Heuristics Applied to Mutation Testing in an Impure Functional Programming Language

Juan Gutiérrez-Cárdenas[1], Hernan Quintana-Cruz[2], Diego Mego-Fernandez[3], Serguei Diaz-Baskakov[4]
Carrera de Ingeniería de Sistemas
Universidad de Lima

Abstract—**The task of elaborating accurate test suites for program testing can be an extensive computational work. Mutation testing is not immune to the problem of being a computational and time-consuming task so that it has found relief in the use of heuristic techniques. The use of Genetic Algorithms in mutation testing has proved to be useful for probing test suites, but it has mainly been enclosed only in the field of imperative programming paradigms. Therefore, we decided to test the feasibility of using Genetic Algorithms for performing mutation testing in functional programming environments. We tested our proposal by making a graph representations of four different functional programs and applied a Genetic Algorithm to generate a population of mutant programs. We found that it is possible to obtain a set of mutants that could find flaws in test suites in functional programming languages. Additionally, we encountered that when a source code increases its number of instructions it was simpler for a genetic algorithm to find a mutant that can avoid all of the test cases.**

***Keywords*—Mutation testing; heuristics; functional programming**

I. Introduction

Usually, when a programmer develops a software, it requires that when this code is executed; the obtained answers are correct given the right set of inputs. However, this required performance is sometimes not achieved, due to incorrect answers given when the execution of the program is completed. The main reason for this so-called abnormal behavior of an application is due to the presence of flaws within the code. In a way to perform an extensive search for these flaws is to elaborate a group of tests cases; in which a program should return the correct answers established by them. The elaboration of these tests cases is not a trivial task because it requires a certain level of detail to try to catch a flaw in the program tested. Because of this, it is needed a form to evaluate the quality of these test cases. An important part to consider when developing a test case is that this should detect if a program behaves abnormally when specific inputs are entered. A naive way to perform the quality of a test case would be to cause intentional flaws in a program and to observe if the answer that this faulty program returns is the same as the original program. We usually called this flawed program a mutant. In the case that a mutant and a original program return different outputs when dealing with the same input that is in a test case, we say that test case has killed the mutant. In the opposite case we can say that the mutant has survived. This mutant generation can be performed by using techniques based on heuristics, such as using Genetic Algorithms. We found two things related to the use of heuristics in mutation testing: a) The different methods for mutation generation using Genetic Algorithms are efficient in terms of searching in an ample solution space and, b) Even though, Heuristics has been used for mutation testing in imperative languages, their use for testing functional programs was scarce. This last characteristic could be due that programs made by using functional paradigms are less prone to flaws, this because of the formal or mathematical aspects of their constructs. We came up with the research question if that strength of the functional paradigm could also be present in impure versions of functional programming languages such as in Scheme. It is worthy of mentioning that an impure functional language is the one that has some imperative within it. With this research question in mind, we wanted to test how well perform mutation testing in functional paradigms. This article is divided in the following sections: In Section II, we made a brief introduction to test, mutation testing and Genetic Algorithms. In Section III, we developed our genetic operators and fitness function; along with a graphical conversion of the code that will be exposed to these genetic operators. In Section IV, we show our results obtained by testing our technique in the implementation of different functional programs, ending with the conclusions of this research work.

II. Background and Related Work

A. Software Testing

Now-a-days, software development is an activity that is present in many aspects of the daily routines and activities in which the human being is immersed. Some of these tasks could be reasonably trivial such as processing pictures or processing documents, while others could perform critic tasks as controlling an airplane. This is the main reason why a simple error in the software could affect a considerable number of users and stakeholders. Nevertheless, the software should not contain errors or behave in unexpected ways deviated from their regular tasks. There exists a certain number of factors that could affect the software behavior brought up from their implementation part. The field of software design tries to discover these factors in a structured and detailed process that is oriented to the development of tests. These test could allow us to prove the validity of the implemented software [1].

One way of testing if a program behaves correctly is to prove it by using a technique known as mutation testing. A mutant is program altered that, when exposes to a set of different inputs, we can observe if, by behaving in a different or anomalous way, a set of test cases could detect this defective program. A simple way to understand this technique is shown in Fig. 1.

We can observe in Fig. 1 that we start with the source code of a program that we would like to test. Considering

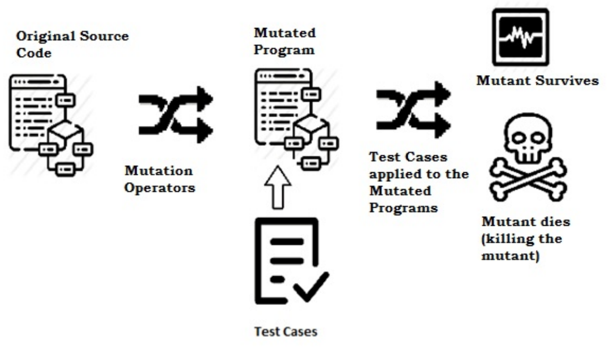

Fig. 1. Relationship between the Mutated programs and the Test Cases

this initial code one can generate modified versions of the same program called mutants. Therefore, a mutant suffers inner modifications of their original code, and after these changes, they are exposed to a set of test cases. The test cases will serve to prove if they are capable of finding errors in this mutated program. This verification process follows the criteria that a test case should be able of detecting anomalous behaviors of a code. If the mutated program, when is exposed to a test case, returns an output different to the one that the original program returns; this would mean that the test case was able to detect this error. Therefore, we can say that the mutant is killed. The problem arises when both, the original and the mutated program, returns the same answer that was given by the test case. In the situation mentioned above, we can say that the mutant survives. The main reason for this behavior is because the test case was unable to detect an unusual mode, and therefore the test cases should be improved. An important issue to mention is that, in the generation of mutants, we should consider generating fully-functioning mutants instead of non-executable ones. An interesting point of view is expressed in the book of Paul Ammann and Jeff Offutt [1], where the authors mention that in the test cases, irrelevant if their manual or automatically generated, they are tested to verify the output of a program. In consequence, if there is an erroneous output, the program should be corrected, and the mutation process is performed in the same fashion as we described before. At this point, we believe that it will be valuable to mention a set of formal definitions that should hold in the process of killing mutants:

