An Elitist Evolutionary Algorithm for Automatically Generating Test Data

Jailton Louzada, Celso G. Camilo-Junior, Auri Vincenzi, Cassio Rodrigues
Institute of Informatics
Federal University of Goiás
Goiânia, Brazil
jailtonalkimin@gmail.com, celsocamilo@gmail.com, aurimrv@gmail.com, caleorodrigues@gmail.com

***Abstract*— The development of an effective and efficient method for generating test data is an extremely challenging process which directly impacts the time that could be spent on activities relevant to software testing. Therefore, various researches related to this area have been carried out. Among the techniques for automatically generating test data, we highlight the use of metaheuristics, a promising area called Search-Based Software Testing (SBST). Thus, this article proposes the use of an Elitist Genetic Algorithm (GA) as a tool for generation and selection of test data applied in Mutation Testing for different benchmarks. The results indicate a good performance of the algorithm used in the benchmarks.**

Keywords: Genetic Algorithm; Automatic Test Data Generating, Search-Based Software Testing; Mutation Testing.

I. INTRODUCTION

The importance of software engineering is notable for the current status and future systems development. Since its inception, models, standards, methodologies and processes have been created to increase the development of systems with more quality.

Software Testing particularly discussed in this work as a discipline of software engineering has many activities often known by the name of Quality Assurance and Quality Control, and are introduced throughout the software development process, and is seen as a key element to provide evidence of the level of reliability of software [1]. The main aim of software testing is to detect as many failures as possible, especially the most critical ones, in the system under test (SUT). However the exhaustive testing is impractical due to various restrictions, usually time and cost.

Several studies point that software testing is one of the most expensive disciplines in the process of software development [15] [16] [17]. According to Harrold [2] the discipline of testing can consume up to 50% of the cost of software development. However, the use of automated techniques can reduce the time and effort in software testing and it has been successfully applied, especially in testing data generating. A possible technique which has drawn great interest in the test data generation is the application and tailoring of metaheuristic search (MHS), due to the large use of these techniques in software testing problems, a promising area was created and named as Search-Based Software Testing (SBST) [4].

Among the existing metaheuristics we can highlight Genetic Algorithm (GA) which has shown a great method for solving problems when the search space is large and the evolution of the problem grows exponentially. Genetic Algorithm is widely used as metaheuristics involving SBST, as can be seen in [5], [6], [7], [8] and [9].

Many studies indicate the use of SBST as a technique for solving problems related to software testing, yet there is a need to evolve the research to make more widespread use of this technique in the commercial area. In this paper we use a Genetic Algorithm Elitist as a metaheuristic for the generation and selection of test data applied in Mutation Testing [1].

This paper is organized as follows: In section 2 we performed an explanation of the Software Testing. In section 3 we describe the SBST metaheuristic. Section 4 describes the concept and operation of a Genetic Algorithm (GA). The proposed approach is shown in section 5, followed by results and analysis in section 6. Finally, in section 7 the conclusions and suggestions for future work are listed.

II. SOFTWARE TESTING

Despite the fact of having processes, tools, procedures and complex tasks like other software engineering disciplines, software testing does not guarantee the reliability and quality of the software crafted. In order to minimize the impacts that generate errors in the test activities, various techniques and models were created, for example, the “Model V” of software testing [14] which can be extended to “VV & T” - Verification Validation and Testing.

Several techniques, as well as test models were set up over time with the purpose of exposing an approach more systematic and theoretically substantiated [1]. Some of the techniques created are: Functional testing, structural testing and fault–based software testing. Functional testing, also known as Black-Box testing [14] [16] received this name because the software is like a box in which no one knows its content, in other words, just input parameters are known and analyzes the results are done. Structural testing as opposed to functional test is called White-Box testing [14] [16], as the inside of the box becomes exposed, the tests are performed directly in the source code. The fault–based software testing is a technique that proposes an approach to requirements gathering through testing the most frequent errors generated in the process of software development. Such errors usually created by programmers and designers, among the criteria-based testing errors include: Error Seeding [1] and Mutation Analysis [1]. In this work emphasis is given to mutation analysis.

