

***“GenTest:-A New Genetic Algorithm-Based Testing
Approach in Testing Automation”***

A DISSERTATION

SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE AWARD OF THE DEGREE

OF

**MASTER OF TECHNOLOGY
IN
SOFTWARE ENGINEERING**

Submitted By:-

TARUNPREET SINGH SURI (2K21/SWE/24)

under the guidance of

Prof. Ruchika Malhotra

Head of Department(Software Engineering)
Delhi Technological University, Delhi



**DEPARTMENT OF SOFTWARE ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road Delhi-110042**

JUNE, 2023

Department of Software Engineering
Delhi Technological University
(Formerly Delhi College of Engineering)
Bawana Road Delhi-110042

CANDIDATE'S DECLARATION

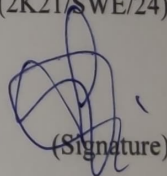
I, hereby declare that the work presented in this thesis entitled "**GenTest:-A New Genetic Algorithm Based Testing Approach in Testing Automation**", in fulfilment of the requirement for the award of the MASTER OF TECHNOLOGY degree in Software Engineering submitted in Department of Software Engineering at DELHI TECHNOLOGICAL UNIVERSITY, New Delhi, is an authentic record of my own work carried out during my degree under the guidance of **Prof. Ruchika Malhotra (HOD, SWE-DTU)**.

The work reported in this has not been submitted by me for the award of any other degree or Diploma.

Date: 29.05.2023

Place: New Delhi

TARUNPREET SINGH SURI
(2K21/SWE/24)



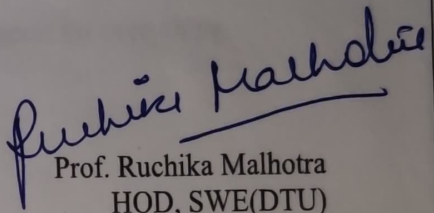
(Signature)

Department of Software Engineering
Delhi Technological University
(Formerly Delhi College of Engineering)
Bawana Road Delhi-110042

CERTIFICATE

This is to certify that **Mr TARUNPREET SINGH SURI (2K21/SWE/24)** is a student at the Department of Software Engineering of Delhi Technological University formerly known as Delhi College of Engineering undergone a major project work of 4th semester titled **“GenTest:-A New Genetic Algorithm Based Testing Approach in Testing Automation”**. To the best of my knowledge, this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Date: 29.05.2023
Place: New Delhi


Prof. Ruchika Malhotra
HOD, SWE(DTU)

SUPERVISOR

(Signature)

Department of Software Engineering

Delhi Technological University

(Formerly Delhi College of Engineering)

Bawana Road Delhi-110042

ACKNOWLEDGMENT

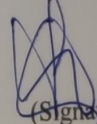
I would like to express my special thanks to my mentor **Prof. Ruchika Malhotra** who supported me in the completion of this project. This project titled "**GenTest:-A New Genetic Algorithm Based Testing Approach in Testing Automation**" was a golden opportunity to do research and enhance my skill set. It was her enigmatic supervision, unwavering support and expert guidance which has allowed me to complete this work in due time. I humbly take this opportunity to express my deepest gratitude to her.

Lastly, I would also like to thank my parents for their lovely support for everything.

Date: 29.05.2023

Place: New Delhi

TARUNPREET SINGH SURI
(2K21/SWE/24)



(Signature)

ABSTRACT

In today's time, the systems have become very complex and very big in size because of which manual testing can be a hugely daunting task which is not only very expensive but also very time-consuming. Many new approaches have come up for easing the task of test case generation and testing software systems. So there is a need for new techniques through which we can automate the process of

To find the solution to the above issue new search-based techniques have been introduced, Search Based Software Testing(SBST) is used in this. These techniques have become very popular for the automation of test data creation. Through this report, we aim to provide an approach based on Search-Based Software Testing in which we will be using a technique based on Genetic Algorithm(GA) and observe how effective that is in generating test data compared to the classic Genetic Algorithm.

In this Thesis, we will work on a modified version of the Genetic Algorithm(GA) to try to automate the process of Test data creation and we will also compare this modified algorithm to the classic version of the Genetic Algorithm.

This work will help the future researchers who want to research and work in this field of Automation of test cases using Search-Based Techniques.