Definition 1: Given a mutant $m \in M$ such that *m* is a valid mutant and *m* is derived from the ground string *p*. Then if the $answer(m) \neq answer(p)$ given a test *t*, *t* kills *m* if the first condition holds.

Note: According to Paul Ammann and Jeff Offutt [1], a ground string is defined as a source code without any alteration.

The authors also mention that a mutant should uphold the following conditions:

a) Reachability: This feature refers that when we test a mutant, the test case should be able to reach until the point where we mutated an instruction.
b) Infection: An infection occurs when a reached mutated instruction will generate a change in the state of a program. The state of a program, in this scenario, is similar to the concept given in the different paradigms in which a programming language could be enclosed.
c) Propagation: Once the mutant is executed, the program could output an erroneous output.

The process of killing mutants is the one that will allow us to determine the effectivity of a test case, for this purpose there exist a metric named the mutation score and it can be defined as follows:

$$mutation\ score = \frac{|deleted\ mutants|}{|non - equivalent\ mutants|} \quad (1)$$

According to equation (1), a perfect metric will hold a value of one, but this theoretical value can be complicated to obtain; so it would be better to work with threshold values.

There exists, also, a special kind of mutants called of equivalent type. An equivalent mutant is the one that returns the same answer as an unmodified program when they are exposed to a set of test cases. In summary, it does not exist a set of test cases that can find differences between the mutated program and the original program [2]. Some authors, such as Budd and Angluin [3], mentions that this is an irresoluble problem; therefore, it does not exist an efficient or at least known-way that could detect a set of mutants that are equivalents. Other authors like Offutt and Pan [4] also mention the problem of the creation of equivalent mutants. These authors followed a restriction-based system considering a variant of the Feasible Path Problem (FTP). The three conditions that a mutant should uphold mentioned by Paul Ammann and Jeff Offutt [1] are also defined by other researchers such as Botacci [5], but with subtle modifications. For example, this author considers that the features that should be present are:

a) Reachability.
b) Necessity, according to these criteria if we compare the original program P with a mutated line on a mutated program M, there should exist a difference in the state of execution of a program.
c) Sufficiency, this feature determines that the output of P and M should be distinct.

Mutation testing is a vast and evolving area of study, that has changed from simple code modifications to complex ones to examine the strength of our test suites. The reader interested in a complete survey about this subject is encouraged to review the research work of Papadakis, Kintis, Zhang, Jia, and Le Traon [11] In the research work of Le, Amin, Gopinath and Groce [10] is mentioned the fact that mutation testing has been primarily applied to imperative programming. Nevertheless due to the relevance that is having this paradigm in some fields such as Big Data, Data Science and with the inclusion of functional paradigm constructs in imperative languages, the interest of performing mutation testing in functional languages was envisioned as an exciting research field. The authors mentioned previously developed a software tool denominated MuCheck that performed mutation testing in a functional paradigm by using Haskell as the chosen language. It is worthy of mentioning that Haskell belongs to the family of pure functional languages, while in our research, we have considered

Scheme, which is categorized as an impure one. An impure functional language is the one that despite belonging to the functional paradigm; it has some characteristics that are present in other paradigms, such as imperative constructs. The reason for choosing this impure functional language was because Scheme, represented actually by the Racket programming language, has applications in different real-world applications. Furthermore, Racket or Scheme is considered a programming language that it could raise the development of the Language-Oriented Programming (LOP). A LOP language is the one that would allow developers to use a programming language that could help them to design programming languages oriented to suit their needs, so eventually, it fulfills the need of using different programming languages constructs in one host programming language by allowing the programmer to develop their personalized programming language [12]. The reader interested in this particular characteristic can review the following link: https://beautifulracket.com/

B. Genetic Algorithms

The Genetic Algorithms (GAs) are a technique of searching and optimization based on heuristics. These are clear contraposition of other methods based on calculus or in dynamic programming. One of the main reasons for their usage is that they perform appropriately in higher dimensions where the problem known as the curse of dimensionality, could be present. Previous techniques based on non-guided searches or randomly generated have existed before the appearance of the GAs, but with the limitation that they could get stuck in a local optimum [6].

Components of a GA and Genetic Operators

In this section, we will describe some components and operators that are commonly used on the GAs. We should consider that the literature about this topic is diverse, but we will focus our work in the research made by Goldberg [6], Mitchell [7] and Sivanandam [8]. Among their principal components are the chromosome, part that allows forming a population; and a fitness function that will select the best individuals by following a set of Genetic Operators. There exist two types of operators that are work jointly with the population and the fitness functions. The operator of Selection allows us to chose a couple of parents for generating a new individual [8]. The techniques for the selection of individuals are diverse, but most of them are centered into probabilistic or stochastic methods. According to Sivanandam [8], there exists what is called a pressure selection, which is a trick that allows one individual to be chosen over another one. Another type of operator is called Reproduction, and according to Holland [9] it is a function that allows forming a new population, considering the selection criteria performed by the fitness function. For reproducing a couple of individuals, they use two functions more. The first one selects randomly two parts of each chromosome to be exchanged with one another in a process called Crossover. After this process and before a new couple of chromosomes are about to be put into the pool of individuals, an operator of mutation can be present. The operation of mutation what it does is to select one portion of each chromosome and randomly, with a very small probability, changes its value one value for another. The importance of this operator is that it will allow, in some cases, to push the chromosomes to be replaced for subtle modifications of them and to permit a diversification into the search space.

