T.C
SAKARYA UNIVERSITY
INSTITUTE OF SCIENCE AND TECHNOLOGY

TEST DATA GENERATION FOR DYNAMIC UNIT TEST IN JAVA LANGUAGE USING GENETIC ALGORITHM

M.Sc. THESIS

Zhela Jalal RASHID

Department	**:**	**COMPUTER AND INFORMATION ENGINEERING**
Supervisor	**:**	**Assist. Prof. Dr. M. Fatih ADAK**

August 2021

T.C
SAKARYA UNIVERSITY
INSTITUTE OF SCIENCE AND TECHNOLOGY

TEST DATA GENERATION FOR DYNAMIC UNIT TEST IN JAVA LANGUAGE USING GENETIC ALGORITHM

M.Sc. THESIS

Zhela Jalal RASHID

Department	**:**	**COMPUTER ENGINEERING**
Field of Science	**:**	**COMPUTER SCIENCE AND TECHNOLOGY**
Supervisor	**:**	**Assist. Prof. Dr. M. Fatih ADAK**

This thesis has been accepted unanimously / with majority of votes by the examination committee on 12.08.2021

Head of Jury **Jury Member** **Jury Member**

DECLARATION

I hereby declare that the thesis entitled “Test data Generation for dynamic unit test in Java language using Genetic Algorithm” which is being submitted to the Sakarya University, in partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Engineering is an authentic work carried out by me.

The material contained in this thesis has not been submitted to any university or any Institution for the award of any degree.

Zhela RASHID

12.08.2021

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank my supervisor Dr. M. Fatih ADAK for his invaluable and consistent guidance throughout this work. I would like to thank him for giving me the opportunity to undertake this topic.

I have been accompanied and supported by many people including my family and Friends. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

No thesis could be written without being influenced by the thoughts of others. I would like to thank my husband Rebaz Saleh who were always there at the hour of the need and provided with all the help and support, which I needed.

Most importantly, I would like to give God the glory for all of the efforts I have put İnto this report.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS…………………………………………………… i

TABLE OF CONTENTS………………………………………………….. ii

LIST OF TABLES……………………………………………………… v

LIST OF FIGURES…………………………………………………...... vi

SUMMARY……………………………………………………………. vii

ÖZET ……………………………………………………………………. viii

CHAPTER 1.

INTRODUCTION……………………………………………………………. 1

1.1. Related Work ………………………………………………………… 2

1.2. Unit testing in object-oriented applications…………………………….. 6

1.3. Test Data Generation ………………………………………………. 6

1.4. Objective and Aims of the Study ……………………………………….. 6

1.5. Structure of the Thesis………………………………………………... 7

CHAPTER 2.

SOFWARE TESTING PRINCIPLES……………………………………………… 8

2.1. Types of Testing…………………………………………………… 8

2.1.1. Black box or functional testing………………………………….. 8

2.1.2. White box testing or structural testing………………………….. 9

2.2. Automated Test Data Generation……………………………………… 10

CHAPTER 3.

GENETIC ALGORITHM……………………………………………………… 12

3.1. History of GeneticAlgorithm………………………………………….. 12

3.2. Search Space Challenge……………………………………………….. 13

3.3. Representation of Each Individual……………………………………. 13

3.4. Basic Steps of Genetic Algorithm…………………………………….. 14

3.5.Chromosomes Representation(Encoding)……………………………. 16
3.5.1. Binary encoding……………………………………………….. 16
3.5.2. Tree encoding………………………………………………….. 16
3.6.Selection………………………………………………………………. 17
3.6.1. Roulette wheel selection……………………………………….. 17
3.6.2. Tournament selection…………………………………………… 18
3.6.3. Elitism………………………………………………………….... 20
3.6.4. Steady-state selection………………………………………….... 21
3.7.Crossover……………………………………………………………… 21
3.7.1. Single point crossover………………………………………….. 22
3.7.2. Two point crossover……………………………………………. 22
3.7.3. Uniform crossover…………………………………………….... 22
3.8. Mutation………………………………………………………………. 23
3.9. Population Size and Elitism Rate…………………………………….. 23
3.9.1. Population size…………………………………………………. 23
3.9.2. Elitism rate……………………………………………………... 23

CHAPTER 4.
METHODOLOGY AND EXPERIMENTS……………………………………… 24
4.1.Experimental Settings………………………………………………….. 24
4.2.Selection Strategies……………………………………………………. 28
4.3.The Fitness Evaluator ………………………………………………… 29
4.4.Termination Conditions……………………………………………….. 30
4.5.Model Evaluation……………………………………………………... 31

CHAPTER 5.
RESULTS AND DISCUSSION………………………………………………. 34
5.1.Performance Measures………………………………………………... 34
5.2.Iterations……………………………………………………………… 35
5.3.Performance Measures for Each Method of the Class……………….. 37

CHAPTER 6.
CONCLUSION AND FUTURE WORK……………………………………….. 39

REFERENCES……………………………………………………………… 55
APPENDIX ……………………………………………………………….. 59
RESUME ………………………………………………………………….. 63

LIST OF TABLES

Table 4.1. SampleClass structure ……………………………………………............ 31
Table 5.1. Elapsed time for each iteration in seconds ……………………………. 35
Table 5.2. Summary of the data …………………………………………………….. 36
Table 5.3. Summary of performance for all methods …………………………….. 37
Table 5.4. Sample input data generated by the algorithm ………………………… 38

LIST OF FIGURES

Figure 2.1. General specification of black box testing 9
Figure 2.2. General specification of white box testing 10
Figure 3.1. Example of a search space.. 13
Figure 3.2. General flow chart for genetic algorithm 15
Figure 3.3. Binary encoding 16
Figure 3.4. Tree encoding 17
Figure 3.5. Roulette wheel selection method 18
Figure 3.6. Roulette wheel selection algorithm 18
Figure 3.7. Tournament selection method 20
Figure 3.8. Elitism selection algorithm 21
Figure 3.9. Steady state selection method 21
Figure 3.10. Single point crossover 22
Figure 3.11. Two point crossover 22
Figure 3.12. Uniform crossover 22
Figure 3.13. Bit inversion mutation 23
Figure 4.1. A method to demonstrate transformation to a GA compatible code ... 25
Figure 4.2. The proposed technique flowchart 27
Figure 4.3. An example of countable and uncountable exception method code... 28
Figure 4.4. Get candidate fitness value function code 29
Figure 4.5. Sample class structure code 32
Figure 4.6. Sample class structure code (continued) 33

SUMMARY

Keywords: Software Testing, Unit Testing, Test Data, Evolutionary Algorithm, Genetic Algorithm

Software testing is an essential and fundamental part of the software development lifecycle. Testing helps delivering a higher quality system to end users. However, it is costly as it needs to be written and updated as the process continues to ensure that it does its job in detecting faults and bugs in the system.

One of the problems in maintaining test cases is the input data used to run the tests such a way that it covers each path and line of code of the software product. Generation of these test data is a typical activity which has to be accomplished through any standard automated test data generation tool.

Random test data generators are among the most widely used tools to generate input data for the tests. However, the data types and parameters has to be manually tailored into the tools and needs to be updated manually once the source code or the test cases are changed. It is a costly process and takes a lot of time and effort to generate and update these data.

There are various test data generator tools are available such as: random test data generator, symbolic evaluator, function minimization methods. In recent years some more advanced heuristic search techniques have been applied to software testing.

In this study, we propose a model which automates the test data generation process. It significantly reduces the time required to generate the input data. At the same time, the data generated by our model outperforms the data generated randomly in terms of accuracy and sensibility of the input data. It is based on the most widely used heuristic algorithms which is genetic algorithm.

We run the model on a sample class with 6 independent public methods of different method signature, return type and number of arguments. It takes 5 seconds to generate10 possible inputs for each method with a mean standard deviation of 0.15 and best candidate fitness average of 8.82 and mean fitness of 9.79. The results will be further discussed in results section of the study.

GENETİK ALGORİTMA KULLANARAK JAVA DİLİNDE DİNAMİK BİRİM TESTİ İÇİN TEST VERİSİ ÜRETİMİ

ÖZET

Anahtar Kelimeler: Yazılım Testi, Birim Testi, Test Verisi, Evrimsel Algoritma, Genetik Algoritma

Yazılım testi, yazılım geliştirme yaşam döngüsünün önemli ve temel bir parçasıdır. Test etme, son kullanıcılara daha kaliteli bir sistem sunmaya yardımcı olur. Fakat, süreç, sistemdeki hataları tespit etme işlemini yerine getirdiğinden emin olmak için yazılması ve sürekli güncel tutulması gerektiğinden maliyetli bir işlemdir.

Test senaryolarını sürdürmedeki sorunlardan biri, testleri yazılım ürününün her bir yolunu ve kod satırını kapsayacak şekilde çalıştırmak için kullanılan girdi verileridir. Bu test verilerinin oluşturulması, herhangi bir standart otomatikleştirilmiş test verisi oluşturma aracıyla gerçekleştirilmesi gereken tipik bir faaliyettir.

Rastgele test veri oluşturucuları, testler için girdi verileri oluşturmak için en yaygın kullanılan araçlar arasındadır. Ancak, veri türleri ve parametrelerin araçlara göre manuel olarak uyarlanması ve kaynak kodu veya test senaryoları değiştirildiğinde manuel olarak güncellenmesi gerekir. Bu maliyetli bir süreçtir ve bu verileri oluşturmak ve güncellemek çok zaman ve çaba gerektirir.

Rastgele test verisi oluşturucu, sembolik değerlendirici, fonksiyon minimizasyon yöntemleri gibi çeşitli test verisi oluşturucu araçları mevcuttur. Son yıllarda, yazılım testine bazı daha gelişmiş iyileştirilmiş arama teknikleri uygulanmıştır.

Bu çalışmada, test verisi oluşturma sürecini otomatikleştiren bir model önerilmiştir. Giriş verilerini oluşturmak için gereken süreyi önemli ölçüde azaltmaktadır. Aynı zamanda, önerilen modelde, üretilen veriler, giriş verilerinin doğruluğu ve duyarlılığı açısından rastgele oluşturulan verilerden daha iyi performans göstermiştir. Önerilen modelde en yaygın kullanılan sezgisel algoritmalara dayanan Genetik algoritma kullanılmıştır.

Geliştirilen model, farklı yöntem imzası, dönüş türü ve argüman sayısı olan 6 bağımsız genel yöntemle örnek bir sınıf üzerinde çalıştırılmıştır. Ortalama standart sapma 0,15 ve en iyi aday uygunluk ortalaması 8,82 ve ortalama uygunluk 9,79 olan her bir yöntem için 10 olası girdi üretmek ortalama 5 saniye sürmüştür. Sonuçlar çalışmanın sonuçlar bölümünde detaylı bir şekilde tartışılmıştır.