A. Mutation Analysis

In the history of the art, Mutation analysis (Mutation testing or Program mutation) emerged in the mid-70s at Yale University and Georgia Institute of Technology based on surveys of classical methods for detecting logical errors in digital circuits [19]. One of the main ideas of the mutation testing was proposed around 1978 by DeMillo [18] who proposed a technique called: Competent programmer hypothesis. Based on the principle, that competent programmers have experience in writing source code correctly and/or very close of the correct. These way small changes could be made in the source code with the intention not to change the syntax of the code, but their semantics.

Nevertheless, the mutation analysis uses a set of programs punctually modified (mutants) P' obtained from a given program P to assess how a set of test cases T are suitable for testing T [20], in other words, the goal is determine a set T of test cases that reveal on P by performing the difference between the execution of P and its set of mutants P '.

In mutation testing some mutation operators are pragmatically used to make changes in programs (source code or byte code) based on one of two purposes: 1) induce simple syntax changes in the code (the most common are: logical mutants, arithmetic and relational), and 2) force certain test results (such as running an arc of the program) [8]. In other words, the application of mutation operators under a source code/byte code is called of mutant. If the test suite detects changes in the code, then the mutant is said to be killed. Another definition in mutation analysis is equivalent mutant, when the resulting program is equivalent to the original one. Equivalent mutants detection is one of biggest obstacles for practical usage of mutation analysis. An example of mutation testing using MuJava tool [10] can be seen in Figure 1, where an AOI (arithmetic operator insertion) is used to insert an arithmetic operator in original source code.

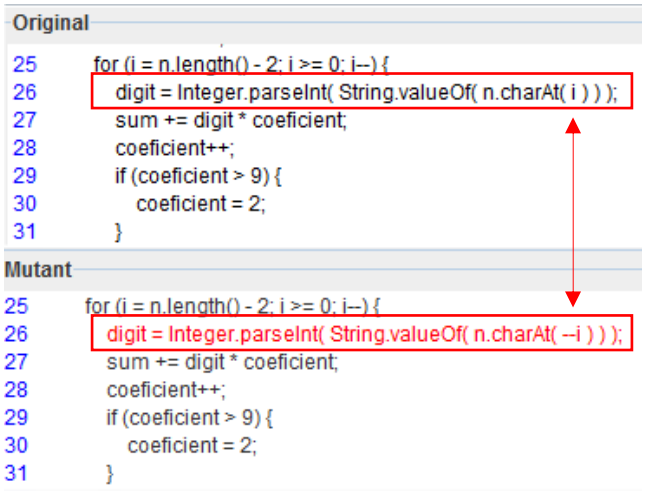

Figure 1 - An example of AOI mutation (Arithmetic Operator Insertion) generated by MuJava tool [10]

Mutation analysis provides an objective measure of the confidence of the test cases analyzed by mutation testing, by a mutation score. According to DeMillo [21] the mutation score is calculated by a relation of killed mutants and mutants generated, and can be expressed by Equation 1:

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)} \quad (1)$$

Where:

P: Original program;
T: Test cases;
DM (P, T): total of dead mutants by the set of T test cases;
M(P): total of mutants generated;
EM(P): total of equivalent mutants to the P program.

The mutation score varies in the range between 0 and 1 and the higher the mutation score will be the most appropriate set of test cases for testing the program. In this work we use the mutation score to assess the best set of test data, in other words, will be the set which kill the most quantity of mutants.

III. SEARCH-BASED SOFTWARE TESTING

The term Search-Based Software Testing (SBST) first appeared around 1976 and was a work of two researchers: Webb Miller and David Spooner [8].

Problems such as prioritization, selection and generation of test cases [5] estimates [22] [23] and selection and generation of test data [6] [7] [11] are general problems that demand time and money due to the high complexity and plethora of possibilities available. Faced with this problem, the use of automated methods are not exhaustive and favors the application of metaheuristics as a tool in the search for optimal

Identify applicable sponsor/s here. *(sponsors)*

or near optimal for carrying out certain tasks of the discipline of software testing. Allowing this way people involved in the process of testing to focus in activities where the ability and intelligence of the software engineer is actually more convenient and less stressful.

Figure 2 displays a histogram of published works using SBST. This graph shows that a lot of works has been published approaching this issue, which further consolidates the use of this technique.