TABLE OF CONTENTS

CANDIDATE'S DECLARATION	i
CERTIFICATE	ii
ACKNOWLEDGMENT	iii
ABSTRACT	iv
TABLE OF CONTENT	v
LIST OF FIGURES	vi
LIST OF ABBREVIATIONS	vii
CHAPTER 1	1
INTRODUCTION	1
CHAPTER 2	3
BACKGROUND STUDY	3
CHAPTER 3	6
LITERATURE REVIEW	6
CHAPTER 4	9
METHODOLOGIES	9
4.1 GenTest	9
4.2 Advantages of GenTest over Classic G.A	12
CHAPTER 5	14
IMPLEMENTATION	14
CHAPTER 6	16
RESULT AND ANALYSIS	16
CHAPTER 7	19
CONCLUSION AND FUTURE SCOPE	19
REFERENCES	21

LIST OF FIGURES

Figure 1.1: Different categories of Meta-Heuristic algorithms [1].	2
Figure 2.1: Flowchart of the Genetic Algorithm [3].	3
Figure 2.2: Pseudo Code of the Classic Genetic Algorithm [4].	5
Figure 4.1: Pseudo Code of the GenTest Algorithm.	9
Figure 5.1: Implementation of the GenTest Algorithm	14
Figure 6.1: Comparison of the GenTest Algorithm with Classic G.A(Triangle Problem)	17
Figure 6.2: Comparison of the GenTest Algorithm with Classic G.A(Next Date)	17

LIST OF ABBREVIATIONS

SBST	Search-Based Software Testing
GA	Generic Testing
M.H	Meta Heuristic

CHAPTER 1

INTRODUCTION

As time is passing our systems are becoming very complex and very big in size and of which manual testing can be a hugely daunting task which does not only very expensive but also very time-consuming. Many new techniques have cropped up in recent times through which we can automatically generate test case data and test our systems/software. Search Based software testing is one such field in which we can use search/optimisation techniques to find the solution to the above-given issue in the software testing domain.

The main reason for increased interest in these techniques is that data generation for test case problems can often be reduced to optimization or search problems. We can often start with some random solutions to a problem and optimise them to get more optimised solutions to our problems. These Problems can be solved using Meta-Heuristic algorithms as these are search algorithms which can optimise a random solution to more optimised solutions for a particular problem.

Different categories of Meta-Heuristic algorithms are given in Figure-1. Search Based algorithms have become very famous in these times in testing and to automate the activity of the creation of test data. Testing is done to find software bugs so we can fix them. In recent years we have observed the rise of Search-Based Software Testing (SBST) and especially the techniques of generating test data that is based on given criteria. We can ensure that the software is bug-free with the help of testing.

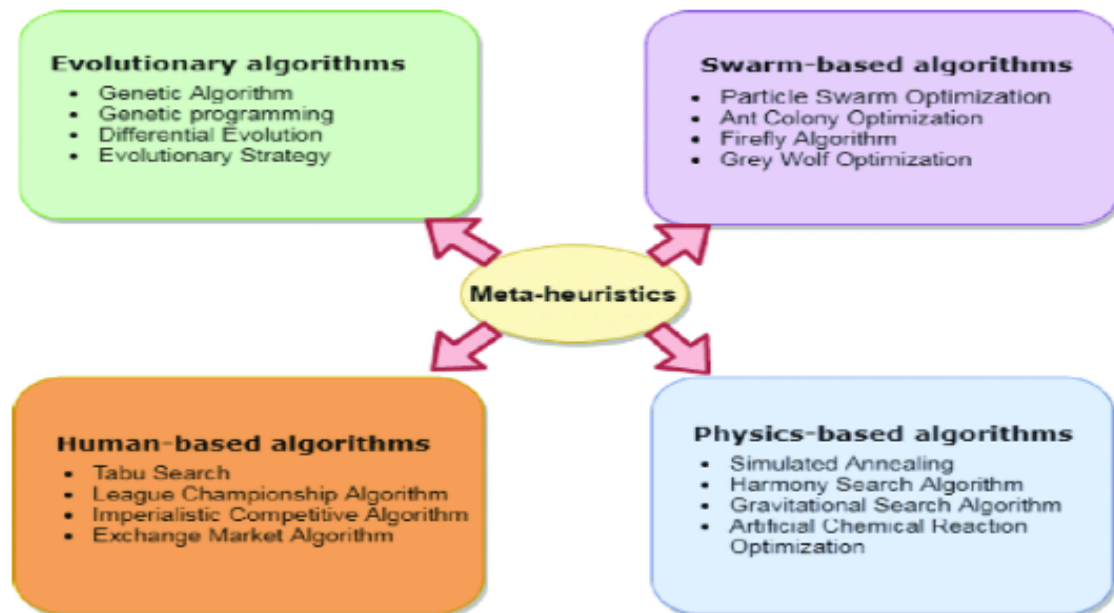


Fig. 1. Different categories of Meta-Heuristic algorithms [1].

In this thesis, we will be defining and using a modified version of the Genetic Algorithm which we have named as **GenTest** to automate the process of test creation. We will also be comparing the performance of this GenTest to the classic version of Generic Algorithm Using the fitness of the solutions created by both of them.