CHAPTER 1. INTRODUCTION

Software testing is one of the areas that are gaining in importance today. Especially Test-Oriented development and agile software processes are a proof of this. Whether the test is performed at the beginning or at the end, what is important is that the test is performed with good data [1].

Writing a good test to the developed software will reduce the maintenance cost in the following processes and will return to the company as a plus gain. Since it is not possible to test all inputs in the test data universal set in software unit test, if the subset to be selected well passes the test, all inputs in the universal set are considered to have passed the test [2]. At this point, it is of great importance to select a good test data. Manually generating test data can take time and selecting data to represent the universal set can be very difficult. Therefore, creating automatic test data with various methods is one of the preferred methods today.

The methods used can be random, static analysis, symbolic execution, search-based and heuristic algorithms. In the production of search-oriented test data, the focus is on determining the situations where the function will fail [3]. In this thesis, the success or failure rates of the test data to be produced using the genetic algorithm and the test unit test will constitute the suitability value of the genetic algorithm.

Although there are similar studies in the literature, the studies conducted in the target area of this study are limited and their focus is different. For example, satisfactory results have been obtained in a study that focuses more on the control flow in the program code and produces test data with the Genetic algorithm [4]. Test data was generated by using the genetic algorithm but this time focusing on the data flow [5]. In a study producing unit test data for static analysis, symbolic execution for the rule was used [6].

In a study in which the genetic algorithm was used in unit testing, they performed the random unit test and completed the same test in 10 percent of the normal test time [7]. Again, using the genetic algorithm, the synthetic program code, which was successfully produced for the production of test data, was used [8]. In another different approach, testing was applied for the Java language using the test data state matching technique [9].

In another study in which the genetic algorithm was used unsuccessfully, they stated that using the genetic algorithm would be a good choice in terms of code coverage [10]. Using advanced search strategies on the C code, test data was generated for dynamic unit testing, and it was found that the code was more successful than traditional methods in terms of coverage [11]. Unit tests to be carried out in this way will be more powerful tests and will affect the software quality positively [12].

In the light of these studies, it is seen that software testing is important as well as the quality of the data to be selected in this test. Therefore, in this thesis, test data will be produced for dynamic unit test in Java language using the Genetic Algorithm, which is known to be successful in generating data.

1.1. Related Work

Xanthakis et al. was the first to use a genetic algorithm to generate test data. With the support of a genetic algorithm, test data was created for this implementation for the structures that were not covered in the random search. Genetic Algorithm was used for generating the test data for all the branch predicates [13].

Pei, M., E. D. Goodman et al. proposed a test data generator for single-path coverage using the genetic algorithm technique. In his survey, he discovered that the majority of test-data generators used symbolic evaluation, which was common at the time. They concluded that dynamic testing was ineffective and static testing was impractical [14].

Roper, M et al. In 1995 created a test data generator based on a genetic algorithm with the aim of traversing all of a source code’s possible branches. The generator is given

a program, which is automatically instrumented and provides input on the achieved branch coverage [15].

Michael et al. created GADGET (Genetic Algorithm Data Generation Tool), a tool that produces test data and allows a program to be instrumented automatically without the use of a programming language. The only restriction was that only scalar inputs could be accepted. GADGET is the first test data generator that has been thoroughly tested for a large-scale real-world problem, b737 (part of autopilot system i.e. real-world control software). They came to the conclusion that the test data generator, which uses a random approach to produce data, does not work well for large problems [16].

Tracey et al. developed a test-case data generator based on the optimization technique. A large range of test parameters for both functional and non-functional properties may be used in this. Specification errors and exception conditions are subjected to optimization. Optimization is applied to testing specification failures and exception conditions. Various case studies are seen in this production to demonstrate the efficiency of this 2 optimization technique for producing test case data [17].

Pargas et al. improvised the outcome of Jones et al. work. Previously, branch knowledge was used to evaluate fitness functions, but here, the control dependency graph was used to evaluate fitness. According to them, this approach provides a more accurate fitness function than the Jones and Michael approaches discussed previously. He also mentioned that this technique with minor changes can also provide path coverage [4].

Wegene et al. developed a Genetic Algorithm-based test data generator for real-world embedded software in 2002, with a focus on white box testing, especially statement and branch coverage. The test focuses on certain partial goals that are difficult to achieve. The stopping conditions are met when all of the branches have been covered or when the number of generations has been reached [18].

Hermadi et al. used genetic algorithm to generate test data for path testing in 2003. A collection of test data was developed using this method for a set of target paths. With this approach, better path coverage was achieved, and efficiency was improved in terms of 1) Search space exploitation, 2) Exploration, and 3) Allows for fast convergence [19].

Tonella P et al. created a test case generator based on a genetic algorithm for unit testing in a generic scenario in 2004. Chromosomes are used as test cases in this approach and they provide information about the objects that must be produced, the methods that must be called, and the values that must be used as input. This algorithm performs mutations with the objective of maximizing a coverage metric. The paper explains how this algorithm is implemented and extended to Java standard library classes [20].

Zhang et al. introduced two fitness functions in 2009, one focused on normalized extended hamming distance (SIMILARITY) and the other on branch distance (BDBFF), both of which were applied to GA based test data generation with focusing on path orientation. For comparing the output of both the fitness functions, a triangle classification program was chosen as an example [21].

Cao, Yang et al. introduced a genetic algorithm-based method for generating test data for a single particular path in 2009. To measure the fitness of each individual in the population, a genetic algorithm was used to find the best solution by selecting the fitness value as the closeness of the execution path and target path with overlapping sub paths. The proposed fitness function was tested in a few experiments, and the function's efficiency was calculated in terms of consumed time and convergence potential [22].

In Rauf and Anwar, a GUI-based test criteria to generate software test data presented. GUI applications were event driven and used GA to generate software test data [23].

In Rauf and Anwar, a GUI-based test criteria to generate software test data presented. GUI applications were event driven and used GA to generate software test data [23].

In 2011 Malhotra et al. proposed another technique based on adequacy-based test data generation, it uses mutation analysis then execute test data generation. Target of GA in the study get the optimum solution. Adequacy-based technique was found better than path testing technique in terms of number of test cases created and the duration taken to create those test cases in study [24, 25, 26].

There has been other research studies that propose different techniques that would possibly replace meta-heuristic and search-based software testing. Lee et al. is one of the studies and proposed fitness evaluator program (FEP). To evaluate their approach, they implemented in a tool which is called ConGA. This tool was then used to evaluation its performance by running on multiple programs written in C language. They come to a conclusion that their proposed idea reduces the test data generation process as compare to other tools [25, 27].

In terms of implementing test data generation tools on different systems, McMinn et al. Have applied a GA-based generated for large scale programs and then concluded that they perform much better than other search algorithms. In addition, to further help addressing test data generation for enterprise applications, Fraser et al. created EvoSuite which automatically generate unit test for Java applications using JUnit [30, 31, 32].

Previous studies have used both gradient descent and GA to find test input for a test case. There are some limitations with gradient descent as it struggles to deal with local optima. Other studies, for example GADGE, uses GA to generate test input for test cases. However, it is more concerned about generating test input that would cover the most paths possible regardless of their semantic aspect of the input.

We try to overcome the limitations of previous studies by first: incorporating GA in the test data input generation process as well as generating data that would sound more natural than the traditional approach. This is done by introducing a new way to deal with exceptions and categorize them based on countable and uncountable. This way it helps maximizing the fitness of the individuals and allows more clear, understandable, readable and natural generation of data.

Finally, we apply our technique in a tool that can be easily integrated and implemented in any java project regardless of the size of the project. It dynamically adapts to the methods and provides input through successive generations of data and tune them using GA operators to maximize its performance. It is fast, highly scalable and can be easily plugged into any java project.

1.2. Unit testing in object-oriented applications

One of the crucial steps in SDLC is unit testing. By testing the units or small pieces of the application, we make sure that the applications functions properly and it does it correctly what we expect from it. To ensure this, software tester, engineers and developers write tests for each and every important feature or functionality in the system. It is obvious that it is a daunting process and takes lot of time and effort as well as a high cost. To overcome these tradeoffs with writing tests, several studies have been performed to further ease this process and reduce time and effort.

1.3. Test Data Generation

Generators specifically designed for creating test-data can be used to automate the testing process and automatically generate test data for the applications. Basically, the generation process encompasses a set of techniques used to determine the optimal set of data used to examine the selected criteria. This includes (path, branch, statement and etc.) coverage. [14].

1.4. Objective and Aims of the Study

To deliver high quality software systems, testing is written to detect bugs in the early stages of the development. This partially ensures customer satisfaction with the product and makes modification and maintain of the software much easier and affordable in the future. However, it comes at a cost of time and budget especially in the early stages of SDLC. For this reason, several tools and techniques have developed

and proposed to facilitate this process and write tests in a shorter period of time with much less effort. The idea of automating this process constitutes the object of our work.

In this study, we propose a model that helps generating test data automatically for unit tests written for application units. To achieve this, the genetic algorithm, which is among the most popular methods in the area and proven to be very successful, is used. The goal of the thesis is to generate input data that minimize the errors that occur during software develop process and to detect them during the test phase.

1.5. Structure of the Thesis

Following this introductory chapter:

Chapter 2: There are many techniques for software testing. In this chapter, we go through main techniques and briefly explain their pros and cons.

Chapter 3: This chapter shines light on the idea behind GA. What is it? What is the aim of using it? What are some of its applications with example? Then we go through main GA operators and their functions and usage as well as challenges associated with their implementation.

Chapter 4: In this chapter we described briefly the tool and the implementation of Genetic Algorithms for generating effective and efficient test data.

Chapter 5: In this chapter the results and the output of the tool are explained with clarification using graphs and charts.

Chapter 6: Presents the scope of the study conclusion and future-work to be conducted

CHAPTER 2. SOFWARE TESTING PRINCIPLES

This chapter discusses various software testing techniques which are being used widely to make software product stable and fault free. The main objective of software testing is to increase software product quality and reliability to make minimal software product error [1].

Below are the principles of Software Testing [35]:
1. The main purpose of system testing is to uncover errors as much as possible.
2. Testing process starts with the smallest part of the program which is a unit then to the largest part which are modules. As the testing process progress the aim of it will be finding errors in integrated clusters of modules and ultimately in the entire system.
3. It is impossible to conduct exhaustive testing. Even for a moderately sized programs, the number of paths is enormous. So it is difficult to test every possible combination of paths.
4. Testing process to be most effective should be done by independent third party. The term “most effective” refers to the type of testing that has the best chance of detecting errors.

2.1. Types of Testing

Another categorization for software testing is [36]:

1. Black- box Testing
2. White- box Testing

2.1.1. Black box or functional testing

In black box testing, a software product is considered as a black box and its implementation logic, structure and intelligence is not considered during testing phase.