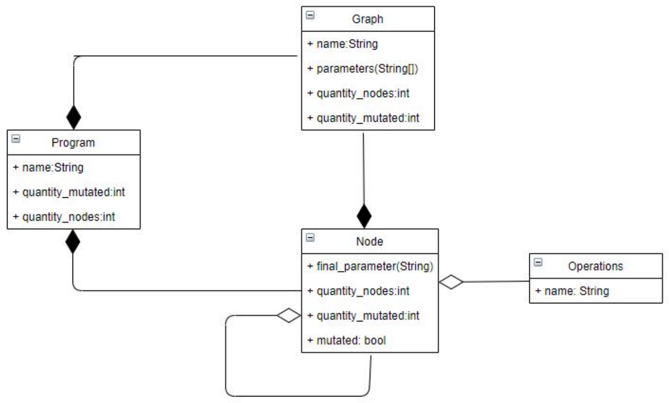

Fig. 2. Relationship between the Mutated programs and the Test Cases

III. Methodology

A. Class Diagram

We have developed a set of classes with the sole goal of parsing our source code. It also helps to define other constituents such as functions and parameters, and operations that would aid in converting our source code into a graph representation. The graph representation would serve then for the mutant generation, which at the end will form part of the population for our genetic Algorithm. Our schemata of classes present the following components:

Program: It represents the code analyzed or its mutation. It contains data of type string which stores the name, a list of graphs that represents the defined functions and a list of nodes which corresponds to the called functions.

Graph: Represents the definition of a un function. It contains data of type string for storing its name, a list also of type string that contains its parameters and a node that points to the actions to be performed.

Node: It can represent a parameter. In the way we program our classes a node can also be a function that is why this class has a list of nodes that represents the parameters of a function. It contains an attribute of type operation and a string data type that stores the name of the parameter in the case that it does not correspond to a function.

Operation: Represents the name of the function that is being executed.

In Fig. 2, we can observe the class diagram of our solution.

B. Algorithms

We have defined the following algorithms for our solution:

```
Algorithm: Parser code for function
search.

if character!= EOF
```

```
  while character = \" or character = \\n"
  or character = \)"
      if scan(character) = EOF
          return
      if character = \(\
        scan(character)
        word <- obtain_word(character)
        if word = \define"
          graph <- parse_function(character)
          program.funciones.push(graph)
          parse(character, program)
        else
          num <- 0
          character <- \(\
          node <- parse_code(character, num)
          node.operation.name <- word
          program.codigo.push(node)
          parse(character, program)
      else if character = \ ’ "
        num <- 0
        node <- parse_code(character, num)
        node.operation.name <- \ ‘ \
        program.codigo.push(node)
        parse(character, program)
```

```
Algorithm: Code parser. Adds the parameters
or arguments of a function

num++
while character = \ \ or character = \\n"
    scan(character)
  if character = \(\
    scan(character)
    if character = \(\
      node.operation.name <- \ \
    else
      node.operation.name <-
       obtain_word(character)
    while character ?= \)"
      while character = \ \
      or character = \\n"
        scan(character)
      node.parameters.
      push(parse_code(character, num))
    scan(character)
  else if character = \ ‘ \
    node.operation.name <- \ ‘ \
    scan(character)
    while character = \ \ or character= \\n"
      scan(character)
    if character = \(\
      scan(character)
      while character ?= \)"
        while character = \ \ or
        character = \\n"
          scan(character)
        node.parameters.push
        (parse_code(character,num))
      scan(character)
    else
      node.parameters_final <-
       obtain_word(character)
  else
    node.parameters_final <-
     obtain_word(character)
return node
```

```
Algorithm: Function parser. Creates a graph
and inputs the information of the declared
function

while character = \ \ or character = \\n"
  scan(character)
  if character = \(\
    scan(character)
    graph.name <- obtain_word(character)
    while ( character!= \)"
      string <- obtain_word(character)
      graph.parameters.push(string)
    scan(character)
  else
    graph.name <- obtain_word(character)
num <- 0
graph.code <- parse_code(character, num)
graph.nodes_quantity <- num
return graph
```

For the part of the Genetic operators that we used, we defined the following algorithms:

```
Algorithm: Obtain word. Returns a string
with the posterior word to the character
that was input as a parameter.

list l
if operation = \list" || operation=\vector"
|| operation = \ ’ "
  if operation = \list" || operation=\ ‘ \
    l.push_back(\vector")
  else
    l.push_back(\list")
else if operation = \car" || operation=\cdr"
  l.push_back(\car")
  l.push_back(\cdr")
  l.remove(operation)
else if operation = \+" || operation = \-" ||
operation = \*" ||
        operation = \/"
  l.push_back(\+")
  l.push_back(\-")
  l.push_back(\*")
  l.push_back(\/")
  l.remove(operation)
else if operation = \<" || operation = \>"
|| operation = \<=" ||
        operation = \>="
  l.push_back(\<")
  l.push_back(\>")
  l.push_back(\<=")
  l.push_back(\>=")
  l.remove(operation)
else if operation = \and" || operation = \or"
  l.push_back(\and")
```

```
    l.push_back(\or")
  l.remove(operation)
else
  return 0
m <- random(l.size)
operation <- l[m]
mutated <- 1
return l
```

```
Algorithm: Program crossover. Extracts parts
of a program and exchanges them.

m <- random(program1.functions.size)
function1 <- program1.functions[m]
function2 <- program2.functions[m]
destination1 <-random(function1.nodes.size)
destination2 <-random(function2.nodes.size)
intercambiar(destination1, destination2)
```

C. Genetic Operators

Because we worked with the functional paradigm, we chose the following operators or instructions to be exchanged. This process is performed within the mutation step of a program.

