989.
21. Girgis M. Automatic test generation for data flow testing using a genetic algorithm. Journal of Computer Science. 2005;11(6):898–915.
22. Giuseppe A, *et al*. Testing web applications: The state of art and future trends. Information and Software Technology. 2006;48:1172–86.
23. Lin JC, Yeh PL. Automatic test data generation for path testing using GAs. Information Sciences. 2000;133:47–64.
24. Carlos J, *et al*. A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software [Internet]. In: AST'08 Proceedings. ACM; c2008 [cited 2025 Nov 13]. Available from: http://www.cs.bham.ac.uk/~wbl/biblio/cache/httpjcbribeiro.googlepages.com_ast12-ribeiro.pdf
25. You L, Lu Y. A genetic algorithm for the time-aware regression testing reduction problem. In: International Conference on Natural Computation. IEEE; c2012. p. 596–9.
26. McMinn P. Search-based software test generation: a survey. Software Testing, Verification and Reliability. 2004;14(2):105–56.
27. Last M, *et al*. Effective black-box testing with genetic algorithms. Lecture Notes in Computer Science. 2006;pp. 134–48. Springer.

28. Alzabidi M, *et al*. Automatic software structural testing by using evolutionary algorithms for test data generation. International Journal of Computer Science and Network Security. 2009;9(4):390–5.
29. Rajappa V, *et al*. Efficient software test case generation using genetic algorithm-based graph theory. In: International Conference on Emerging Trends in Engineering and Technology. IEEE; c2008. p. 298–303.
30. Peng X, Lu L. A new approach for session-based test case generation by GA. In: IEEE Proceedings; c2011. p. 91–6.
31. Kruse PM, *et al*. A highly configurable test system for evolutionary black box testing of embedded systems. In: GECCO Proceedings. ACM; c2009. p. 1545–51.
32. Zhao R, Lv S. Neural network-based test case generation using genetic algorithm. In: 13th IEEE International Symposium on Pacific Rim Dependable Computing; c2007. p. 97–100.
33. Patton RM, *et al*. A genetic algorithm approach to focused software usage testing [Internet]. Annals of Software Engineering. Available from: http://www.cs.ucf.edu/~ecl/papers/03.rmpatton.pdf

How to Cite This Article
Saxena S, Rizbi QM. Software Testing Using Genetic Algorithms. International Journal of Multi Research. 2025; 1(6): 04-10.