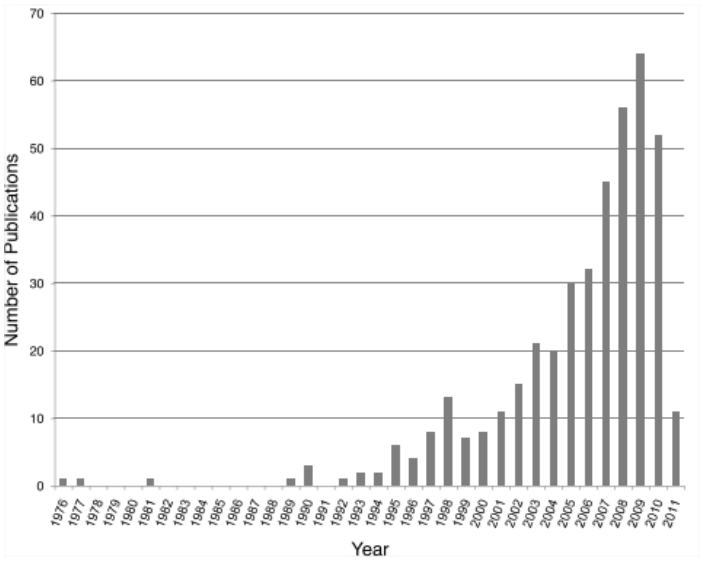

Figure 2 - Publications about SBST, dating back 1976 [4]

IV. GENETIC ALGORITHM

Genetic Algorithm (GA) was first described by Holland [24]. A GA is an algorithm that simulates the evolutionary process of a population of individuals based in the evolution theory proposed by Charles Darwin and that, based on various factors, parameters and genetic operators, aims to evolve individuals (possible solutions) until an optimal, or close to optimal, solution is found. GA has since then been used as a metaheuristic highly important, mainly in the resolution of engineering and software problems, and of course, software testing [5] [6] [7] [12].

The function of a GA can be described as follows: candidate solutions are represented by "chromosomes" that possess a chain of genes that form the solution of the problem. For every individual selected, an evaluation function is applied to determine how the individual is fit as a solution, ie. if the individual will be the most adequate possible solution for the solving of the problem, this process is repeated for several iterations (generations) using genetic operators such as mutations and crossovers.

Metaheuristics such as GA have been frequently used in relation to problems in the discipline of software testing. Michael et al [13] discuss the use of automatic generation of test data using genetic algorithms. Phil McMinn [4] also discusses the use of metaheuristics such as GA making an explanation of the state of art and the application of these metaheuristics in the solving of problems in software testing. In the next chapter the use of a GA applied in the generation and selection of test data for mutation testing will be addressed in a more pragmatic way.

V. PROPOSED METHOD

In this paper we propose the generation and selection of test data evolved by a GA that uses as fitness function the mutation score found by running programs mutants generated from the benchmarks used (see section 5.3). The tool MuJava [10] was used to generate mutant programs that interact with the GA and the core of the program created for experimentation. In Figure 3 it is possible to see the flowchart of operation of the program created for the experiment, which shows the interaction of the tool MuJava and the GA. In the next sections will be set out in more detailed data of the experiment, and the parameters of the GA, the fitness function and rates of mutation and crossover.

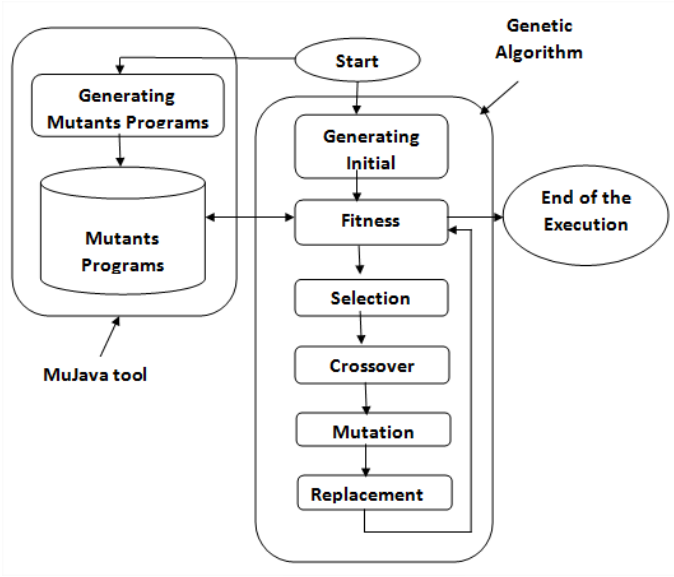

Figure 3 – Flowchart of operation of the experiment

In figure 3 the program is divided into two modules: The module for communication with the tool MuJava that interfaces between the main program and the tool MuJava, and the module that implements the genetic algorithm and performs the generation, evolution and selection of data test that will be used in the tests and comparison of results.