1) Fitness Function: The Fitness or Adaptation function is the one in charge of selecting the best individuals among a population of entities in a given generation of a GA. We have defined the following items and equations for this section:

$T = quantity\,of\,tests$

$e_n = \,correct\,answer\,for\,the\,n\,test\,(value\,of\,1\,or\,0)$

$t_n = quantity\,of\,mutants\,not\,detected\,by\,test\,n$

$m = quantity\,of\,mutants$

$c_n = it\,counts\,the\,results\,for\,the\,n\,test, irrelevant\,if\,the\,answer\,is\,correct\,or\,not$

$N_a = quantity\,of\,nodes\,in\,the\,actual\,program$

$N_i = quantity\,of\,nodes\,in\,the\,original\,program$

These parts have served to formulate the following equations, that we have preferred to call them criteria:

Criteria 1:

$\sum_{n=1}^{T}\left[e_n\left(1-\frac{7t_n}{10m}\right)\right]*10$

This equation counts the number of tests that left the mutant undetected. Approximately, 70 percent of the score is given to the test that detected a certain amount of mutants. The highest score is given to the test that picked most of the remaining mutants. The value of 7/10 is for giving more weight to those mutants that evade the tests.

Criteria 2:

$\sum_{n=1}^{T}(c_n)*\frac{1}{T}$

It counts the number of answers given by a mutant, irrelevant if these are correct or not.

Criteria 3:

$\left(1-\frac{|N_a-N_i|}{3N_i}\right)$

It measures how much has varied the number of nodes of the mutant with regard to the original program. Less variation means a higher score. This criterion is used for neglect programs with a mutation threshold higher than what could be considered an useful mutation.

When we combine the criteria above mentioned we would end up with the following fitness function:

$\sum_{n=1}^{T}\left[e_n\left(1-\frac{7t_n}{10m}\right)\right] * 10 + \sum_{n=1}^{T}(c_n) * \frac{1}{T} + \left(1-\frac{|N_a-N_i|}{3N_i}\right)$

Once that we have defined our algorithms and operators to be used in our Genetic Algorithm we proceed to the implementation of our proposal. We have tested our program with three different types of programs, made under the functional paradigm and using Scheme as a programming tool. The programs tested were a) a sorting function using quicksort, b) a function that solves a quadratic equation, and c) two functions that implement the Prim and Kruskal algorithms for obtaining minimum weighted spanning trees. We will describe the results of our experimentation in the following section.

IV. Results

At the moment of executing our program with the target source codes; it starts to search for an opening bracket in the source code. If this is found the next instruction is analyzed and if it corresponds to the term define; a graph that represents this function is created along with its corresponding parameters. This procedure is done in two steps: a) we obtain the name or the parameters of a function, and b) we collect the statements or calling to other programs if this is the case. With these, we create a node that stores all the information of a function and is ready to be mutated. It is worthy of mentioning that with the graph obtained from a program, we generate several copies and each of these copies is mutated according to the operators defined in Section 3.3. Each mutated program or mutant is executed with all the tests and, later, a score is assigned by using our fitness function. When we end up with this procedure, we keep 20 percent of the mutants with a higher score, crossing them form generating new individuals. The process mentioned above is repeated until we reached the number of programs that we defined beforehand as the number of individuals in each generation. For all the three programs we have chosen the following parameters:

- Number of mutants or individuals by generation: 100
- Number of tests: 3
- Generations: 80

A. Quicksort

The original code is the following:

```
Quicksort: Original source code

#lang racket

(define (partition compare l1)
      (cond
         ((null? l1) '())
         ((compare (car l1)) (cons (car l1)
```

TABLE I. NUMBER OF TESTS EVADED BY THE QUICKSORT MUTANT

	Generation	Altered Nodes	Evaded Tests	Score
Mutant 1	1	33	2	21.02
Mutant 2	3	33	2	20.51
Mutant 3	2	33	2	20.25

```
        (partition compare (cdr l1))))
        (else (partition compare (cdr l1))
        )))

  (define (quicksort l1)
     (cond
        ((null? l1) '())
        (else (let ((pivot (car l1)))
           (append (append (quicksort
           (partition (lambda (x)(<x pivot))
           l1))
          (partition(lambda(x)(=x pivot)) l1
           (quicksort (partition (lambda (x)
            (> x pivot)) l1))))
            )))

(quicksort (read))
(sleep 5)
```

In Fig. 3, we can observe the number of generations that our GA had to pass for reaching a stable score. In Table I, we can see that we managed only to evade two of the test cases given. In this situation, we can argue that while the original program contains less number of code lines, this would be less likely to obtain a useful mutant program. The mutated code is the following:

```
Quicksort: Mutated version.

(define (partition compare l1 )
  ( cond (   ( null? l1 ) '( ) )
         (   ( compare ( car l1 ) ) (cons(
         car l1 ) ( partition compare
         ( cdr l1 ) ) ) )
         ( else ( partition compare(cdr l1)
         ) ) ) )

(define (quicksort l1 )
  ( cond (   ( null? l1 ) '( ) )
         ( else (let ((pivot ( car l1)))
           ( append ( append ( quicksort
           ( partition ( lambda ( x )
           (>x pivot) )l1)) --modified line
           partition ( lambda ( x )
           ( = x pivot ) ) l1 ) )
           (quicksort(partition(lambda(x)
           ( > x pivot ) ) l1 ) ) ) ) ) ) )

( quicksort ( read ) )
```

and in Fig. 4 we can observe the graph generated of a mutant version of Quicksort.