We will be applying this modified Algorithm to two problems (Triangle problem, Next date problem) and we will be getting the optimised test cases for these problems which are generated by our GenTest Algorithm and we will compare the performance of this algorithm on these problems with the classic version of GA.

CHAPTER 2

BACKGROUND STUDY

2.1 Genetic Algorithm

“Genetic Algorithm (GA) is an evolutionary searching algorithm used to search results for optimization and search problems” [2]. GA has part of the evolution such as generation of population, mating, cross over and mutation which proceeds towards the generation of fitter offspring or solutions. It starts with the Initialization of the population by randomly generating solutions.

The next step is that of Selection in which solutions which are individual are picked through a process. The next step after this is of Reproduction of the next generation of solutions from those which are picked with the help of genetic operators: crossover and mutation. After this process of making the next solution is in the loop till the time termination condition has been met.

The genetic algorithm Flowchart is given in figure-2.1

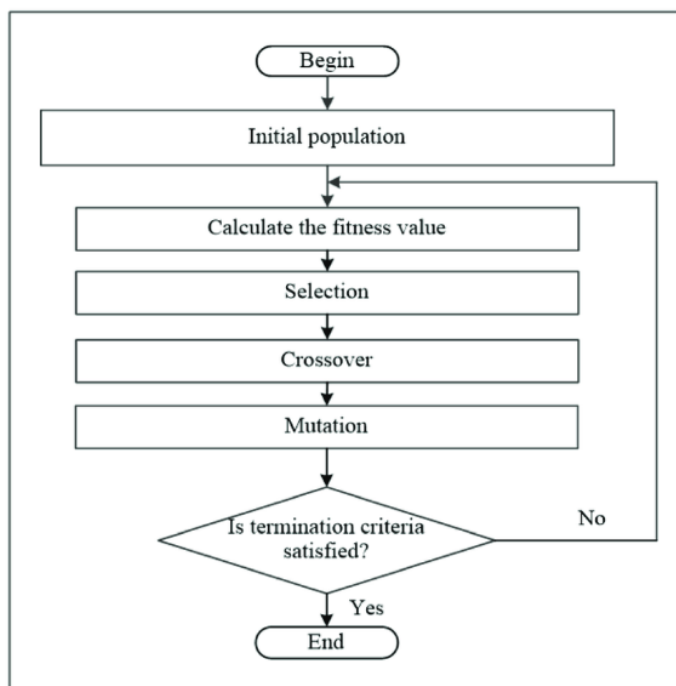


Fig. 2.1 Flowchart of the Genetic Algorithm [3].

Instead of being an algorithm for a particular problem, genetic algorithms are a type of a methodology for tackling optimisation problems. In other words, it's crucial to comprehend the fundamentals of genetic algorithms and implement it to the challenges you need because there isn't a single algorithm or source code that can be used to solve all problems.

G.A. has terms like chromosome, gene, offspring, and fitness which are prevalent when using G.A. A group of biologically genetic material known as a chromosome expresses a response to a genetic algorithm. A gene is a component of a chromosome that holds a single piece of genetic data. For instance, if a chromosome has the values [X Y Z], it has three genes, each of which has the letters X, Y, and Z. The term "offspring" refers to chromosomes produced by the mating of those in the preceding generation.

The genetic makeup of the progeny is carried over into the descendants. The offspring share information present in the genes of the previous generation. Fitness is a value of a chromosome that indicates how suited is the chromosome as a solution to the challenge. The way the G.A. works is to pick the chromosome that fits the best out of a group of chromosomes from a particular generation, and then repeat searches in that direction until an ideal answer is found.

The genetic algorithm progresses through a series of stages. Initially, we establish a collection of starting chromosomes. Subsequently, we evaluate the fitness of these initial chromosomes. Next, the algorithm generates offspring based on the existing chromosomes we also mutate the offspring so that there can be genetic diversity in the solutions and after that, we assess the fitness of the offspring. Once we establish the termination condition, if it is determined to be false, the process of creating offspring from the current chromosomes is repeated. However, if the termination condition is found to be true, the algorithm concludes and terminates.

The Pseudocode of Classic GA is given in figure-2.2.

```
Genetic_Algorithm() {  
    Initialize random population;  
    Evaluate the population;  
    Generation = 0;  
    While termination criterion is not  
    satisfied {  
        Generation = Generation + 1;  
        Select good chromosomes by  
        reproduction procedure;  
        Perform crossover with probability  
        crossover (Pc);  
        Perform mutation with probability  
        of mutation (Pm);  
        Evaluate the population;  
    }  
}
```

Fig. 2.2. Pseudo Code of the Classic Genetic Algorithm [4].