A. GA Parameters

Some parameters are stopping criteria, selection functions, maximum number of generations, maximum population size, crossover rate, mutation rate, individual size, rate of elitism, binary encoding and objective function, which are best described in the sections the following.

The selection method used was the method tournament selection, which draws two randomly selected individuals in the population to participate in a crossover that has the best evaluation. The crossing of individuals is performed among those who had better evaluation. Then after the choice of two individuals (parents) a crossover is performed, the crossover operator used was the One Point Crossover. After selection and crossover of individuals are performed at the end created a mutation in the individual, the mutation is carried out the inversion of bits that uses a probability that can be parameterized in the program to perform the inversion of bits.

B. Fitness Function

The fitness function is the reference of the GA, because it determines the problem to be solved, in other words, it is the goal of optimization in solving the problem. In this work the fitness function is the sum of the number of ratios of types of killed mutants, multiplied by their weights, as seen in equation 2.

$$FF = w1 * (MutA / MutKill) + w2 * (MutL / MutKill) + w3 * (MutR / MutKill) \quad (2)$$

The weights w1, w2 and w3 used in the equation, are respectively the weights assigned to each type of mutation operator used to carry mutations in the benchmarks, which in this case are: arithmetic mutation operator, mutation operator and mutation operator logical relational.

The amount of a mutant killed by type is stored in the variables *MutA*, *MutL* and *MutR*, which stores the amount of mutant arithmetic, logical and relational respectively.

To calculate the fitness function is initially found the proportions of dead mutants according to each of its types of mutation operators (arithmetic, logical and relational) then multiplied by each of their respective weights (w1, w2 and w3) and finally a sum of the rates of mutation is made to find the value of fitness function.

In the canonical model the genetic algorithm does not use elitism, however for this experiment will use an elitist strategy so that only the best individual is chosen for the next generations. In section 6 we will show the results of tests using the GA elitist.

C. Benchmarks

For tests in this experiment some Java benchmarks are used. The goal is to use commercial benchmarks such as to bring the practice of mutation tests and the use of metaheuristics applied to software engineering. Three benchmarks are used: Validator of the number checker Social Security, Validator number of the Voter Registration and Leave PDF Counters. Both benchmarks were chosen because they are widely used by software development community, especially for commercial use.

The input domain of the benchmarks are real numbers, being that benchmarks for the Social Security the input domain of data are represented by numbers of 11 digits. In the Voter Registration benchmark the input domain are also real numbers, but are represented by 13 digits. Finally for the benchmark PDF Leave Counter the domain of input that is also represented for real numbers and allows for a flexible amount of numbers.

The representation of the GA is binary; in this case the data generated through the generations could be used to benchmark a process of converting binary to real be created so that each benchmark for the GA had different parameters. The benchmark Social Security had chromosomes whit 38 genes, the Voter Registration benchmark had chromosome with 40 genes. The amount of genes of the PDF Leave Counter benchmark was of 30 genes.

D. Test Plan

For this work were applied all possible combination using GA parameters, this way we can cover all possibilities. For each number of Generation (300, 500 and 1000) were made 9 (3^2) test cycles applied different combination of parameters like: Mutation rate and Crossover rate.

To guarantee the best result possible for each combination were made 10 executions using the same AG parameters and extracted the following information: Minimum quantity mutant killed, Maximum quantity mutant killed and Average of the top ten mutants that more kill for each 10 executions.

And finally we run all tests with heuristic enabled and disabled, to show the results using an elitist heuristic.

The results and parameters used in each test cycle will be better exposed in section 6.

VI. RESULTS AND ANALYSIS

A. Results with heuristics

To view the results proposed in this experiment, three tables are presented, each respectively shows the total number of killed mutants, the variations in the percentage of crossover and mutation, the number of generations for each experiment and results information with heuristics enabled or not for each of the benchmarks.