The only objective is to provide input parameters and to take output of the software product with respect to given inputs. During black box testing inputs/ output are analyzed as shown in

Figure 2.1. General specification of black box testing

Black box testing attempts to find errors in the following categories:

A. Function missing

B. Errors in the interface

C. Data structure error

D. Performance errors

E. Termination and Initialization

2.1.2. White box testing or structural testing

Software engineers can write test using white box testing that:

A. Each path within the code has been traversed.

B. Make sure that logical conditions meet their criteria.

C. Assure its validity by exercising its internal structure.

D. Loops are visited at least once within its boundaries.

White Box Testing Approach

Figure 2.2. General specification of white box testing

2.2. Automated Test Data Generation

Software testing is an important and fundamental part of the software development life cycle. Despite its criticality and value in ensuring the stability of software products. Software testing has certain limitations and problems. One of the problems in software testing is to generate a set of data for testing the software product.

Primary objective of these testing data set is to cover each path and line of code of the software product [3]. Generation of these test data is a typical activity which has to be accomplished through any standard automated test data generation tool. There are various test data generator tools are available such as: random test data generator, symbolic evaluator, function minimization methods [38].

Recently, different algorithms have been used in the area to improve its efficiency. These algorithms are based on advanced heuristic approach among them is evolutionary algorithm. It is found that they perform much better than random generator in most of the time [39].

Meta-heurist search technique is used by evolutionary testing using GA. As a result of the searching process of this method, test parameters are generated to satisfy a set of predefined criteria for each test. The process includes an objective function aka (fitness

function) to measure the effectiveness of each generated input and facilitate the algorithm in the selection decision.

CHAPTER 3. GENETIC ALGORITHM

Genetic algorithms [40] are a subset of evolutionary computation a branch of artificial intelligence. It's an adaptive heuristic search approach based on natural selection and genetics in evolutionary theory. The basic concept of the Genetic Algorithm (GA) is to simulate processes in natural systems that are required for evolution, especially those that follow the survival of the fittest principles. Generally it is used in those cases where the search space is wide and cannot be easily traversed using traditional search methods.

A population, individual, chromosome and gene are the most important elements in genetic algorithm. At the beginning, the algorithm generates an initial population in order to begin with selecting fittest individuals gradually. These selected candidates contribute to the upcoming population so that it helps with discarding least fit candidate from the population. This process continues until the algorithm come to a conclusion that it has selected fittest candidates or it never gets to the optimal solution which is called termination condition as shown in Figure 3.2.

3.1. History of Genetic Algorithm

Ingo Rechenberg was the first who introduced evolutionary computing in the 1960s in his work "Evolution strategies". His concept was then further developed by other researchers. Later John Holland invented Genetic Algorithms (GAs), which he and his students and colleagues created and their work published under a book named "Adaptation in Natural and Artificial Systems" in 1975 [41].

John Koza in 1992 was used genetic algorithm to do tasks in the application programs. He named this method "Genetic Programming" (GP) [42].

3.2. Search Space Challenge

When a problem occurs, a set of solution is proposed in order to overcome it. The goal is to select the most efficient solution. Search space encompasses all available solutions while each solution represents a possible solution for the problem. Each point in the search space represents one possible solution [43].

Fitness value is used to rank each solution for the problem. GA strives to select the solution with the best fitness value. This is the major factor that helps with deciding which solution to select either (maximum or minimum) depending on the underlying problem.

One of the challenges in dealing with such problems is the size of its search space. Due to its large size it might be difficult to decide where actually to start. For this reason, many techniques and methods have been used to deal with this issue, for example, hill climbing, simulated annealing and GA and etc. The solutions selected by these algorithm are considered optimal as it outperforms its random selected counterpart by huge margin.

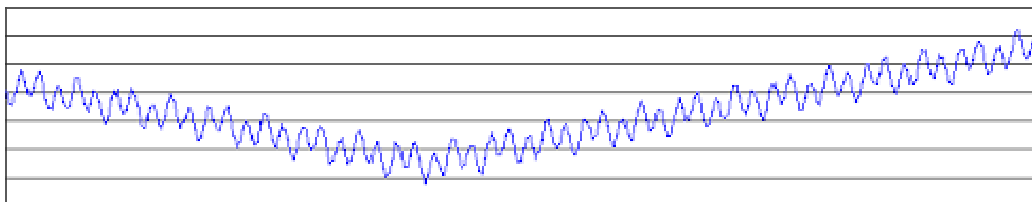

Figure 3.1. Example of a search space

3.3. Representation of Each Individual

GAs work such a way that it selects one in each generation from a population. However, in most cases a random population is generated to start with. In other cases, some input can be feed into the GA to help initializing the starting population.

A chromosome is described as a set of smaller units called genes. The fitness function evaluates the quality of each individual by the value of its genes. After the first population, the members go through evaluation phase in order to selected fittest individuals for further processing. The difference in their fitness value shows how different individuals are from each other.

In this process, we try to maximize the fitness value. The individuals with higher fitness value has a higher chance of surviving. The individuals will participate later in GA operators which are selection, mutation and crossover to produce fitter individual than those produced randomly. [43].

Once the individuals are selected, they go through GA operators in order to ensure a diversity among the generated individuals. The elements of two individuals are coupled through a process called crossover. It is should be noted that, unlike mutation, it is from two different individuals not only one. The main concept of crossover is to make sure that a better offspring can be produced from the coupled parents.

Unlike crossover, mutation happens within particular individuals by swapping its elements. This prevents stagnation near a local optima and ensures diversity in the chromosomes. A strategy should be defined prior to the process in order to identify the individuals that can stay and those who cannot adapt to the change. This process is repeated until the termination condition is satisfied.

3.4. Basic Steps of Genetic Algorithm

Below are the steps of Genetic Algorithm:

1. [Start] Generate random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness f(x) of each chromosome x in the population
3. [New population] Create a new population by repeating following steps until the new population is complete.

(a) [Selection] Select two parent chromosomes from a population according o their fitness value.

(b) [Crossover] with a crossover probability cross over the parents to form new offspring (children). If no crossover was performed, offspring is the exact copy of parents.

(c) [Mutation] with a mutation probability mutate new offspring at each locus (Position in chromosome).

(d) [Accepting] Place new offspring in the new population

4. [Replace] Use new generated population for a further run of the algorithm
5. [Test] if the end condition is satisfied, stop, and return the best solution in
6. Current population
7. [Loop] Go to step 2

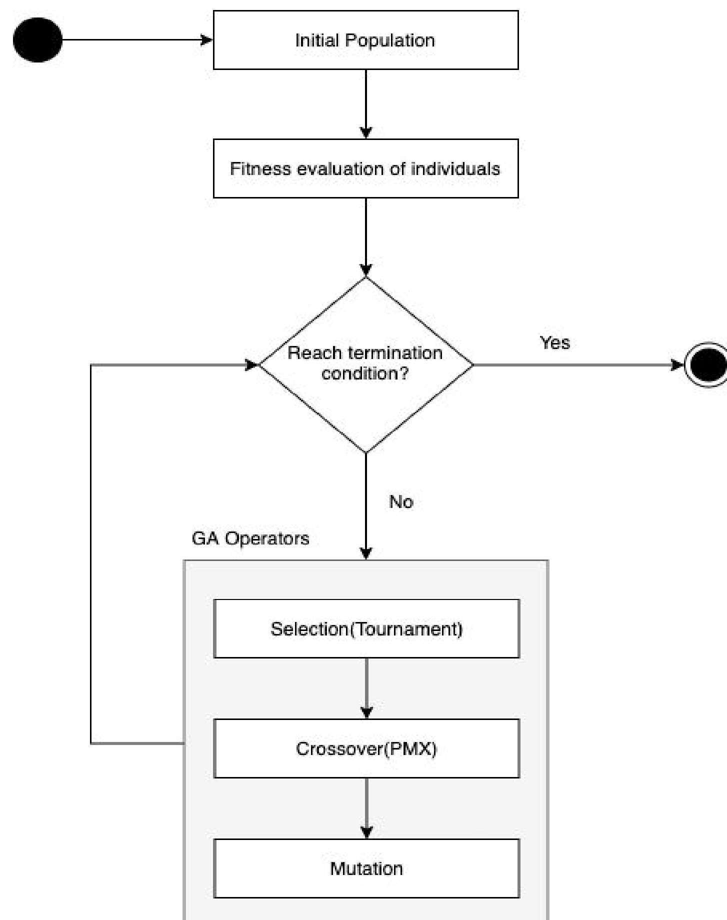

Figure 3.2. General flow chart for genetic algorithm (GA)

3.5. Chromosomes Representation(Encoding)

Chromosome representation can have a huge impact on the performance of a GA-based function. In Genetic Algorithm there are different methods of chromosome representation, e.g. using binary, Gray, integer or floating data types.

3.5.1. Binary encoding

Bit format is the most common type of encoding type. Here the variable values are encoded as bit strings, composed of characters copied from the binary alphabet 0, 1 [40].

The solutions to a problem are represented by chromosomes, where each chromosome consists of a set of variables. Figure 3.3. Describes a problem which consists of a chromosome of three variables A, B and C. Each of these bit fields (A, B and C) represents an input variable value and its smallest unit is one bit that carries the information [34].

Chromosomes					
A	B	C			
0110	0011	1010	A= 6	B= 1	C = 10
0101	0011	1011	A= 5	B= 3	C = 11
0100	1111	1110	A= 4	B= 15	C = 14
0000	0010	0111	A= 0	B= 2	C = 7

Figure 3.3. Binary encoding

3.5.2. Tree encoding

Tree Encoding is one of the types of encoding that uses in programming expressions, i.e. like genetic programming. Here every chromosome is a tree of an object such as, a function or a command in the program. Figure 3.4. Shows an example of Tree encoding [44].

Chromosomes					
A	B	C			
0110	0011	1010	A= 6	B= 1	C = 10
0101	0011	1011	A= 5	B= 3	C = 11
0100	1111	1110	A= 4	B= 15	C = 14
0000	0010	0111	A= 0	B= 2	C = 7

Figure 3.4. Tree encoding

3.6. Selection

Chromosomes are selected from the population for crossover operation. The problem is how to select these chromosomes. There are many methods in selecting the best candidates. Which are Roulette wheel selection, Tournament selection, Steady state selection, Elitism and some others. Some of them are discussed in this section.

3.6.1. Roulette wheel selection

In this selection method, the element (individual/parent) occupies a portion of the wheel based on the fitness value it has. This makes sure that the individual that has better quality will have a higher chance of selection, while the individuals with lower fitness value still have a chance to be selected as shown in Figure 3.5.

It has two major advantages. First, due to the fact that the fittest candidates have higher probability, they more likely survive the selection process and pass to the next stage. The second advantage is that, the individuals with lower fitness value might still pass to the next stage and produce better individuals if they are crossed with other individuals in the population.