CHAPTER 3

LITERATURE SURVEY

B. T. M. Anh in [5] has proposed an enhanced genetic algorithm which is given in Figure-3. In this algorithm, we first find the targets which are to be present for each method of the class which is under the test. Following this, there will be initial population will be created which will be random. This population is predefined in this generation. A test case is each individual in the population. The evolution step takes place in a loop until a solution is found that covers the set of targets which need to be covered or the resources which were being used have been fully used (execution time, number of generations). We check if the current target is covered or not in every iteration of the evolution. The test case which takes care of the target is stored for the output. For each run of the loop of evolution, a new generation is then built and is mixed with the fittest individuals from the current population. The genetic operators (selection, crossover and mutation) are used in accordance with the fitness value of test cases.

Na Zhang, B. Wu and X. Bao have also introduced a new technique known as a multi-population genetic algorithm[6]. Simple genetic algorithm has some issues in it like being stuck in local Optimal or the convergence which is not mature. To fix the local optimal issue we can use a multi-population algorithm which helps us to get more quality test cases as a result. To test this method 10 C language programs were used as the tests. This method was tested against the normal genetic algorithm and the random algorithm for the generation of test suites. The results showed that the test suites produced by the above method had more code coverage than both the genetic algorithm and the random algorithm and the size of the test suites of the above method was also lower than the other two methods, which shows that test suites of this methods were covering more code having lesser test cases than the other two methods.

Test data generation is a pricey, tedious, and error-prone activity, according to *Harsh Bhasin*[7]. Making the automation of the method used to generate test cases as efficient and potent as possible is therefore urgently required. The work that was presented aimed to achieve maximum coverage by automating the test generation method. They chose programmes based on their lines of code and function, and these programmes served as the basis for testing their methodology. The outcomes of their research have been confirmed. Their ideas can be used to create a bigger system.

In [8], *Jungsup Oh* employed a messy GA to cover transitions in Simulink/Stateflow models. They unveiled a programme that applies their strategy and tests it against three standard Simulink models for embedded software. Compared to a professional tool for State Flow testing and random search, their messy GA can obtain statistically higher coverage.

A Ruby Test case Generator (RuTeG) tool was created by *Mairhofer, S.* in [9] and uses prior work on OOP test generation to produce test cases that maximise the coverage requirement with the help of a genetic algorithm. The results of this tool were compared with a random test generator tool and it showed significantly more code coverage than the random generator tool.

In a study presented by *Alsmadi* [10], the authors optimise the production of test cases from application UI by using genetic algorithms. This is accomplished by creating the graphical user interface controls graph and encoding each control's location in the graphical user interface graph to be uniquely represented.

R. Khan and A.K. Srivastava have introduced a new method for the generation of automatic test case formulation testing [11] using a Hybrid genetic algorithm.

The termination condition for this algorithm is to either get all the coverage to all the paths which are identified in a CFG or to cross 1000 iterations. To test this Hybrid Genetic Algorithm, Mutants were injected into the initial Programs, so to get the mutation score.

In this experiment, they had taken 10 test suits of different sizes and had done mutation analysis. The results were compared to the results which were gathered by using different algorithms(Random Algorithm, Genetic Algorithm and the proposed Hybrid Genetic Algorithm).

The results showed that the mutation score achieved by Hybrid Genetic Algorithm is far better than the other two algorithms.

CHAPTER 4

METHODOLOGY

4.1 GenTest Algorithm

GenTest algorithm is a slightly enhanced version of the Classic Genetic Algorithm which we discussed earlier. This Enhanced version of the Genetic Algorithm has two main differences from the classic genetic algorithm stated below.

This algorithm has adaptive mutation because of which the mutation rate depends upon the fitness value of the best individual in the population in comparison to the classic version of the algorithm in which the mutation rate is fixed and not dynamic.

In this algorithm, we also store the best individual with the highest fitness in the current generation to be part of the next generation. As we store the best individual across all the generations and also make it part of the forthcoming generation we can converge to an optimal solution more timely as compared to the classic version of the algorithm we do not have a guarantee that the best individual from a particular generation would pass on to the next generations.

The Psuedo Code for the GenTest algorithm is presented in figure-4.1

```
1 function GenTestAlgorithm():|
2   population <- initializePopulation()
3   best_individual <- None
4   best_fitness <- -inf
5
6   repeat until termination condition is met:
7     fitness_values <- evaluatePopulation(population)
8
9     for each individual in population:
10      if fitness_values[individual] > best_fitness:
11        best_individual <- individual
12        best_fitness <- fitness_values[individual]
13
14    new_population <- []
15    selected_parents <- selectParents(population, fitness_values)
16    offspring <- crossover(selected_parents)
17    offspring <- adaptive_mutate(offspring, best_fitness)
18    new_population <- replacePopulation(population, offspring)
19    new_population.append(best_individual)
20    population <- new_population
21  return best_individual
```