Legend:

HEUR = Heuristic (enabled / disabled);
NGEN = Number of Generations;
MUTRAT = Mutation rate;
CROSRAT = Crossover rate;
MINMUTQNT = Minimum Quantity Mutants Killed;
MAXMUTQNT = Maximum Quantity Mutants Killed;
AVERMUTQNT = Average Top Ten Mutants Killed Per Execution.

Nº	Heur	NGen	MutRat	Cros Rat	Min Mut Qnt	Max Mut Qnt	Aver Mut Qnt
1	Enabled	300	0.01	0.65	75.0	78.0	78.0
2		300	0.01	0.75	74.0	78.0	78.0
3		300	0.01	0.85	74.0	78.0	78.0
4		300	0.20	0.65	74.0	78.0	78.0
5		300	0.20	0.75	75.0	78.0	78.0
6		300	0.20	0.85	75.0	78.0	78.0
7		300	0.70	0.65	74.0	78.0	78.0
8		300	0.70	0.75	74.0	78.0	78.0
9		300	0.70	0.85	76.0	78.0	78.0
10		500	0.01	0.65	75.0	78.0	78.0
11		500	0.01	0.75	75.0	78.0	78.0
12		500	0.01	0.85	74.0	78.0	78.0
13		500	0.20	0.65	74.0	78.0	78.0
14		500	0.20	0.75	75.0	78.0	78.0
15		500	0.20	0.85	76.0	78.0	78.0
16		500	0.70	0.65	74.0	78.0	78.0
17		500	0.70	0.75	77.0	78.0	78.0
18		500	0.70	0.85	74.0	78.0	78.0
19		1000	0.01	0.65	75.0	78.0	78.0
20		1000	0.01	0.75	74.0	78.0	78.0
21		1000	0.01	0.85	74.0	78.0	78.0
22		1000	0.20	0.65	74.0	78.0	78.0
23		1000	0.20	0.75	74.0	78.0	78.0
24		1000	0.20	0.85	74.0	78.0	78.0
25		1000	0.70	0.65	75.0	78.0	78.0
26		1000	0.70	0.75	75.0	78.0	78.0
27		1000	0.70	0.85	74.0	78.0	78.0
28	Disabled	300	0.01	0.65	71.0	75.0	74.0
29		300	0.01	0.75	72.0	76.0	75.0
30		300	0.01	0.85	72.0	76.0	76.0
31		300	0.20	0.65	73.0	76.0	76.0
32		300	0.20	0.75	73.0	76.0	76.0
33		300	0.20	0.85	74.0	76.0	76.0
34		300	0.70	0.65	74.0	76.0	76.0
35		300	0.70	0.75	74.0	76.0	76.0
36		300	0.70	0.85	74.0	76.0	76.0
37		500	0.01	0.65	74.0	76.0	76.0
38		500	0.01	0.75	75.0	76.0	76.0
39		500	0.01	0.85	74.0	76.0	76.0
40		500	0.20	0.65	75.0	76.0	76.0
41		500	0.20	0.75	74.0	76.0	76.0
42		500	0.20	0.85	74.0	76.0	76.0
43		500	0.70	0.65	75.0	77.0	77.0
44		500	0.70	0.75	74.0	77.0	77.0
45		500	0.70	0.85	75.0	77.0	77.0
46		1000	0.01	0.65	76.0	77.0	77.0
47		1000	0.01	0.75	76.0	77.0	77.0
48		1000	0.01	0.85	75.0	78.0	78.0
49		1000	0.20	0.65	74.0	78.0	78.0
50		1000	0.20	0.75	74.0	78.0	78.0
51		1000	0.20	0.85	75.0	78.0	78.0
52		1000	0.70	0.65	76.0	78.0	78.0
53		1000	0.70	0.75	74.0	78.0	78.0
54		1000	0.70	0.85	75.0	78.0	78.0

Table 1 - Results of the experiment corresponding to a Benchmark 1

Table 1 presents the data of the Social Security benchmark with 100 mutation classes. As can be seen with increased mutation rate and crossover and the number of generations the number of mutants killed increased, however the changes between the amounts of mutants killed with increased number of generations were not substantial.