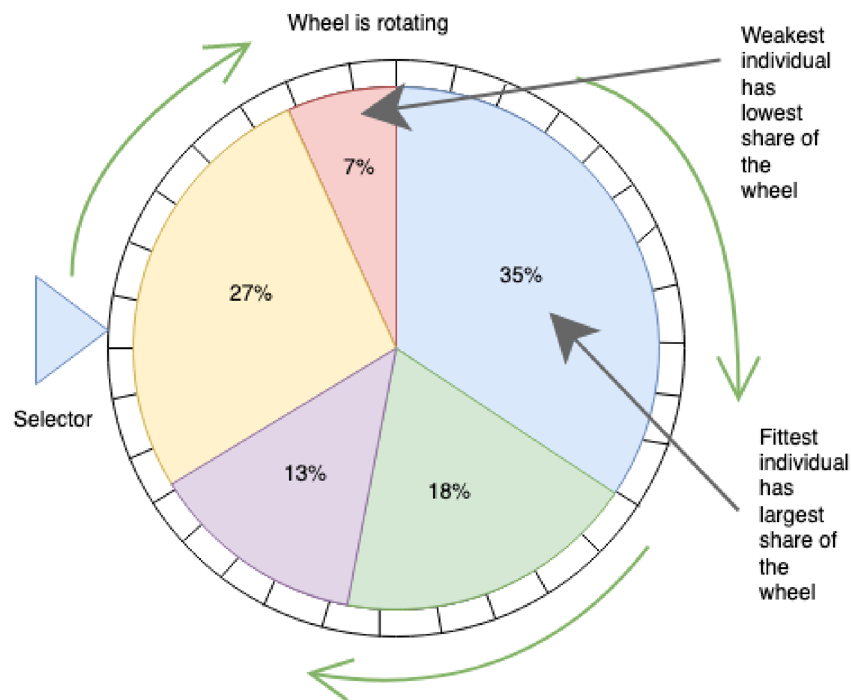

Figure 3.5. Roulette wheel selection method

Data: Population
Result: The selected individual

```
/* Calculate the sum of fitness of all individual */
```

$fitnessTotal \leftarrow \sum_{i=1}^{membersOfPopulation} fitness(i)$

$randomNumber \leftarrow randomNumberBetween(0, fitnessTotal)$

$currentFitness \leftarrow 0$

for $i \leftarrow 1$ **to** $membersOfPopulation$ **do**

  $currentFitness \leftarrow$
   $currentFitness + fitnessValue(membersOfPopulation[i])$

  if *fitness of* $membersOfPopulation[i] <= currentFitness$ **then**

    return $membersOfPopulation[i]$

  end

end

Figure 3.6. Roulette wheel selection algorithm

3.6.2. Tournament selection

It is one of the most widely used selection algorithms. In this algorithm, a random number of individuals are selected from a population in order to compete and pass to

the next generation. Those with highest fitness values are selected and passed to the next tournament. This way, the best parents are selected and hence it gradually narrows down search space.

The idea of the algorithm is simple and can be easily implemented. In addition it can even work with negative fitness values, which is a major advantages when working with real numbers. For example, the fitness function of an algorithm is the multiplication of each genes in the chromosome as shown in Figure 3.7. In this example, the individuals with higher multiplication values are selected and passed to the second round of the tournament.

The first round between two pairs which are 2, 5, 4 and 2, 3, 4. The result of the first multiplication of genes is 40 and the second one is 24. Therefore, it is clear that the first chromosome wins and passes to the next round of the tournament. The same process is repeated for selecting the next parent. At the end, the individuals with stronger traits remain and passes to the next generation.

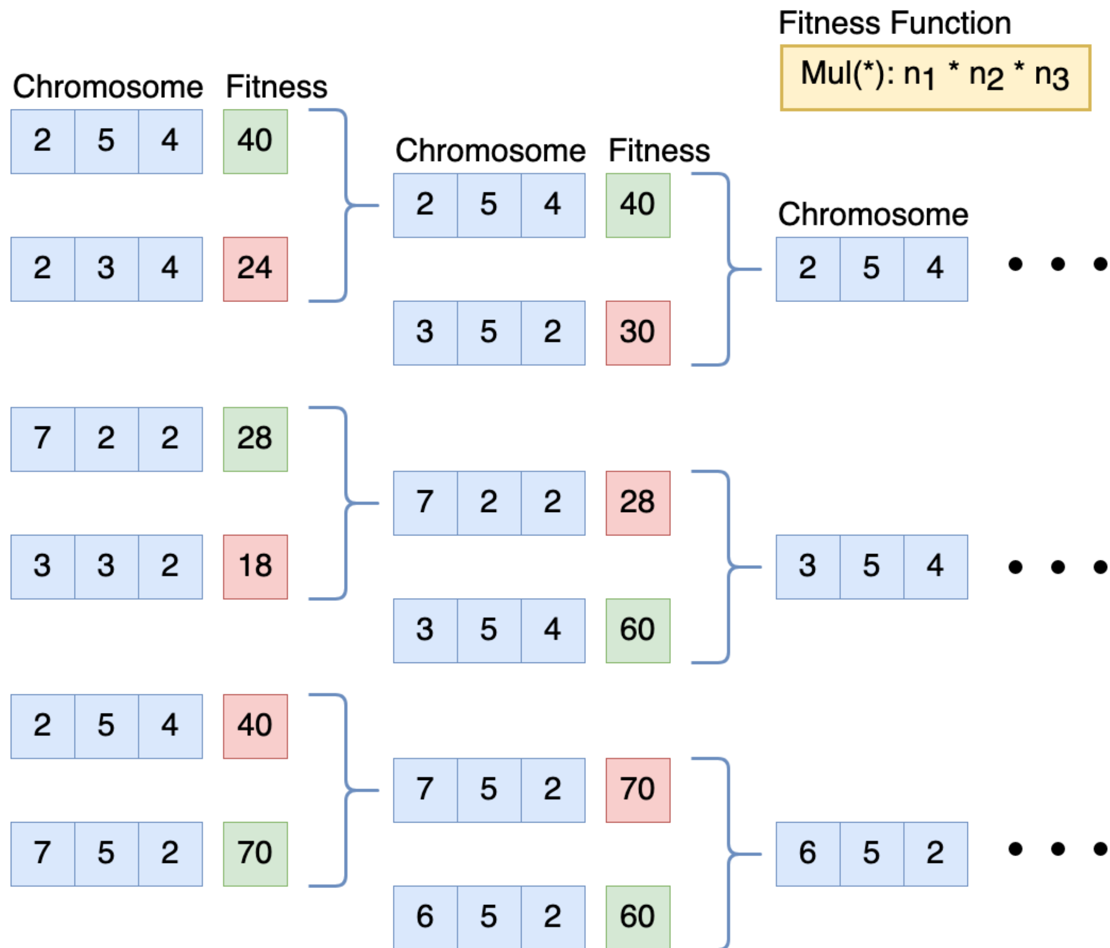

Figure 3.7. Tournament selection method

3.6.3. Elitism

After creating a new population from the crossover and mutation operation, we have a big possibility that we lose the best chromosomes. Elitism is a method that copies few of the best chromosomes to the new created population. The benefit of elitism is to increase the performance of the Genetic Algorithm, because it avoids losing the best chromosomes.

The pseudo code below shows the process of genetic algorithm with elitism.

Data: Chromosomes with random values
Result: Pass best chromosomes to the next generation
for $i \leftarrow 1$ **to** $maxIterations$ **do**
  choose elitist from the current population
  cross those from the remaining chromosomes that enter the crossover rate
  cross those who enter the mutation rate
  has the best fitness value from the newly formed and existing population
   as well as the population
end

Figure 3.8. Elitism selection algorithm

3.6.4. Steady-state selection

In Steady state selection a part of the chromosomes can sustain to the next generation. In every generation the highest chromosomes are chosen to create the new offspring, Then the one with the lowest fitness value are removed and replaced with the new offspring. The process continue and the rest of the population survive to the next generation.

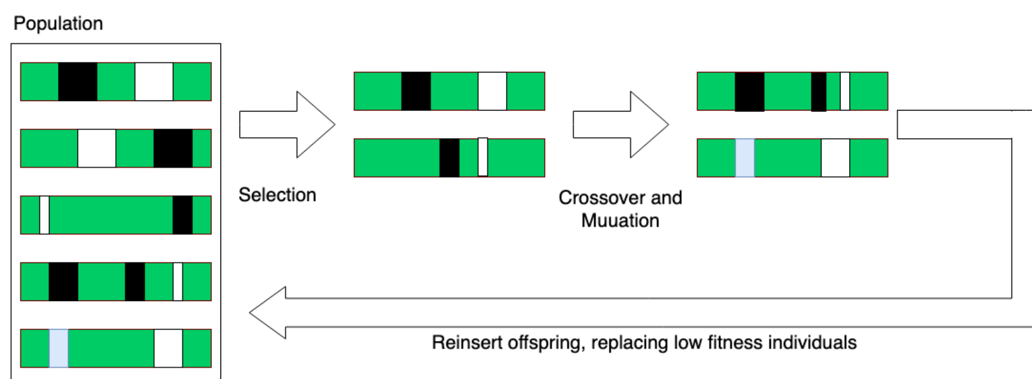

Figure 3.9. Steady state selection method

3.7. Crossover

Crossover is the process of selecting two parents from the population to create one or more offspring’s using their genetic material.

3.7.1. Single point crossover

This is the simplest type of crossover where a crossover point is selected, here the beginning part of the first parent is copied till the crossover point and the remaining part is copied from the second parent.

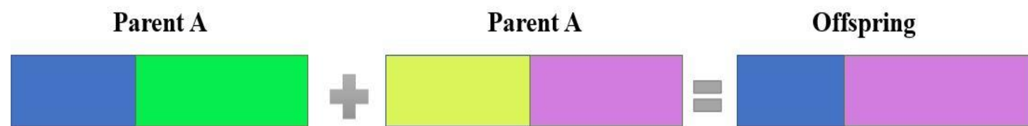

Figure 3.10. Single point crossover

3.7.2. Two point crossover

In two point crossover, two position are selected. Binary string of the first part of the chromosome is copied till the crossover point of the first parent. The center part is copied from the second parent, the rest is chosen from the last part of the first parent.

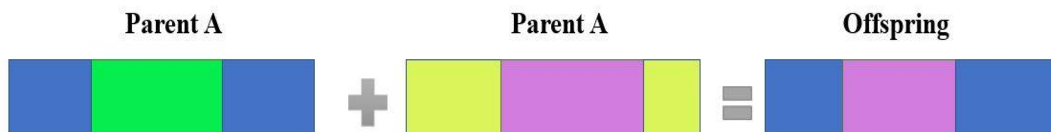

Figure 3.11. Two point crossover

3.7.3. Uniform crossover

In this type of selection randomly bits are copied from the first or from the second parent.