Fig. 4.1 Pseudo Code of the GenTest Algorithm

The algorithm has the following steps:-

1. Initialization: The algorithm starts by generating an initial population of individuals, where each individual represents a potential solution to the problem.
2. Fitness Calculation: The fitness values of the individuals in the population are calculated using a fitness function. In this case, the fitness function determines the quality or suitability of each individual's solution to the problem.
3. Tracking the Best Individual: The algorithm keeps track of the best individual found so far (best_individual) and its corresponding fitness value (best_fitness). Initially, these variables are set to None and 0, respectively.
4. Adaptive Mutation Rate: The mutation rate is determined dynamically based on the fitness of the best individual. A lower fitness value corresponds to a higher mutation rate, allowing for more exploration and diversification in the early stages of it. The adaptive mutation rate is calculated using the formula $\text{mutation_rate} = 1 / (\text{best_fitness} + 1)$.
5. Selection: The selection process chooses parents from the population to create the next generation. In this code, the selection is performed by randomly selecting individuals from the population as parents.
6. Crossover: Crossover is applied to the selected parents to produce offspring. The crossover operation combines the genetic information of the parents to create new solutions.

7. Mutation: Mutation introduces small random changes in the offspring's genetic information. In this algorithm, the mutation operation is applied to the offspring with a probability determined by the adaptive mutation rate. The mutation allows for exploration and the introduction of new genetic material into the population.
8. Elitism: The best individual found so far (best_individual) is preserved and added to the new population. This ensures that the best solution is not lost during the evolution process.
9. Next Generation: The new population is created by combining the best individual from the previous generation (elitism) and the offspring generated through crossover and mutation.
10. Termination: The algorithm proceeds for a fixed number of generations. In this code, the termination condition is defined by the value of the GENERATIONS constant.
11. Best Fitness Tracking: The best fitness value of each generation (best_fitness) is stored in a list (best_fitnesses) to track the progress of the algorithm over time.
12. Return: The algorithm returns the list of best individuals based on the best fitness value generations.

4.2 Advantages of GenTest over Classic G.A

Adaptive Mutation: The GenTest algorithm incorporates adaptive mutation, where the mutation rate is dynamically adjusted based on the fitness of the best individual. This adaptive mutation strategy provides several advantages:

a. **Exploration-Exploitation Balance:** By adjusting the mutation rate based on the fitness, the algorithm can strike a better balance between exploration and exploitation. In the early stages of the algorithm, when the fitness is low, a higher mutation rate promotes exploration, allowing the algorithm to search a wider range of solutions. As the fitness improves, the mutation rate decreases, enabling a more focused search around promising regions. This adaptive approach helps the algorithm efficiently explore the search space while also exploiting good solutions.

b. **Avoiding Premature Convergence:** The adaptive mutation rate helps prevent premature convergence by maintaining a higher mutation rate when the fitness is low. This ensures that the algorithm continues to explore the search space, avoiding getting trapped in local optima. By allowing for a higher exploration rate, the GenTest algorithm has a better chance of finding optimal or near-optimal solutions.

Tracking the Best Individual: In the GenTest algorithm, the best individual found so far (`best_individual`) and its corresponding fitness value (`best_fitness`) are stored and tracked throughout the generations. This tracking mechanism provides several benefits:

a. **Preserving the Best Solution:** By storing the best individual and its fitness value, the algorithm ensures that the best solution is not lost during the evolution process. The best individual is preserved and carried forward to subsequent generations, guaranteeing that the algorithm continues to improve upon the best solution found so far.

b. **Monitoring Progress:** Keeping track of the best individual and its fitness value allows for monitoring the progress of the algorithm over time. By examining the changes in the best fitness value across generations, it becomes possible to assess the performance of the algorithm and determine whether it is converging or stagnating.

c. **Better Final Solution:** The ability to track the best individual and its fitness value helps ensure that the algorithm converges to a high-quality solution. By retaining the best individual throughout the generations, the algorithm maintains a reference to the best solution found, which can lead to better final results.

In summary, the enhanced genetic algorithm with adaptive mutation improves the exploration-exploitation balance by adjusting the mutation rate based on fitness, thereby avoiding premature convergence and improving the chances of finding optimal or near-optimal solutions. Additionally, tracking the best individual throughout the generations ensures the preservation of the best solution and allows for monitoring the algorithm's progress, ultimately leading to better final solutions.

CHAPTER 5

IMPLEMENTATION

In this section, we present the implementation details of the GenTest Algorithm. We have applied this algorithm to two problems (Triangle Problem and Next Date Problem). The implementation is done using Python programming language. The code provided below incorporates the modifications discussed in the methodology section, including adaptive mutation and tracking the best individual.