The percentage of mutants killed in the best case was 78%, however this value should be deducted from the amount of equivalent mutant to this benchmark for the percentage of equivalent mutants was 22%. In this case the experiment was able to kill completely the mutants.

Nº	Heur	NGen	MutRat	Cros Rat	Min Mut Qnt	Max Mut Qnt	Aver Mut Qnt
1	Enabled	300	0.01	0.65	119.0	126.0	124.95
2		300	0.01	0.75	118.0	127.0	125.94
3		300	0.01	0.85	119.0	127.0	125.99
4		300	0.20	0.65	119.0	129.0	127.89
5		300	0.20	0.75	120.0	129.0	128.78
6		300	0.20	0.85	120.0	130.0	128.62
7		300	0.70	0.65	121.0	130.0	129.89
8		300	0.70	0.75	121.0	131.0	129.87
9		300	0.70	0.85	122.0	132.0	130.80
10		500	0.01	0.65	123.0	132.0	131.64
11		500	0.01	0.75	122.0	132.0	131.32
12		500	0.01	0.85	123.0	134.0	132.46
13		500	0.20	0.65	122.0	134.0	132.22
14		500	0.20	0.75	121.0	136.0	134.85
15		500	0.20	0.85	124.0	137.0	136.40
16		500	0.70	0.65	124.0	138.0	136.93
17		500	0.70	0.75	127.0	139.0	137.25
18		500	0.70	0.85	128.0	140.0	140.0

19		1000	0.01	0.65	128.0	140.0	140.0
20		1000	0.01	0.75	126.0	140.0	140.0
21		1000	0.01	0.85	126.0	140.0	140.0
22		1000	0.20	0.65	129.0	140.0	140.0
23		1000	0.20	0.75	130.0	140.0	140.0
24		1000	0.20	0.85	130.0	140.0	140.0
25		1000	0.70	0.65	131.0	140.0	140.0
26		1000	0.70	0.75	131.0	140.0	140.0
27		1000	0.70	0.85	132.0	140.0	140.0
28		300	0.01	0.65	114.0	116.0	115.09
29		300	0.01	0.75	114.0	116.0	114.98
30		300	0.01	0.85	115.0	117.0	116.79
31		300	0.20	0.65	116.0	117.0	116.75
32		300	0.20	0.75	114.0	118.0	117.98
33		300	0.20	0.85	115.0	116.0	115.57
34		300	0.70	0.65	115.0	117.0	116.89
35		300	0.70	0.75	118.0	119.0	118.85
36		300	0.70	0.85	117.0	120.0	118.99
37		500	0.01	0.65	118.0	123.0	122.75
38		500	0.01	0.75	119.0	125.0	124.95
39		500	0.01	0.85	118.0	127.0	126.95
40		500	0.20	0.65	119.0	127.0	126.89
41		500	0.20	0.75	119.0	128.0	126.95
42		500	0.20	0.85	121.0	128.0	127.35
43		500	0.70	0.65	120.0	129.0	124.50
44		500	0.70	0.75	120.0	127.0	126.54
45		500	0.70	0.85	120.0	130.0	129.30
46		1000	0.01	0.65	119.0	130.0	130.0
47		1000	0.01	0.75	120.0	130.0	130.0
48		1000	0.01	0.85	120.0	130.0	130.0
49		1000	0.20	0.65	117.0	130.0	130.0
50		1000	0.20	0.75	118.0	130.0	130.0
51		1000	0.20	0.85	119.0	130.0	130.0
52		1000	0.70	0.65	124.0	130.0	130.0
53		1000	0.70	0.75	122.0	130.0	130.0
54		1000	0.70	0.85	125.0	130.0	130.0

Disabled

Table 2 - Results of the experiment corresponding to the Benchmark 2

The results of the Voter Registration benchmark in Table 2 show that for the best case the number of killed mutants were 140, in this case as the number of mutants generated for the benchmark was 192 mutants the percentage of mutants killed was 98.60%, however it should be considered the equivalent amount of mutant was 50 mutants with 26% of all mutants.

For this testing cycle a percentage of approximately 1, 4% of mutants were not killed; this is due to the fact that test data have not been optimized to the point of maintaining an efficient manner, probably executed for a greater number of generations would better accuracy of results.