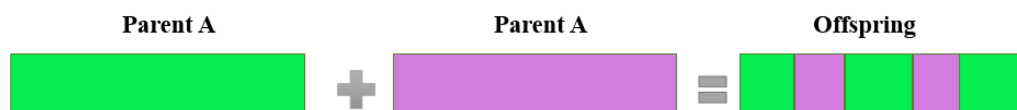

Figure 3.12. Uniform crossover

3.8. Mutation

After a certain iteration, there is a possibility that genes in chromosomes can repeat each other as a result of crossings. To eliminate this condition and provide chromosome diversity, some chromosomes are mutated. For example, in Figure 3.13, only two genes of the chromosome mutate (the value becomes one if it is zero, and zero if it is one).

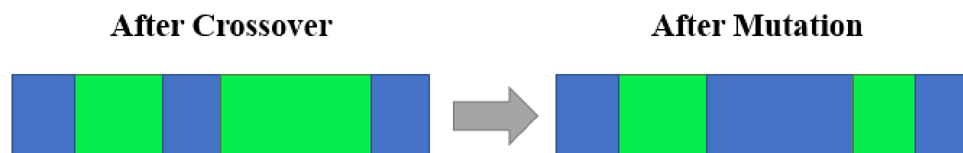

Figure 3.13. Bit inversion mutation

3.9. Population Size and Elitism Rate

There are two main parameters of GA. Which are the population size and Elitism rate that are described below.

3.9.1. Population size

This is about the size of the population that GA needs to explore to find the optimal solution. In Genetic Algorithm it is very important to have a big population, because if there are few chromosomes, the possibility to perform crossover operation is very low.

3.9.2. Elitism rate

From this percentage we can conclude the percentage of best individuals that are survived to the next generation without any modifications, after applying crossover and mutation operators.

CHAPTER 4. METHODOLOGY AND EXPERIMENTS

In this chapter, we discuss the techniques used to implement the tools and different configuration options to run and evaluate our model.

4.1. Experimental Settings

In implementing any tool that works with Evolutionary algorithm & GA, the type of the problem that the algorithm solves has to be carefully analyzed to make sure that it is a suitable algorithm for the specified problem. Before considering GA for any problem, two main factors has been considered in order to make sure it will be applicable for the problem.

The first and most important factor is the process of encoding the solutions to the problem. There are several ways to perform this process. One of the simplest and most well-known encoding method is bit string where each solution is a sequence of binary digits (zeros and ones). Mutation and crossover can be easily applied on this technique. However, it does not imply that this technique can't be used for complicated problem. It depends on the way the solutions are encoded.

The second most important factor is the objective function which is so called fitness function. This functions allows measuring the quality and efficiency of particular solutions. It does not necessarily need to classify individual based on true, false or right or wrong. But it should be able to rank the candidates so that it directs the model to select fittest solution.

The next step in the implementation is to transform the domain in to GA form. In this case, a gene is represented by a method parameter, a chromosome is the combination of all genes in other words all parameters of a method. An individual is a set of

Chromosomes which in this study is a set of combinations for a method that would possibly traverse all paths of the method. For the rest of GA requirements the concepts grows to adapt with the algorithm.

For instance, we have the below method:

```
public void setMemberAge(long age)
      throws UncountableException, CountableException {

   if (age > 120 || age < 1) {
      throw new UncountableException("Invalid age!");
   }

   if (!(age > 12 && age < 20)) {
      throw new CountableException("Is not teenage");
   }
}
```

Figure 4.1. A method to demonstrate transformation to a GA compatible code

This method has two arguments. Both of them are integer typed. In this scenario, age and yearsOfService are two different genes. They both together form the chromosome. The genes with different parameter value combinations for multiple different chromosomes, and hence all together form the individual.

Several parameters has to be taken into consideration that will eventually affect the algorithm performance. These parameters includes but not limited to, number of populations, number of individuals in each population, number of chromosomes in each individual and so on. While all these parameters have impact on the efficiency of the algorithm, but they can be tuned in order to suit the needs of specific application/class. But the major challenge is with the number of chromosome since the outcome and performance of the algorithm is strongly tied to this parameter.

The challenge is, how we should decide about the number of combinations that would possibly traverse most of the paths of the method. For the purpose of this study we

came with a random number of 10. That means each individual is composed of 10 different chromosomes. The ideal way of selecting this number would be traversing each method to get the number of paths and then used as the number of chromosomes for each individual.

One of the major reasons why GA is widely used is because of its simplicity in implementation. It is quite easy to implement as compared to other algorithms used in the area. In this study Java is selected as it is among most popular programming languages and has lots of packages and libraries that can be easily integrated into the project. In addition, it is one of the most advanced languages in terms of applying object-oriented paradigm.

To create this tool Watchmaker used, which is an extensible, high-performance, Object-oriented framework for implementing platform-independent evolutionary algorithms in Java [45].

In this study we focused on public methods only that are accessible both inside and outside the scope of the class. Any instance of that class will have access to public methods and can invoke them. For the data type we considered only primitive type which are (byte, short, int, long, float, double, boolean, char). The reason for choosing only primitive types is because the tool is in the first stage after it developed we can implement other data types like string and objects. The search space contains many individuals where each individual consists of 10 chromosomes. The reason behind choosing 10 chromosome is that selecting this number would be traversing each method to get the number of paths and then used as the number of chromosomes for each individual. Figure 4.2. Shows the flowchart of the proposed technique.

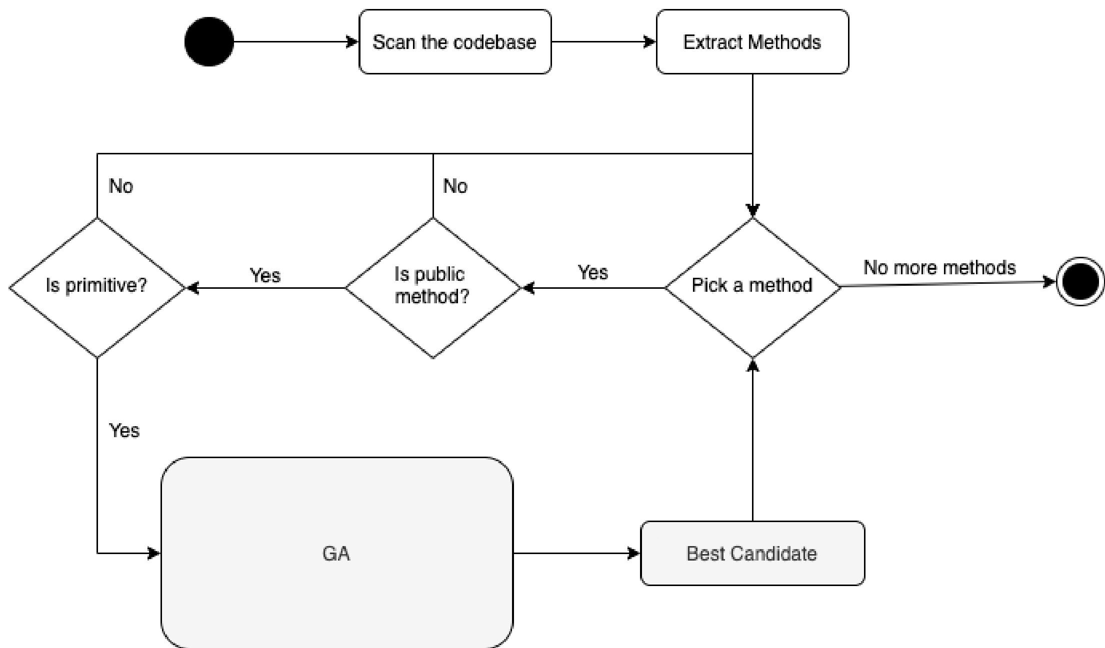

Figure 4.2. The proposed technique flowchart

In the source code before sending any method to the algorithm their conditions checked to know they are public and primitive types or not. Another challenge when dealing with data generation is how to generate a meaningful data that would sound more natural than random data. For example, if we assume that we have a method which determine whether a person is teenage or not. We name this method as isTeenage which takes a single parameter which is the age of the person. Let's assume that one of the generated numbers is 1000, this can actually be sent for test. However in real life a person does not live 1000 years. This is only an example and the random integer generator of java can generate several different numbers that even it is syntactically correct, but it does not sound natural when a software engineer writes tests for this kind of method.

In real life applications this kind of situation is handled by making sure that the data is validated and then sent to the method for processing. But we assume that this scenario stays as is and we want our model to deal with this case. We created two types of exceptions which inherit the main java Exception class. We named them Countable and Uncountable exceptions. This will allow the fitness function to determine whether it is an exception that makes sense to be counted. In other words, is this really a data

which sounds natural when applied to a method for testing? So for the above example for isTeenage(int age) method if the random generator generates any number that is more than 120 years and less than 1 year will not be counted and hence our test data makes more sense when applied to the methods.

If we look at the method above we can see it is a public method which passes a primitive data type (long age). This method is accepting age as a parameter but in case the user entered 1000 this is a logical error where age is impossible to be 1000. For this reason we created the below exceptions to handle this problem.

```
public class CountableException extends BaseException {

    public CountableException(String message) {
        super(message);
    }
    @Override
    boolean isCountable() {
        return true;
    }
}

public class UncountableException extends BaseException {

    public UncountableException(String message) {
        super(message);
    }
    @Override
    boolean isCountable() {
        return false;
    }
}
```

Figure 4.3. An example of countable and uncountable exception method code

4.2. Selection Strategies

Selection is one of the most important GA operations. The performance of the algorithm highly depends on the performance of the selection algorithm. Because the GA further process the individual candidate based on the solutions or candidates went through selection process. Poor selection algorithm leads to poor performance of the

algorithm. Usually, it prefers strong candidates over weak candidates. However, there are sometimes where some features in a weak candidate might make up a strong offspring if selected. For this reason a good selection algorithm usually goes with fitter candidates however sometimes other candidates might be selected to ensure diversity. Sometimes, switching between different selection algorithms leads to much better results of the GA algorithm.

4.3. The Fitness Evaluator

The fitness function is the crucial part of GA. This function is responsible for telling the algorithm how fit the generated data is. We try to maximize the function in order to reach to our target fitness and then decide about how good each candidate/individual is and how it fits in to our problem. Below is the fitness function in our tool:

```
public double getFitness(List<ParamModel> candidate, List<? extends List<Param
Model>> population) {
   int errors = 0;
      for (int i = 0; i < Config.NUMBER_OF_CHROMOSOMES; i++) {
         ParamModel model = candidate.get(i);

         try {
            this.m.invoke(this.obj, model.getParams());
         }
         catch (IllegalAccessException e) {
             e.printStackTrace();
         } catch (IllegalArgumentException e) {
             e.printStackTrace();
         }
         catch (InvocationTargetException e) {
            if(CountableException.class == e.getCause().getClass()) {
                errors = errors + 1;
         }
      }
   }
   return errors;
}
```

Figure 4.4. Get candidate fitness value function code

Before an individual/candidate is represented or shaped by a list of chromosome it should be send for evaluation. Then run it on the real method that we want to test the data for. If an error found, catch it and check to know if it is a countable error or not. If it is a countable error then it adds to the errors list. Our target is to maximize the errors until we reach the target fitness.