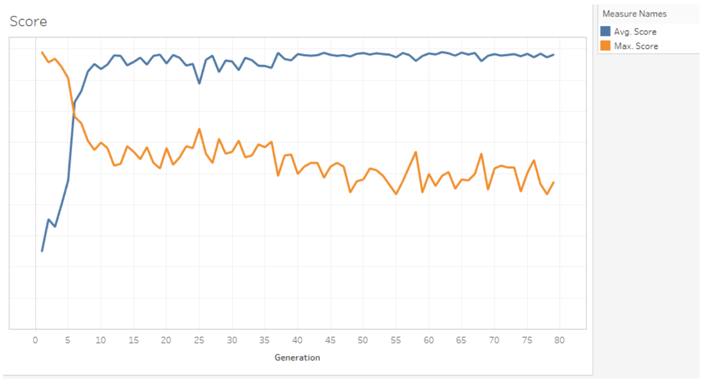

Fig. 3. Number of generations performed by the GA for the Quicksort program

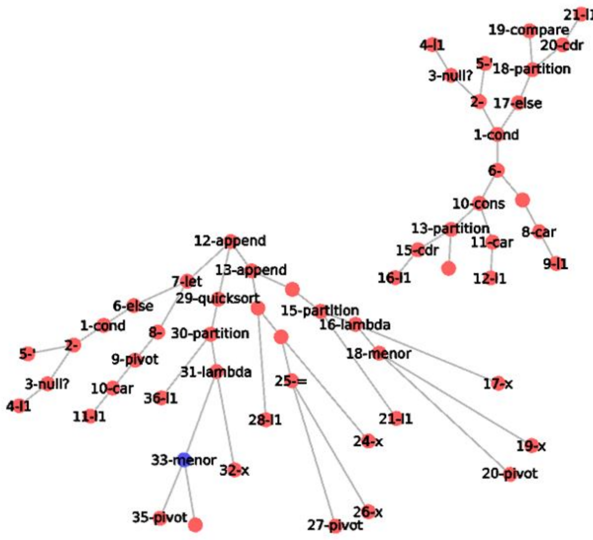

Fig. 4. Generated graph sample of a mutated version of Quicksort

B. Quadratic Equation

In this example, we tried to generate mutant versions of a quadratic equation solver. We can see in Table II, the different test cases with their corresponding correct outputs that should be returned by each mutated program. The approximate running time for the generation of mutants was roughly 10 hours, and a version of the original program is the following:

```
Quadratic Equation Solver: Original source code.
#lang racket
(define (calc-disc a b c)
  (+ (expt b 2) (* -1 (* 4 a c))))

(define (quad-eq a b c)
  (if (< (calc-disc a b c) 0)
      (begin
        false
        )
      (begin
        (if (= (calc-disc a b c) 0)
            (/ (* -1 b) (* 2 a))
```

TABLE II. INPUTS AND OUTPUTS OF THE THREE DISTINCT TEST CASES TO BE USED IN THE QUADRATIC EQUATION SOLVER.

	Test 0	Test 1	Test 2
Input	0.5 0.5 0.125	1 1 9	8 20 2
Output	-0.5	#f	-0.10435 -2.39564

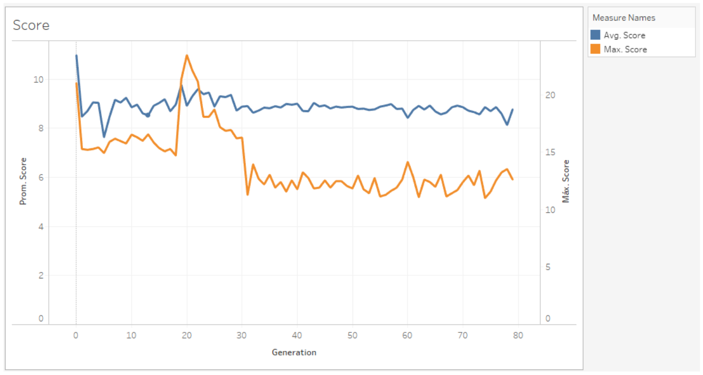

Fig. 5. Scores obtained in each of the generations

```
        (values (calc-eq a b (sqrt
        (calc-disc a b c)))
        (calc-eq a b (* -1 (sqrt
        (calc-disc a b c)))))
    ))))


(define (calc-eq a b d)
  (/ (+ (* -1 b) d)(* 2 a)))


(quad-eq (read) (read) (read))
```

In Fig. 5 and Table III, we can observe that the generation in which we obtained a mutant that bypassed the three test cases appeared on generation 19. The presence of this mutant means that the test case was not able to find unusual situations in a flawed version of the program. This implies that the test suit requires improvements or to cover a more considerable amount of probable cases that would allow the mutant to be detected.