Nº	Heur	NGen	MutRat	Cros Rat	Min Mut Qnt	Max Mut Qnt	Aver Mut Qnt
1		300	0.01	0.65	114.0	118.0	115.0
2		300	0.01	0.75	116.0	128.0	126.0
3		300	0.01	0.85	117.0	128.0	127.95
4		300	0.20	0.65	119.0	127.0	121.5
5		300	0.20	0.75	118.0	130.0	127.90
6		300	0.20	0.85	119.0	130.0	128.98
7		300	0.70	0.65	117.0	130.0	128.80
8		300	0.70	0.75	119.0	130.0	129.75
9		300	0.70	0.85	120.0	130.0	129.85
10		500	0.01	0.65	123.0	130.0	129.39
11		500	0.01	0.75	124.0	131.0	129.0
12		500	0.01	0.85	124.0	130.0	128.10
13		500	0.20	0.65	125.0	132.0	130.85
14		500	0.20	0.75	126.0	133.0	129.0
15		500	0.20	0.85	130.0	134.0	129.70
16		500	0.70	0.65	129.0	135.0	133.99
17		500	0.70	0.75	129.0	136.0	135.0
18		500	0.70	0.85	129.0	137.0	136.80
19		1000	0.01	0.65	130.0	138.0	137.1
20		1000	0.01	0.75	130.0	140.0	140.0
21		1000	0.01	0.85	129.0	140.0	140.0
22		1000	0.20	0.65	130.0	140.0	140.0
23		1000	0.20	0.75	130.0	140.0	140.0
24		1000	0.20	0.85	135.0	140.0	140.0
25		1000	0.70	0.65	136.0	140.0	140.0
26		1000	0.70	0.75	136.0	140.0	140.0
27		1000	0.70	0.85	136.0	140.0	140.0
28		300	0.01	0.65	110.0	115.0	113.5
29		300	0.01	0.75	111.0	120.0	119.5
30		300	0.01	0.85	112.0	120.0	119.0
31		300	0.20	0.65	111.0	122.0	1219
32		300	0.20	0.75	118.0	127.0	126.95
33		300	0.20	0.85	119.0	127.0	126.90
34		300	0.70	0.65	125.0	126.0	125.5
35		300	0.70	0.75	119.0	123.0	122.10
36		300	0.70	0.85	121.0	128.0	127.5
37		500	0.01	0.65	121.0	129.0	128.13

Enabled

Disabled

38		500	0.01	0.75	127.0	129.0	128.98
39		500	0.01	0.85	126.0	129.0	128.5
40		500	0.20	0.65	123.0	129.0	127.95
41		500	0.20	0.75	123.0	129.0	128.5
42		500	0.20	0.85	125.0	133.0	132.80
43		500	0.70	0.65	125.0	133.0	132.87
44		500	0.70	0.75	126.0	134.0	133.60
45		500	0.70	0.85	125.0	135.0	134.96
46		1000	0.01	0.65	125.0	135.0	135.0
47		1000	0.01	0.75	125.0	135.0	135.0
48		1000	0.01	0.85	133.0	135.0	135.0
49		1000	0.20	0.65	128.0	135.0	135.0
50		1000	0.20	0.75	130.0	135.0	135.0
51		1000	0.20	0.85	130.0	135.0	135.0
52		1000	0.70	0.65	129.0	135.0	135.0
53		1000	0.70	0.75	128.0	135.0	135.0
54		1000	0.70	0.85	130.0	136.0	136.0

Table 3 - Results of the experiment corresponding to the Benchmark 3

The third experiment presents the results of PDF Leave Counter benchmark shows the results of tests performed on 150 mutants generated. As defined in the table for the best case with heuristics enabled the experiment was able to get good results (140.0 mutants killed), so all mutants were killed with the test data generated. For this experiment 10 mutants of 150 mutants were equivalent, that is, deducting these values from the total number of killed mutants we obtained a good result.

B. Results using random data without heuristics

	Quantity of test data	Quantity of killed mutants
Benchmark 1	500	72
Benchmark 2	500	111
Benchmark 3	500	121

Table 4 - Results of the experiment using random data without heuristics

The results viewed in table 4 uses a function pseudorandom, that generate exactly 500 test data and apply these values in all benchmark respectively. This experiment shows the maximum quantity of mutant killed for each benchmark.