It is worth mentioning that a ParamModel represents a chromosome. Each ParamModel object or chromosome is consist of an array of genes which represent a parameter of the method.

4.4. Termination Conditions

Termination conditions are set of conditions which tells the algorithm when to stop. There are a few reasons why the evolution need to stop. First it needs to set a limitation for number of generation, because sometimes the algorithm might not reach to an optimal solution or generation so it keeps running forever. Another case is when the optimal solution reached before actually reaching the limit number of generation. For example, we tell the algorithm to stop after 1000 generations. However the algorithm might reach to the ideal or optimal generation in less than 100 generation. So at that time, what would be the point of running the algorithm until the last generation? Therefore, we allow the algorithm to terminate after reaching the target fitness. Setting the target fitness will save the process a lot of time. There will be a significant impact on the performance of the tool when it is run and tested with a large code base.

Another condition is related to time, how long should we allow the algorithm to run? Sometimes none of the above condition will be satisfied which might be due to the fact that it takes a lot of time to generate the random data due to its complexity. It is not necessary the case with our study, but in some other cases the random generation of data might take a lot of time. Therefore, we do not want the algorithm keep running for a long time and we want to terminate it after a certain period of time.

There are other conditions as well, but we will not go through them as we have not implemented them in our tool. We have set 1000 generation as a threshold for the

generations. In terms of the target fitness, we have set 10 as the target fitness we are willing to get before actually stopping the algorithm. For the most of our cases, we reach the target fitness way before reaching the generation limit. We will talk about the results in details in the results section.

4.5. Model Evaluation

In this study genetic algorithms used to generate test data for a java class Sample-Class.java which contains 6 public methods with different primitive types. The class structure is shown in Table 4.1. and approximation level are the two building blocks used for fitness function. The number of matched branching nodes between the traversed branches and target branch by an individual is called ”partial aim,”, where the local distance value is calculated for the individuals and the approximation level is the number of matched branching nodes between the traversed branches and a target branch by an individual.

Table 4.1. SampleClass structure

Method Name	Return Type	Parameters
determineGuess	boolean	int userAnswer, int computerNumber
setMemberAge	void	long age
isQualifiedForRetirement	boolean	long age, long yearsOfService
withdraw	double	boolean active, double amount
calculateScore	int	char entered, char prompted

```
package ga;
import ga.exception.CountableException;
import ga.exception.UncountableException;
public class SampleClass {

   public boolean determineGuess(int userAnswer, int computerNumber)
         throws UncountableException, CountableException {
      if (userAnswer <= 0 || userAnswer > 100) {
         throw new UncountableException("Invalid number range!");
      }
      if (userAnswer > computerNumber) {
         throw new CountableException("Your guess is too high, try again");
      }
      if (userAnswer < computerNumber) {
         throw new CountableException("Your guess is too low, try again");
      }
      return (userAnswer == computerNumber);
   }
   public void setMemberAge(long age)
         throws UncountableException, CountableException {

      if (age > 120 || age < 1) {
         throw new UncountableException("Invalid age!");
      }
      if (!(age > 12 && age < 20)) {
         throw new CountableException("Is not teenage");
      }
   }
   public boolean isQualifiedForRetirement(long age, long yearsOfService)
         throws UncountableException, CountableException {
      if (age < 1 || age > 120 || yearsOfService < 1 ||
            yearsOfService > 60 || age < yearsOfService) {
         throw new UncountableException("Invalid number range!");
      }
      if (yearsOfService < 15) {
         throw new CountableException("Not enough years of service");
      }
      if (age < 60) {
         String errorMsg = "The person is young, therefore not qualified for retiremen
t";
         throw new CountableException(errorMsg);
      }
      return true;
   }
   public double withdraw(boolean active, double amount)
```

Figure 4.5. Sample class structure code

```
            throws UncountableException, CountableException {
        if (amount < 1) {
            throw new UncountableException("Invalid amount number!");
        }
        if (!active) {
            throw new CountableException("The account is not active");
        }
        if (amount < 1 || amount > 3000) {
            throw new CountableException("");
        }
        return amount;
    }
    // A user wants to improve typing skill
    public int calculateScore(char entered, char prompted)
            throws CountableException {
        int score = 0;
        char smallEnteredChar = Character.toLowerCase(entered);
        char smallPromptedChar = Character.toLowerCase(prompted);
        if (Character.compare(smallEnteredChar, smallPromptedChar) != 0) {
            throw new CountableException("The letters are not the same!");
        }
        boolean enteredCase = Character.isUpperCase(entered);
        boolean promptedCase = Character.isUpperCase(prompted);
        if (enteredCase != promptedCase) {
            String errorMsg = "The entered key and prompted cases key are not the same
";
            throw new CountableException(errorMsg);
        }
        return score;
    }

    public void registerUser(String username, String password)
            throws CountableException {
        if (username.length() < 6) {
            throw new CountableException("Username can not be less than 6 characters!
");
        }
        if (password.length() < 6 || password.length() > 12) {
            throw new CountableException("Password should be between 6-
12 characters!");
        }
        // create user and return username
    }

}
```

Figure 4.6. Sample class structure code (continued)

CHAPTER 5. RESULTS AND DISCUSSION

In this chapter we highlight the results and the output of the tool with detailed explanations of the results.

When working with GA, the results are different from a cycle or iteration when running several times since it depends on the random data generated by the underlying system and platform and even hardware. To minimize the standard deviation between the data generated by the model in each cycle, we ran it 10 times and calculated the average. This way, we can draw conclusion and formalize the results of our model.

5.1. Performance Measures

To measure the performance of the algorithm, 6 metrics have been taken into consideration which are the elapsed time, fitness average, mean fitness, total number of generations, terminated by generation count, and standard deviation.

The elapsed time is used to calculate the total time required to generate the fittest input for the test case. Standard deviation is used to measure the dispersion of the input data with regard to the fitness function.

Number of generations used to determine the required number of generation to get to the best candidate. Finally, the termination by generation count implies that the algorithm has terminated before actually generating an ideal input for the specified method.

5.2. Iterations

To test the performance of the algorithm, we ran it 10 times to find out the performance difference of the model when it is ran multiple times. This can be stabilized in the future by feeding the random generator with seeds that would generate same input when ran several times. All the metrics mentioned in the previous section have been collected as it is shown in Table 5.1.

Table 5.1. Elapsed time for each iteration in seconds

Iteration	Total Time (Second)	Algorithm(Second)	I/O(Second)
1	5	3.31	1.69
2	5	2.5	2.5
3	6	3.57	2.43
4	6	3.57	2.43
5	3	0.82	2.18
6	4	1.53	2.47
7	6	3.28	2.72
8	5	2.46	2.54
9	3	0.99	2.01
10	4	1.68	2.32
Average	4.7	2.37	2.33

The results indicate that it takes (5, 5, 6, 6, 3, 4, 6, 5, 3, 4) seconds respectively including I/O with the average elapsed time of 4.7 seconds. This significantly decreases when we calculate the generation process only instead of I/O which is only 2.37 seconds. The elapsed time is directly proportional to the number of methods, number of parameters and the complexity of each method.

According to the results the fastest time the algorithm was able to generate fittest candidate is 3 seconds and the longest or slowest time is 6 seconds. But most of the time the tool was able to process the methods within 5 seconds. This is the total time from running the application until generating and exporting the results to csv file for the sake of analysis. Generally the average time required to run the algorithm is only 2.21 seconds as we mentioned before.

Regarding the best candidate fitness, as mentioned before the optimal candidate fitness value is 10. In the iteration where the results recorded the average fitness value of the best candidate is 6.56 as shown in Table 5.2. It is worth mentioning that this is the average value for each generation of each round until the best candidate reached. Otherwise the best candidates reach the max value except the only case where the algorithm ended by the generation count. The highest the best candidate fitness is, the closer it is to the optimal input for the method.

Table 5.2. Summary of the data

Attributes	Results
Best candidate fitness average	6.56
Best candidate standard deviation	0.37
Mean fitness	7.52
Total Number Of Generations	1740
Terminated by generation count	1
Elapsed time(Seconds)	1.68

The total number of generation for the entire class is 1740. It varies according to the parameters and complexity of the methods. That is because it depends on the range of the input for each data type. For example a double type surpasses int type by a large margin. Therefore, whenever the algorithm starts generating random values to guess the correct input it has a larger range of possibility. When the number of parameters of methods increase, the complexity increases by N2 as it has to provide a combination of values rather than one single scalar value.

The average sd is 0.3678177 which basically means that how far are we from the optimal solution. It is inversely proportional to the mean fitness. Because in mean fitness we try to get to the fitness value as much as possible while in sd we try to eliminate the difference as much as possible. Hence, the higher mean fitness is, the lower sd is. There is not quite a fixed value for sd as we start running the algorithm, after each generating each individual it has to measure the sd and select an individual with lower sd in the upcoming generations.

5.3. Performance Measures for Each Method of the Class

Table 5.3 shows the performance measure of the algorithm for each method where BA, BSD, MF, TC, ET, TG, Min, Max stands for best candidate fitness average, best candidate standard deviation, mean fitness, terminated by generation count, elapsed time (seconds), total generations, min generation, max generation. It can be clearly seen that only isQualifiedForReqtirement has a round which was terminated by the generation count which is 1000. Generally when a process terminated by generation count means that it has not reach the optimal solution for the certain round.

Table 5.3. Summary of performance for all methods

Method Name	BA	BSD	MF	TC	ET	TG	Min	Max
calculateScore	8.88	0.23	9.91	0	0.08	42	3	8
determineGuess	7.37	0.6	9.33	0	0.11	76	9	11
isQualifiedForRetirement	6.79	0.24	7.37	1	1.13	1304	29	42
setMemberAge	4.6	0.9	6.9	0	0.12	188	19	21
withdraw	5.91	0.84	8.2	0	0.24	130	12	16

We can determine the least generations required to get to optimal solution from the min and max attributes of Table 5.3. calculateScore requires the least generations of 3 while isQualifiedForRequirement requires 29 generations to get to the fittest value. Method parameters play an important role in this scenario. For instance, calculateScore accepts two parameters of char, while isQualifiedForReqtirement requires 2 parameters of long. It is obvious that long has a much larger range of possibilities. Therefore, it requires more generations to make a right guess.