TABLE III. NUMBER OF TESTS EVADED BY THE QUADRATIC EQUATION MUTANT SOLVER

	Generation	Altered Nodes	Evaded Tests	Score
Mutant 1	20	7,19	3	23.4392
Mutant 2	21	7,19	3	22.1092
Mutant 3	19	7,19	3	21.3392

The source code of the mutant that managed to evaded the three test cases is the following:

```
Quadratic Equation Solver: Mutated Version.

#lang racket
(define (calc-disc a b c )
  ( + ( expt b 2 ) ( * -1 ( * 4 a c ) ) ) )
(define (quad-eq a b c )
  ( if ( < ( calc-disc a b c ) -1 )
        ( begin false )
           ( begin
              (if ( = ( calc-disc a b c ) 0)
                ( * -1 b )  --missing (* 2 a)
                ( values ( calc-eq a b
                ( sqrt ( calc-disc a b c )))
                ( calc-eq a b ( * -1
                ( sqrt ( calc-disc a b c )))
                ) ) ) ) ) )

(define (calc-eq a b d )( / ( + ( * -1 b )d)
( * 2 a ) ) )

( quad-eq ( read ) ( read ) ( read ) )
```

TABLE IV. INPUTS AND OUTPUTS VALUES FOR THE GENERATION OF A TEST SUITE USING THE ALGORITHMS OF PRIM AND KRUSKAL

	Test 0	Test 1	Test 2	Test 3
Input	((4 0 1) (4 0 2) (6 0 3) (6 0 4) (2 1 2)(8 2 3) (9 3 4))	((75 0 2) (9 0 1) (95 1 2) (19 1 3) (42 1 4) (51 2 3) (31 3 4))	((13 0 3) (24 0 1) (22 0 4)(13 0 2) (22 1 2)(13 1 3) (13 1 4) (19 2 3) (14 2 4) (19 3 4))	((10 0 1) (10 1 2) (8 1 6) (13 1 7) (8 2 7) (132 8) (10 2 3) (10 3 4) (8 3 8) (10 5 6) (10 6 7) (10 7 8) (10 8 9))
Output	'((0 1) (0 3) (0 4) (1 2) (1 0) (2 1) (3 0) (4 0))	'((0 1) (1 3) (1 0) (2 3) (3 1)(3 2) (3 4) (4 3))	'((0 3)(0 2) (1 3) (1 4) (2 0) (3 1) (3 0) (4 1))	'((0 1) (1 6) (1 0) (2 7) (3 4) (3 8) (4 3) (5 6) (6 1) (6 5) (7 2) (8 9) (8 3) (9 8))

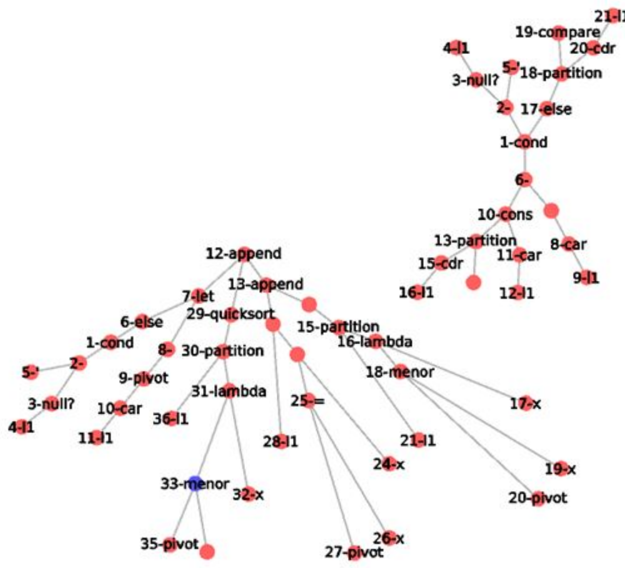

Fig. 6. Scores obtained in each of the generations

, and the generated graph of the mutant can be observed in Fig. 6:

C. Prim and Kruskal

As the latest test for our proposal, we applied our technique for the generation of mutants of two well-known algorithms such as Prim and Kruskal. For these two algorithms, we decided to prove how many tests were evaded from a test suit of four items. We generated approximately 100 mutants, and our GA ran for 80 generations. The average time for obtaining useful mutants was roughly about 20 hours. In Table IV, we can observe the corresponding inputs and the right outputs that should be obtained when executing these algorithms.

TABLE V. DESCRIPTION OF THE MUTANT THAT MANAGED TO AVOID THE THREE TEST CASES FOR THE ALGORITHM OF PRIM

	Generation Number	Altered Nodes	Evaded Tests	Score
Mutant 1	15	9, 10, 11, 12, 13, 21, 25	3	29.2684

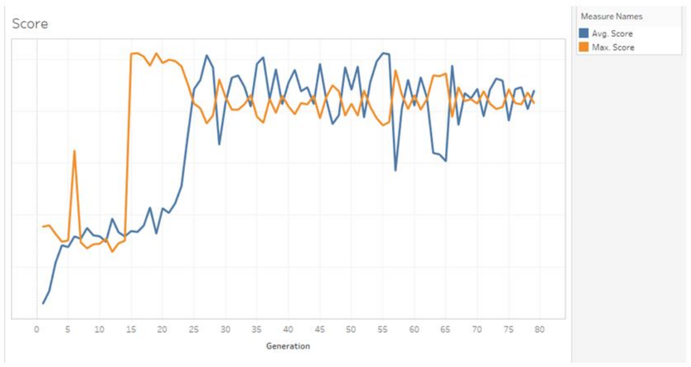

Fig. 7. Scores obtained in each of the generations

In Fig. 7, we can observe the number of generations that had to pass until our GA converged to a stable value for the Prim case. In Table V, we can see the data from a mutant that was not detected by the three cases of the test suite and was found in generation number 15.