VII. CONCLUSIONS

The results found in this work were desirable. Considering the importance of automating repetitive and exhausting activities of software testing, as is the case in the generation and selection of test data to mutation testing, was evident that the use of metaheuristic as SBST which was studied in this work add a huge gain in performance of activities whose problems can be modeled as optimization problems.

However some problems were encountered during the experiments, which somewhat complicates the development of testing activities using the technique of mutants analysis, as was the case of equivalent mutants and changes in source code that prevented its execution, because some arithmetic or logical changes that the tool submitted.

Another major factor which must be emphasized is that the use of metaheuristics make more diverse test set, allowing more effective and optimized. As can be observed in the experiments with the use of metaheuristics elitist optimal results were found more than just apply the AG in its canonical form.

As a proposal for future work are presented the following researches: Use of different metaheuristics, development of the tools that enable the practice of mutation testing and the use of the creation and selection of tests data optimized.

REFERENCES

[1] E. F. Barbosa, U. D. São, P. Icmc, M. E. Delamaro, and M. Jino, “Introdução ao Teste de Software,” Dados, pp. 1-49.

[2] M. J. Harrold and A. Drive, “Testing□: A Roadmap,” Atlantic, no. June, pp. 1-10, 2000.

[3] F. G. D. Freitas et al., “Aplicação de Metaheurísticas em Problemas da Engenharia de Software□: Revisão de Literatura,” Reverse Engineering, 2009.

[4] P. Mcminn, “Search-Based Software Testing□: Past , Present and Future,” Transformation, 2004.

[5] J. H. Andrews, T. Menzies, and F. C. H. Li, “Genetic Algorithms for Randomized Unit Testing,” vol. 37, no. 1, pp. 80-94, 2011.

[6] L. R. Santos, C. G. Camilo-junior, A. M. R. Vincenzi, and R. F. Jorge, “An Elitist Genetic Algorithm as Test Data Generator evaluated by the Mutation Test,” Test, pp. 1-11, 2011.

[7] I. Hermadi and M. a. Ahmed, “Genetic algorithm based test data generator,” The 2003 Congress on Evolutionary Computation, 2003. CEC ’03., pp. 85-91, 2003.

[8] W. Miller and D. L. Spooner, “Automatic Generation of Floating-Point Test Data,” Computing, no. 3, pp. 223-226, 1976.

[9] V. Papailiopoulou, “Automatic Test Generation for LUSTRE/SCADE Programs,” 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 517-520, Sep. 2008.

[10] Y.-seung Ma and J. Offutt, “MuJava□: An Automated Class Mutation System,” Interface, pp. 1-34.

[11] B. Korel, “Automated software test data generation,” IEEE Transactions on Software Engineering, vol. 16, no. 8, pp. 870-879, 1990.

[12] S. Ratcliff, D. R. White, and J. A. Clark, “Searching for Invariants using Genetic Programming and Mutation Testing,” Sort.

[13] C. C. Michael, G. McGraw, and M. a. Schatz, “Generating software test data by evolution,” IEEE Transactions on Software Engineering, vol. 27, no. 12, pp. 1085-1110, 2001.

[14] Systematic Software testing - Rick D. Craig and Stefan P. Jaskiel – 2002

[15] R. S. Pressman. Software Engineering - A Practitioner's Approach. McGraw-Hill, 4 edition, 1997.

[16] G. J. Myers. The Art of Software Testing. Wiley, New York, 1979.

[17] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

[18] A. D. Friedman. Logical Design of Digital Systems. Computer Science Press, 1975.

[19] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34–43, April 1978.

[20] R. A Demillo. Software Testing and Evaluation. The Benjamin/Cummings Publishing Company Inc 1987.

[21] R. A. Demillo. Mutation analysis as a tool for software quality assurance. In COMP- SAC80, pages –, Chicago, IL, October 1980.

[22] Laird, L.M., M.C. Brennan, Software Measurement and Estimation, Wiley-Interscience, 2006.

[23] Pfleeger, S.L., F. Wu, e R. Lewis, Software Cost Estimation and Sizing Methods: Issues, and Guidelines, Rand Corporation, Location, 2005.

[24] J.H. Holland, Adaptation in Natural and Artificial Systems. Univ. of Michigan Press, 1975.