Programming Language: Python

5.1. Code Implementation:

The figure-5.1 shows the code implementation of the GenTest Algorithm with adaptive mutation for test data generation in the Next Date Problem problem:

```
1 def GenTest_algorithm():
2     population = generate_population()
3     best_individual = None
4     best_fitness = 0
5     best_fitnesses = []
6     mutation_rate = 0.01
7
8     for generation in range(GENERATIONS):
9         # Calculating fitness for each individual
10        fitness_values = [calculate_fitness(day, month, year) for day, month, year in population]
11
12        # Finding the best individual with best fitness
13        max_fitness = max(fitness_values)
14        max_index = fitness_values.index(max_fitness)
15        if max_fitness > best_fitness:
16            best_individual = population[max_index]
17            best_fitness = max_fitness
18
19        best_fitnesses.append(best_fitness)
20
21        # Calculating adaptive mutation rate
22        mutation_rate = 1 / (best_fitness + 1)
23
24        # Creating the next generation
25        new_population = []
26
27        # Performing crossover and mutation to create the rest of the new population
28        while len(new_population) < POPULATION_SIZE-1:
29            parent1 = random.choice(population)
30            parent2 = random.choice(population)
31            child = crossover(parent1, parent2)
32            child = mutate(child, mutation_rate)
33            new_population.append(child)
34
35        # Add the best individual to the new population
36        new_population.append(best_individual)
37        population = new_population
38
39    return best_fitnesses
```

Fig. 5.1 Implementation of the GenTest Algorithm

5.2. Considerations:

In the implementation, we have adhered to the principles of genetic algorithms, including representation, fitness calculation, selection, crossover, and mutation. The modifications introduced in the enhanced algorithm are as follows:

Adaptive Mutation: The `calculate_adaptive_mutation_rate` function dynamically adjusts the mutation rate based on the fitness of the best individual. The formula $\text{mutation_rate} = 1 / (\text{best_fitness} + 1)$ is used to calculate the adaptive mutation rate.

Tracking the Best Individual: The best individual found so far and its corresponding fitness value are stored in the `best_individual` and `best_fitness` variables, respectively. The `best_fitnesses` list is used to track the best fitness values across generations.

The implementation also includes other necessary functions and code snippets to support the main algorithm, such as generating the initial population, calculating fitness values, selecting parents, performing crossover and mutation operations, and generating random individuals.

5.3. Parameter Tuning:

In the implementation, certain parameters need to be tuned to achieve optimal results. These parameters need to be configured according to the type of problem we are solving. These parameters include the population size, the number of generations, and the specific ranges for generating random individuals (e.g., day, month, year). The values of these parameters can be adjusted based on the specific requirements and characteristics of the triangle problem/next date problem or the test data generation scenario.

CHAPTER 6

RESULT AND ANALYSIS

In this section, we present the results and analysis obtained from running the enhanced genetic algorithm with adaptive mutation for test data generation in the triangle problem. The algorithm was implemented using Python programming language, incorporating the modifications discussed in the methodology section.

6.1. Experimental Setup:

Triangle Problem:-

Population Size: 100

Number of Generations: 800

Next Date Problem:-

Population Size: 100

Number of Generations: 300

6.2. Results:

The results of the GenTest algorithm with the adaptive mutation are compared with the classic genetic algorithm for the test data generation in the triangle problem and the next date problem. The performance metrics considered are the best fitness values achieved over generations and the time taken for execution.

6.3. Performance Comparison:

The following figures shows the graph of the comparison of the best fitness values achieved by the GenTest algorithm and the classic genetic algorithm for both of the Problems.