The original source of the implementation of the Prim algorithm is the following:

```
Prim Algorithm source code: Original version.

#lang racket
(require graph)

(define grafoP(weighted-graph/undirected null))

(define (mayor valor lista)
  (if (empty? lista)
      valor
      (if (> (car valor) (caar lista))
          (mayor valor (cdr lista))
          (mayor (car lista) (cdr lista)))))

(define (orden x y)
  (if (= (length y) 0)
       x
      (orden (cons (mayor '(-1 -1 -1) y)x)
             (remove (mayor '(-1 -1 -1) y)
             y))))

(define ordenar (lambda (lista)
       (orden '() lista)))

(define (agregarVecinos grafoP lista nodo)
  (if (empty? lista)
      '()
      (if (or (= (cadar lista) nodo) (=
       (caddar lista) nodo))
          (if (and (= (cadar lista) nodo)
              (not (has-vertex? grafoP
              (caddar lista))))
              (cons (list (caar lista)
              (caddar lista) nodo)
                    (agregarVecinos grafoP
                    (cdr lista) nodo))
              (if (and (= (caddar lista)
               nodo) (not (has-vertex?
               grafoP (cadar lista))))
                  (cons (list (caar lista)
                  (cadar lista) nodo)
                   (agregarVecinos grafoP
                   (cdr lista) nodo))
                  (agregarVecinos grafoP
                  (cdr lista) nodo)))
        (agregarVecinos grafoP (cdr lista)
         nodo))))


(define (prim grafoP nodos temporal nodo
 primero)
  (if (equal? primero #t)
      (let ((aux(remove-duplicates(ordenar
      (agregarVecinos grafoP
      nodos nodo)))))

        (begin
          (add-edge! grafoP (cadar aux)
          (caddar aux) (caar aux))
          (prim grafoP nodos
          (remove-duplicates
          (ordenar
          (append (cdr aux)
           (agregarVecinos grafoP
           nodos (cadar aux))))) nodo #f)))
      (if (empty? temporal)
          (get-edges grafoP)
          (let ((aux (remove-duplicates
          (ordenar (append temporal
            (agregarVecinos grafoP nodos
            (cadar temporal)))))))
            (if (not (has-vertex? grafoP
            (cadar aux)))
                (begin
                  (add-edge! grafoP (cadar
                  aux)(caddar aux)(caar aux))
                  (prim grafoP nodos(cdr aux)
                  nodo #f))
                (prim grafoP nodos (cdr aux)
                 nodo #f))))))

(define nodos (read))
```

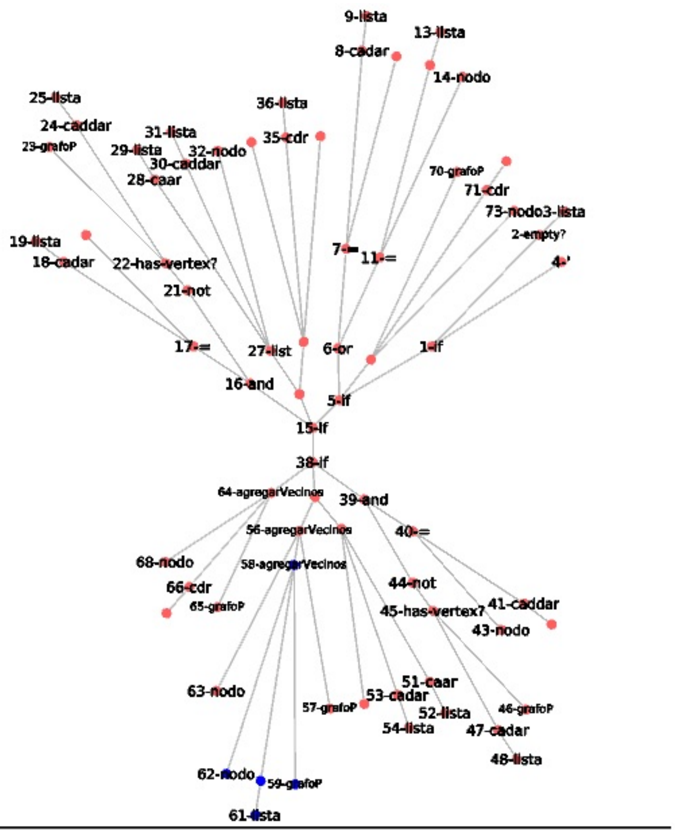

Fig. 8. Graph generated for the mutated version of the Prim algorithm

```
(prim  grafoP nodos '() 0 #t)
```

and a section of the mutated version of the source code with the portions changed commented:

```
Prim Algorihtm source code: Mutated Version

#lang racket
(define grafoP ( weighted-graph/undirected
 null ) )

(define (mayor valor lista )( if ( empty?
 lista )
         valor (if ( > ( car valor )(caar
         lista ) )
         ( mayor valor ( cdr lista ) )
         ( mayor ( car lista )
         ( cdr lista ) ) ) ) )

(define (orden x y )( if ( = ( length y )0)
x ( orden
        ( cons ( mayor '( -1 -1 -1 ) y ) x)
        ( remove ( mayor '( -1 -1 y ) y ) y)
        --modified line) ) )

(define ordenar ( lambda ( lista )
( orden '( ) lista ) ) )
```

The generated graph of the mutant can be observed in Fig. 8.