While the ratio of the difference between min and max values are relatively the same among the methods, but we cannot formalize the ratio as it significantly changes when running multiple iteration. This is because it highly depends on the initial value of the input which is 100.The total number of generations are 42, 76, 1304, 188, 130 respectively. While setMemberAge has a single parameter, but it requires a higher number of generation than withdraw although the later has two parameter. The complexity of the method also plays an important role in the number of generations required to guess the right input. There is significant difference in the time required to

generate all the inputs between these two methods. Data generation for the first one requires only 0.116 while for the second one requires 0.242. It clearly shows that while withdraw needs less generation to get to the highest fitness but requires more time due to its complexity.

To further investigate the results and dive more deep into the way the algorithm work, logging the creation of generation, mutation and crossover is integrated into the core module of the proposed tool as it is shown in Table A.1, Table A.3, Table A.4. This is only a small portion of the log and it is only for one method which is calculateScore.

Table 5.4. Contains best candidate for each method in only one of the rounds for the sake of validating the generated input. These candidates does not necessarily contain data that would possibly traverse all paths of the method. This is one of the weakness of our proposed method and can be improved by integrating with tools that traverses all possible paths of a given method.

Table 5.4. Sample input data generated by the algorithm

Method name	Best Candidate
calculateScore	[[W, M], [M, c], [O, e], [k, l], [d, k], [j, m], [t, M], [m, J], [c, V], [m, k]]
determineGuess	[[6, 7], [1, 3], [5, 9], [3, 5], [3, 1], [8, 4], [6, 3], [6, 8], [8, 5], [5, 3]]
isQualifiedForRetirement	[[3, 2], [7, 7], [30, 4], [96, 2], [50, 6], [6, 6], [30, 8], [5, 5], [7, 2], [33, 8]]
setMemberAge	[[5], [9], [94], [7], [7], [2], [66], [6], [50], [1]]
withdraw	[[true, 376091.67], [false, 53263.29], [true, 482417.49], [false, 820788.87], [true, 222086.01], [false, 886813.84], [false, 398886.68], [true, 633687.5], [true, 157370.31], [false, 130025.44]]

CHAPTER 6. CONCLUSION AND FUTURE WORK

Automated testing is one of the major steps in software development life-cycle. It is crucial that the software tested before production in order to deliver a high quality application for the client. Though, it comes at cost of writing and maintaining it.

In this study we propose a model to mitigate the testing process by automating the data generated for the tests. This way, it will reduce a lot of time setting and updating test data for test cases.

In order to achieve this, we use Genetic Algorithms which is known to be very effective in the area where there is a large searching space. As the data used to test is relatively large, we need to use an algorithm that reduces the range and provide reasonable input to the test cases. For this reason, GA is the perfect fit for our concept.

Only public methods that are accessible both inside and outside the scope of the class are used in this study. Regarding data types, only primitive types which are (byte, short, int, long, float, double, Boolean, char) are again in the scope of this study. The reason for choosing only primitive types is because the tool is in the first stage (prototyping), other data types like string and other types objects can be integrated later into the tool.

One of the major challenges in generating test data is generating meaningful data that would probably make sense and what is so called human readable and understandable. Tools that are available right now are source code unaware and generate data randomly. To overcome this issue, to new error class types are introduced which are countable and uncountable. This way, the algorithm tries to depend on the countable errors rather than uncountable and meaningless data.

The fitness function is the crucial part of GA. This function is responsible for telling the algorithm how fit the generated data is. In this type of application, fitness function is maximized to reach to our target fitness and then decide about how good each candidate/individual is and how it fits in the acknowledged problem.

The performance of the tool is quite promising. After testing on a sample class, it was able to generate test data within less than 3 seconds with an average standard deviation of 0.37 and total number of generations of 1740.

We assert that, using our proposed technique will help companies save a lot of time and effort. It is still in early stages, but can be further improved and used as either a library or a standalone tool. In addition, it will help teams in ensuring code quality, because otherwise the tool will end up generating poor data that would sound more like a random data. Many different adaptations, tests, and experiments have been left for the future. Below are some of the future works that can be done on top of our proposed model:

1. Test the tool with different data types and object such as String, custom objects instead of only primitive types (byte, short, int, long, float, double, Boolean, char).

2. Other types of methods and data types can be incorporated in the future to further develop and apply in real life projects.

3. Other selection, mutation, crossover algorithms can be used to enhance the performance of the tool.

4. Integrate other frameworks and tools that is able to generate the number of paths in each method to further maximize the fitness function.

5. Caching techniques can be integrated into the tool to increase its performance.

REFERENCES

[1] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo and N. Juristo, A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? IEEE Transactions on Software Engineering, 43(7), 597-614, 2016.

[2] J. B. Goodenough and S. L. Gerhart, Toward a theory of test data selection, IEEE Transactions on Software Engineering, SE-1(2), 156-73, 1975.

[3] P. McMinn, "Search-based software test data generation: a survey," Software Testing, Verification and Reliability, vol. 14, 2004.

[4] R. P. Pargas, M. J. Harrold and R. R. Peck, "Test-data generation using genetic algorithms," Software Testing, Verification and Reliability, vol. 9, 1999.

[5] M. R. Girgis, "Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm," Journal of Universal Computer Science, vol. 11, 2005.

[6] M. Z. Zhang, Y. Z. Gong, Y. W. Wang and D. H. Jin, "(2019)," Unit Test Data Generation for C Using Rule-Directed Symbolic Execution. Journal of Computer Science and Technology, vol. 34, 1935.

[7] J. H. Andrews, T. Menzies and F. C. H. Li, "Genetic Algorithms for Randomized Unit Testing," IEEE Transactions on Software Engineering, vol. 37, 2010.

[8] C. C. Michael, G. McGraw and M. A. Schatz, "Generating software test data by evolution," IEEE Transactions on Software Engineering, vol. 27, 2001.

[9] W. Visser, C. S. Păsăreanu and R. Pelánek, "Test input generation for Java containers using state matching," in Proceedings of the 2006 international symposium on Software testing and analysis, 2006.

[10] P. P. Mahadik and D. M. Thakore, "Search-Based Junit Test Case Generation of Code Using Object Instances and Genetic Algorithm," International Journal of Software Engineering and Its Applications, vol. 10, 2016.

[11] J. Burnim and K. Sen, Heuristics for Scalable Dynamic Test Generation, 23rd IEEE/ACM International Conference on Automated Software Engineering, 443-446, 2008.

[12] F. Toure, M. Badri and L. Lamontagne, "Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software," Innovations in Systems and Software Engineering, vol. 14, 2018.

[13] Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Kastiskas and K. Karapoulios, "Application of genetic algorithms to software testing," in 5th International Conference on Software Engineering and its Applications, 1992, p. 625–636.

[14] M. Pei, E. D. Goodman, Z. Gao and K. Zhong, 1994 Automated Software Test Data Generation Using Genetic Algorithm, Technical Report GARAGe of Michigan State University, 1994.

[15] M. Roper, I. Maclean, A. Brooks, J. Miller and .. Wood, 1995, 1995.

[16] C. C. Michael, G. E. McGraw, M. A. Schatz and C. C. Walton, "1997," Genetic Algorithms for Dynamic Test Data Generation, Technical report, Reliable Software are Technologies, Sterling, VA. May, vol. 23, 1997.

[17] N. J. Tracey, J. Clark, K. Mander and J. McDermid, 1998, An Automated Framework for Structural Test-Data Generation, In Proceedings 13th IEEE Conference in Automated Software Engineering, Hawaii, 1998.

[18] J. Wegener and M. Grochtmann, "“Verifying timing constraints by means of evolutionary testing”, Real-Time Systems," vol. 3, p. 275–298, 1998.

[19] H. Irman and M. Ahmed, "Genetic algorithm based test data generator," Evolutionary Computation, vol. 1, 2003.

[20] P. Tonella, "Evolutionary testing of classes," ACM SIGSOFT Software Engineering Notes, vol. 29, p. 119–128, 2004.

[21] M.-Z. Zhang, Y.-Z. Gong, Y.-W. Wang and D.-H. Jin, "Unit test data generation for c using rule-directed symbolic execution," Journal of Computer Science and Technology, vol. 34, p. 670–689, 2019.

[22] C. Yong and Z. Yong, Tingting Shi1, Liu Jingyong,, Comparison of Two Fitness Functions for GA-based Path-Oriented Test Data Generation, 2009.

[23] C. Ramamoorthy, S. Ho and W. Chen, "“On the automated generation of program test data”," IEEE Trans. Software Eng., vol. SE-2, no. 4. PD, Vols. SE-2, p. 293–300, 12 1976.

[24] S. Parnami, "Generation of test data and test cases for software testing a genetic algorithm approach," 2013.

[25] Marek Obitko, marek@obitko.com. (n.d.). About - Introduction to Genetic Algorithms - Tutorial with Interactive Java Applets. https://www.obitko.com/tutorials/genetic-algorithms/about.php.

[26] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," International Journal of Embedded Systems and Applications (IJESA), vol. 2, p. 29–50, 2012.

[27] P. McMinn, "Search-based software testing: Past, present and future," in 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011.

[28] R. Malhotra and M. Garg, "An adequacy based test data generation technique using genetic algorithms," Journal of information processing systems, vol. 7, p. 363–384, 2011.

[29] C. B. Lucasius and G. Kateman, "Understanding and using genetic algorithms Part 1," Concepts, properties and context., vol. 19, 1993.

[30] S. Y. Lee, H. J. Choi, Y. J. Jeong, T. H. Kim, H. S. Chae and C. K. Chang, "An improved technique of fitness evaluation for evolutionary testing," in 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops, 2011.

[31] J. Koza, "Agntic Programming, Ann Arbor", 1992.

[32] S. Kanmani and P. Maragathavalli, "―Search-based software test data generation using evolutionary testing techniques‖," International Journal of Software Engineering (IJSE), 2012.

[33] N. Jain and R. Porwal, "Automated test data generation applying heuristic approaches—a survey," in Software Engineering, Springer, 2019, p. 699–708.

[34] R. S. P. Hermadi, "Software Engineering: A Practitioner's Approach", 3rd Edition, McGraw Hill, New York (1992), 1992, p. 559.

[35] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," IEEE Transactions on Software Engineering, vol. 36, p. 226–247, 2009.

[36] G. H., Gross, "An Evaluation of Dynamic, Optimization-based Execution Time Analysis", International Conference on Information Technology: Prospects and Challenges in the 21st Century (ITPC-2003), Kathmandu, Nepal, May 23-26, 2003.

[37] D. E. Goldberg and J. H. Holland, "Genetic Algorithms and Machine Learning," Machine Learning, vol. 3, 1988.

[38] G. Fraser and A. Arcuri, "Whole test suite generation," IEEE Transactions on Software Engineering, vol. 39, p. 276–291, 2012.

[39] C. Emmeche, "Garden in the Machine, The Emerging Science of Artificial Life", 1994, p. 114.

[40] ". D. Education, " That Games Guy, 25, 2019.

[41] Y. Chen and Y. Zhong, "Automatic path-oriented test data generation using a multi-population genetic algorithm," in 2008 Fourth International Conference on Natural Computation, 2008.