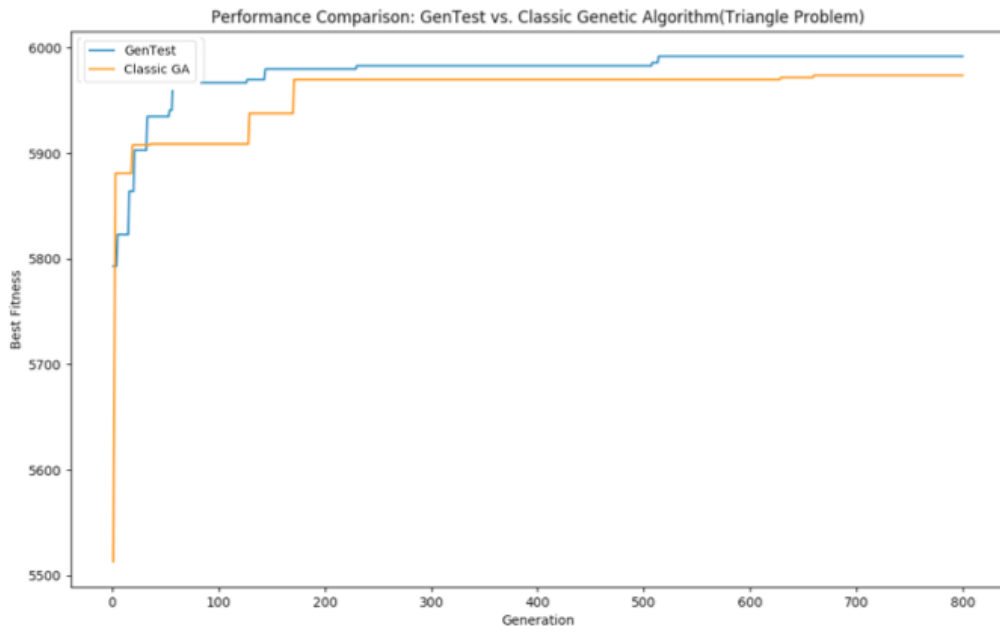


Fig. 6.1 Comparison of the GenTest Algorithm with Classic G.A(Triangle Problem)

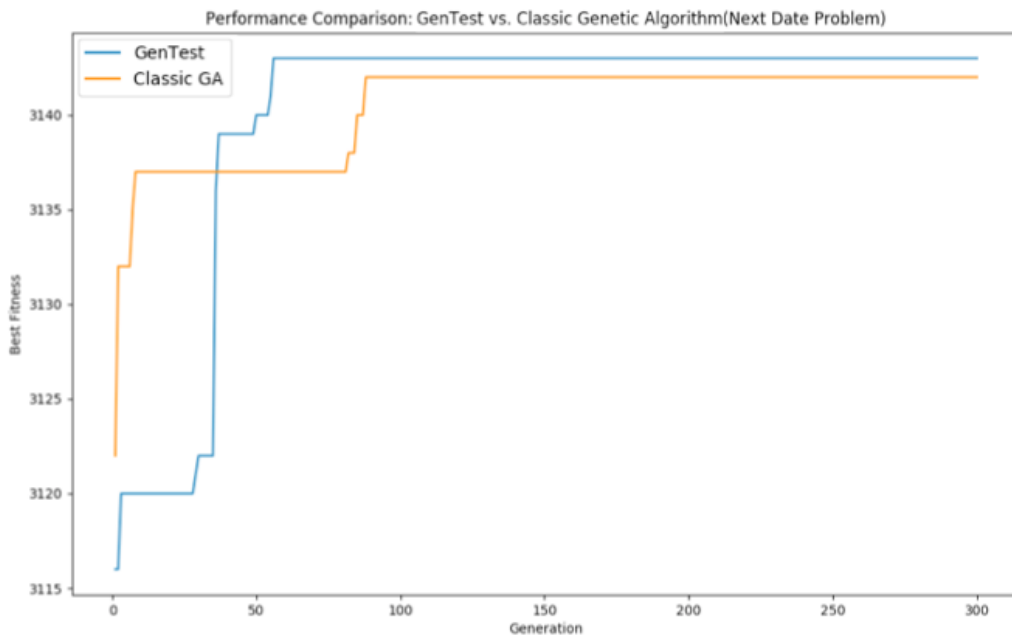


Fig. 6.2 Comparison of the GenTest Algorithm with Classic G.A(Next Date Problem)

6.4. Analysis

Fitness Values: The comparison graph demonstrates that the GenTest algorithm with adaptive mutation consistently outperforms the classic genetic algorithm in terms of fitness values. The best fitness achieved by the GenTest algorithm improves at a faster rate and converges to a higher value compared to the classic algorithm. This indicates that the adaptive mutation and best individual tracking mechanisms enhance the algorithm's ability to find better solutions.

Convergence Rate: The GenTest algorithm shows a faster convergence rate compared to the classic algorithm. It converges towards the optimal or near-optimal solutions more efficiently, thanks to the adaptive mutation that allows a better exploration-exploitation balance throughout the optimization process.

Solution Quality: The GenTest algorithm, with its ability to track the best individual, consistently produces better solutions compared to the classic algorithm. By preserving and carrying forward the best individual, the GenTest ensures continuous improvement and a higher probability of finding the optimal solution.

CHAPTER 7

CONCLUSION AND FUTURE WORKS

7.1. Conclusion

In this project, we implemented an enhanced genetic algorithm known as GenTest Algorithm with adaptive mutation for test data generation in the triangle problem and next date problem. The algorithm was compared with the classic genetic algorithm to evaluate its performance and effectiveness. The implementation was done using Python programming language, incorporating the modifications discussed in the methodology section.

Through our implementation and analysis, we have arrived at the following conclusions:

1. **Performance Improvement:** The GenTest algorithm with adaptive mutation outperforms the classic genetic algorithm in terms of fitness values, convergence rate, and solution quality. The adaptive mutation and best individual tracking mechanisms contribute to better exploration-exploitation balance, faster convergence, and higher-quality solutions. The adaptive mutation ensures efficient exploration in the early stages and focused exploitation around promising regions as the fitness improves, leading to improved optimization outcomes.

2. **Solution Preservation:** The best individual and its fitness value are tracked throughout the generations in the GenTest algorithm. This tracking mechanism guarantees that the best solution found so far is preserved and carried forward. By maintaining a reference to the best individual, the algorithm continuously improves upon the best solution and increases the chances of finding optimal or near-optimal solutions.

3. Computational Efficiency: The GenTest algorithm, with its adaptive mutation and best individual tracking, may have a slightly higher execution time compared to the classic algorithm due to the additional computation involved in adapting the mutation rate. However, the difference in execution time is generally negligible and does not significantly impact the overall computational efficiency.

In conclusion, the GenTest algorithm with adaptive mutation provides significant improvements over the classic genetic algorithm for test data generation in the triangle problem. By dynamically adjusting the mutation rate and tracking the best individual, the algorithm achieves better optimization outcomes, faster convergence, and higher-quality solutions. The GenTest algorithm can be applied in various software testing scenarios where test data generation is required.

7.2. Future Works

Future research directions could include exploring additional enhancements to the algorithm, such as incorporating other evolutionary operators, exploring different adaptation strategies for mutation, or applying the GenTest to other test data generation problems. Further optimization and experimentation could also be conducted to fine-tune the algorithm's parameters and assess its performance on larger and more complex test data generation scenarios.

Overall, the GenTest algorithm with adaptive mutation presents a promising approach to address the test data generation challenges in the triangle problem and has the potential to be applied in real-world software testing scenarios, contributing to improved software quality and reliability.

REFERENCES

- [1] Bhattacharyya, Trinav & Chatterjee, Bitanu & Singh, Pawan & Yoon, Jin & Geem, Zong Woo & Sarkar, Ram. (2020). Mayfly in Harmony: A New Hybrid Meta-Heuristic Feature Selection Algorithm. IEEE Access. 8. 10.1109/ACCESS.2020.3031718.
- [2] Jarmo.T.Alander, TimoMantere, Genetic Algorithm Based Software Testing, Department Of Information Technology, University Of Vassa,Finland, 2005
- [3] Albadr, M.A.; Tiun, S.; Ayob, M.; AL-Dhief, F. Genetic Algorithm Based on Natural Selection Theory for Optimization Problems. Symmetry 2020, 12, 1758.
- [4] Peeyee, Muhammad & Rahman, Shuzlina & Abdul Hamid, Nurzeatul & Zakaria, Zaki. (2019). Heuristic-based model for groceries shopping navigator. Indonesian Journal of Electrical Engineering and Computer Science. 16. 932. 10.11591/ijeecs.v16.i2.pp932-940.
- [5] B. Thi Mai Anh, "Enhanced Genetic Algorithm for Automatic Generation of Unit and Integration Test Suite," 2020 RIVF International Conference on Computing and Communication Technologies (RIVF), 2020, pp. 1-6
- [6] Zhang, Na Biao, Wu Bao, Xiaoan. (2015). Automatic Generation of Test Cases Based On Multi-population Genetic Algorithm. International Journal of Multimedia and Ubiquitous Engineering. 10. 113-122 10.14257/ijmue.2015.10.6.11.
- [7] Harsh Bhasin, Neha Singla, Shruti Sharma, Cellular Automata Based Test Data Generation, ACM SIGSOFT Software Engineering Notes, July 2013 Vol 38, No4. Mairhofer, S. et al.: Search-based Software Testing and Test Data Generation for a Dynamic Programming Language (2011)
- [8] Jungsup Oh, Mark Harman, Shin Yoo, Transition Testing for Simulink/Stateflow Model Using Messy Genetic Algorithms, ACM, GECCO'11, July 12-16, 2011, Dublin Ireland
- [9] Mairhofer, Stefan & Feldt, Robert & Torkar, Richard. (2011). Search-based software testing and test data generation for a dynamic programming language. Genetic and Evolutionary Computation Conference, GECCO'11. 1859-1866. 10.1145/2001576.2001826.
- [10] Izzat Alsmadi, Using Genetic Algorithms For Test Case Generations And Selection Optimization, Electrical and Computer Engineering 1-4, 2010.
- [11] R. Khan, M. Amjad and A. K. Srivastava, "Generation of automatic test cases with mutation analysis and hybrid genetic algorithm," 2017 3rd International Conference on Computational Intelligence Communication Technology (CICT), 2017, pp. 1-4.