[42] Y. Cao, C. Hu and L. Li, "An approach to generate software test data for a specific path automatically with genetic algorithm," in 2009 8th International Conference on Reliability, Maintainability and Safety, 2009.

[43] P. M. S. Bueno, M. Jino and W. E. Wong, "Diversity oriented test data generation using metaheuristic search techniques," Information Sciences, vol. 259, p. 490–509, 2014.

[44] B. Beizer, Software Testing Techniques, Van Nostrand Reinhold, 2nd edition, 1990.

[45] S. Ali, L. C. Briand, H. Hemmati and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," IEEE Transactions on Software Engineering, vol. 36, p. 742–762, 2009.

[46] D. Dyer, "The Watchmaker Framework for Evolutionary Computation (evolutionary/ genetic algorithms for Java)", https://watchmaker.uncommons.org/. [Accessed: 01- Feb - 2021].

APPENDIX

Appendix 1: Contains best candidate per each round for calculateScore method with the number of generations and the elapsed time.

Appendix 1. Best candidate per round for calculateScore method

Round	Generation	Elapsed time(ms)	Best Candidate
11	1	20	[[F, A], [l, r], [T, T], [d, M], [a, M], [D, D], [t, a], [H, N], [L, B], [M, z]]
11	2	24	[[m, m], [M, s], [A, P], [J, J], [m, a], [h, l], [A, d], [b, M], [w, M], [j, L]]
11	3	27	[[S, J], [k, a], [A, S], [c, K], [M, L], [k, k], [b, E], [B, e], [k, j], [A, d]]
11	4	29	[[W, M], [M, c], [O, e], [k, l], [d, k], [j, m], [t, M], [m, J], [c, V], [m, k]]
12	1	3	[[L, R], [J, J], [e, d], [j, J], [K, h], [J, M], [l, L], [Y, f], [d, d], [B, t]]
12	2	4	[[V, S], [M, M], [M, l], [L, c], [n, h], [d, J], [A, A], [K, w], [s, d], [s, v]]
12	3	5	[[M, a], [N, q], [a, a], [p, T], [a, A], [l, T], [L, r], [J, L], [T, l], [n, b]]
12	4	6	[[l, r], [B, M], [C, C], [e, A], [l, M], [C, m], [w, J], [e, s], [C, S], [P, J]]
12	7	8	[[B, l], [B, J], [C, C], [e, A], [l, M], [w, J], [C, m], [e, s], [n, S], [i, H]]
12	8	9	[[k, a], [t, C], [K, p], [K, d], [T, m], [M, B], [a, p], [J, R], [T, i], [D, l]]
7	1	2	[[m, m], [m, r], [c, s], [g, R], [a, r], [C, j], [a, a], [t, m], [f, D], [B, B]]
7	2	3	[[d, B], [S, E], [c, C], [m, J], [t, M], [a, a], [A, A], [j, A], [a, C], [t, l]]
7	3	3	[[T, T], [o, L], [r, Z], [E, d], [w, R], [J, D], [j, I], [D, k], [J, f], [B, O]]
7	4	4	[[n, r], [h, D], [r, z], [K, k], [j, T], [J, t], [C, J], [h, D], [m, J], [r, B]]
8	1	2	[[s, T], [N, t], [L, P], [D, D], [s, T], [N, K], [C, K], [l, I], [e, k], [m, t]]

Appendix 2. Best candidate per round for calculateScoreMethod continued

8	2	2	[[A, G], [L, a], [C, R], [R, v], [d, F], [M, m], [K, n], [C, C], [t, L], [a, A]]
8	3	3	[[W, h], [B, H], [S, y], [A, c], [w, J], [y, J], [b, b], [s, D], [c, A], [R, Z]]
8	4	4	[[c, m], [b, h], [j, m], [G, O], [F, D], [l, B], [M, e], [M, S], [z, T], [A, H]]
9	1	1	[[S, S], [G, u], [a, s], [v, k], [e, B], [c, h], [J, a], [r, C], [A, M], [A, R]]
9	2	2	[[A, M], [H, H], [L, a], [b, n], [s, S], [i, E], [U, I], [p, t], [L, R], [A, e]]
9	3	3	[[A, M], [H, H], [L, a], [G, M], [L, b], [c, E], [E, L], [p, t], [L, R], [A, e]]
9	4	4	[[G, J], [b, g], [d, B], [s, j], [M, s], [n, A], [w, O], [E, D], [B, m], [J, H]]
10	1	2	[[J, k], [A, h], [s, r], [e, e], [A, z], [m, K], [D, D], [r, k], [k, N], [e, C]]
10	2	3	[[m, m], [c, K], [e, M], [b, e], [d, G], [J, c], [R, J], [J, r], [E, A], [A, R]]
10	3	4	[[l, M], [A, K], [b, g], [A, W], [k, g], [L, g], [e, C], [K, K], [l, j], [N, r]]
10	4	4	[[H, j], [s, o], [A, R], [E, M], [S, e], [M, l], [m, E], [b, D], [i, V], [c, B]]

Appendix 3: is a sample of the mutation operator for calculateScore method between the first parent and the second parent. Also contains the first offspring that is created after the mutation process.

Appendix 3: Sample mutation for calculateScore method

	Candidate
First parent	[[A, T], [D, d], [H, n], [c, M], [s, G], [a, r], [l, d], [E, C], [E, K], [k, B]]
Second parent	[[j, C], [c, M], [b, L], [T, h], [E, c], [L, h], [E, N], [A, T], [k, D], [k, P]]
First offspring	[[A, T], [D, d], [H, n], [c, M], [s, G], [a, r], [l, d], [E, C], [E, K], [k, B]]
Second offspring	[[j, C], [c, M], [b, L], [T, h], [E, c], [L, h], [E, N], [A, T], [k, D], [k, P]]
First parent	[[R, C], [L, k], [m, a], [m, s], [j, m], [Z, C], [K, c], [k, D], [O, r], [C, l]]
Second parent	[[m, h], [E, M], [g, n], [t, L], [E, J], [c, G], [a, A], [J, C], [a, i], [L, j]]
First offspring	[[m, h], [E, M], [m, a], [m, s], [j, m], [Z, C], [K, c], [k, D], [O, r], [C, l]]
Second offspring:	[[R, C], [L, k], [g, n], [t, L], [E, J], [c, G], [a, A], [J, C], [a, i], [L, j]]
First parent	[[S, c], [j, N], [g, V], [E, b], [m, j], [a, W], [j, D], [k, M], [w, L], [l, G]]
Second parent	[[s, D], [C, g], [C, T], [j, m], [t, Z], [a, l], [d, j], [Y, E], [T, G], [e, n]]
First offspring	[[s, D], [C, g], [C, T], [j, m], [t, Z], [a, W], [j, D], [k, M], [T, G], [e, n]]
Second offspring	[[S, c], [j, N], [g, V], [E, b], [m, j], [a, l], [d, j], [Y, E], [w, L], [l, G]]
First parent	[[k, w], [k, k], [s, r], [k, V], [a, C], [O, E], [E, D], [m, n], [V, J], [M, p]]
Second parent	[[j, y], [e, D], [N, D], [C, m], [M, T], [d, d], [t, m], [c, j], [A, n], [n, J]]
First offspring	[[j, y], [e, D], [N, D], [C, m], [M, T], [d, d], [t, m], [c, j], [A, n], [M, p]]
Second offspring	[[k, w], [k, k], [s, r], [k, V], [a, C], [O, E], [E, D], [m, n], [V, J], [n, J]]

Appendix 4: is a sample of the crossover operator for calculateScore method between the original candidate and the mutated candidate.

Appendix 4. Sample crossover for calculateScore method

	Candidate
Original	[[e, l], [R, A], [R, e], [R, j], [j, a], [M, N], [s, R], [c, C], [f, E], [c, s]]
Mutated	[[e, l], [R, A], [R, e], [R, j], [j, a], [M, N], [s, R], [c, C], [c, s], [f, E]]
Original	[[L, M], [l, J], [j, A], [N, W], [j, m], [G, I], [A, l], [G, S], [W, p], [S, L]]
Mutated	[[L, M], [l, J], [j, A], [N, W], [j, m], [G, I], [A, l], [W, p], [G, S], [S, L]]
Original	[[N, C], [c, n], [V, M], [G, l], [a, a], [A, d], [K, m], [b, D], [e, G], [d, m]]
Mutated	[[N, C], [c, n], [V, M], [a, a], [G, l], [A, d], [K, m], [b, D], [e, G], [d, m]]
Original	[[T, e], [a, l], [F, A], [h, k], [J, k], [m, R], [l, v], [p, M], [L, v], [N, c]]
Mutated	[[N, c], [a, l], [F, A], [h, k], [J, k], [m, R], [l, v], [p, M], [L, v], [T, e]]
Original	[[s, m], [C, e], [l, M], [o, R], [W, R], [R, d], [j, S], [d, m], [c, G], [j, C]]
Mutated	[[s, m], [C, e], [l, M], [o, R], [W, R], [j, S], [R, d], [d, m], [c, G], [j, C]]
Original	[[M, m], [p, J], [D, r], [k, a], [m, a], [e, M], [D, M], [a, D], [j, f], [M, I]]
Mutated	[[M, m], [D, r], [p, J], [k, a], [m, a], [e, M], [D, M], [a, D], [j, f], [M, I]]
Original	[[A, e], [b, m], [G, I], [m, a], [N, l], [l, m], [a, e], [i, d], [h, M], [B, D]]
Mutated	[[A, e], [b, m], [G, I], [N, l], [m, a], [l, m], [a, e], [i, d], [h, M], [B, D]]
Original	[[s, y], [J, a], [j, s], [k, A], [j, J], [K, c], [H, D], [l, r], [f, c], [j, t]]
Mutated	[[J, a], [s, y], [j, s], [k, A], [j, J], [K, c], [H, D], [l, r], [f, c], [j, t]]
Original	[[j, G], [u, m], [A, g], [k, J], [A, j], [c, A], [K, r], [m, s], [G, b], [M, A]]
Mutated	[[j, G], [u, m], [A, g], [k, J], [A, j], [c, A], [K, r], [G, b], [m, s], [M, A]]

RESUME

Name Surname : Zhela Jalal RASHID

EDUCATION STATUS

Degree	Education Unit	Graduation Year
Masters	Sakarya University / Computer Science and Technology / Computer Engineering	Present
Bachelor	Salahadin University / Software engineering	2011
High School	Zheen School	2007

WORK EXPERIENCE

Year	Place	Position
2013-2016	Shar Hospital	Computer Engineer
2016-2018	Westfall Academy School New York, USA	Coding Instructor
2011-2013	Junior Private School	Computer Teacher

FOREIGN LANGUAGE

English

Arabic

Turkish

Polish- Fair

HOBBIES

Reading, Traveling, Art & Drawing, Learning new languages, teaching, volunteering