When we tested an implementation of the Kruskal algorithm we also found a mutant that, with sublet modifications, was able to evade all the test cases defined. The source code of our implementation was:

```
Kruskal implemention: original source code

#lang racket

(define nodos1 (read))

(define grafoK (weighted-graph/undirected
 null))

(define (mayor valor lista)
  (if (empty? lista)
      valor
      (if (> (car valor) (caar lista))
          (mayor valor (cdr lista))
          (mayor (car lista) (cdr lista)
          ))))

(define (orden x y)
  (if (= (length y) 0)
      x
      (orden (cons (mayor '(-1 -1 -1) y)x)
      (remove (mayor '(-1 -1 -1) y) y))))

(define ordenar (lambda (lista)
       (orden '() lista)))

(define ciclo (lambda (v1 v2 grafo)
     (if (and (has-vertex? grafo v1)
     (has-vertex? grafo v2))
       (let ((aux1 (get-neighbors grafo v1))
       (aux2 (get-neighbors grafo v2)))
       (if (and (equal? (member v2 aux1) #f)
       (equal? (member v1 aux2) #f))
          #t
          #f))
          #f)))


(define kruskalAux (lambda (grafoK nodos
funcionCiclo primero)
      (if (equal? primero #t)
         (begin
           (add-edge! grafoK (cadar nodos)
           (caddar nodos) (caar nodos))
           (kruskalAux grafoK (cdr nodos)
           funcionCiclo #f))
           (if (empty? nodos)
               (get-edges grafoK)
               (if (equal? (funcionCiclo
               (cadar nodos)
               (caddar nodos) grafoK) #f)
                   (begin
                      (add-edge! grafoK (cadar
                      nodos) (caddar nodos)
                      (caar nodos))
                      (kruskalAux grafoK (cdr
                      nodos) funcionCiclo #f))
```

```
                        (kruskalAux grafoK (cdr
                        nodos) funcionCiclo
                        #f))))))

(define (kruskal grafoK nodos)
  (kruskalAux grafoK (ordenar nodos)ciclo #t)

(require graph)

(kruskal grafoK nodos1)
```

and the mutated version was the following, notice that we have only included the portion of the source code that was mutated:

```
Kruskal implementation: Mutated version

#lang racket

(define ciclo
  ( lambda ( v1 v2 grafo )
     ( if ( and ( has-vertex? grafo v1 ) (
     has-vertex? grafo v2 ) )
          ( let (    ( aux1 ( get-neighbors
          grafo v1 ) )
            (aux2 ( get-neighbors grafo v2)))
              ( if ( or ( equal? --mutated line
                    ( member v2 aux1 ) #f )
              ( equal? ( member v1 aux2 ) #f )
              ) #t #f ) ) #f ) ) )


(define (kruskal grafoK nodos )( kruskalAux
 grafoK( ordenar nodos ) ciclo grafoK ) )
 --mutated line
```

In Fig. 9 we have shown a graph of the mutated version of the implementation of the Kruskal algorithm, while in Table VI, we depict the number of generations that had to pass for our GA to reach a convergence point. Also, we show which nodes were altered and the final score of the mutant obtained.

TABLE VI. Description of the mutant that managed to avoid the three test cases for the algorithm of Kruskal

	Generation Number	Altered Nodes	Evaded Tests	Score
Mutant 1	10	6,24	4	38.64

We can derive some discussion about the results obtained from the three programs that we tested. In the case of the Quicksort, we could argue that the test set evaluates all the probable paths of the mutated codes. Therefore, and because it does not have so many lines of code that could be mutated, then the tests successfully detect them. For the Quadratic equation code, we found a similar case that when we test the Quicksort one, the limit number of lines of code and the inclusion of a mutated math operator (/*(2 a)) it did not alter the result when compared to a test. Consequently, we can conclude that compact source codes will be less prone to exhibit a variety of mutations when applied a heuristic technique. For the case of the Prim and Kruskal algorithm, which were form by moderate lines of code, we found subtle modifications present on the mutated codes, that bypassed some test cases on our test suit.

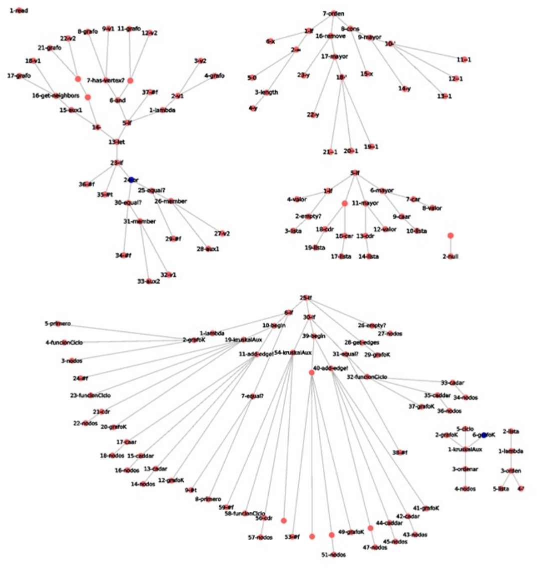

Fig. 9. Graph generated for the mutated version of the Kruskal algorithm

V. Discussion

Mutation testing techniques have been broadly used for testing a source code based on imperative programming, while research oriented to perform mutation testing on functional paradigms is rarely explored [11] survey). We argue that a primary reason could be that pure functional languages have a rigorous and formal way to develop programs in these environments, making them less prone to flaws due to their mathematical nature. Pure functional languages, such Haskell, do not endorse some constructs that are present in imperative programming languages, such as side-effects presented by a change of state in a program, but impure functional languages can exhibit these characteristics. Impure functional programming languages, for example, Racket, has increased the interest in new paradigms such as the LOP way of programming that can be used for deploy Domain Specific Languages within a chosen language (eDSL) applications or software that consolidate different programming solutions into a host programming language [12]. Therefore, we believe that even though the LOP paradigm could make more suited and tailored applications developed by programmers, it will be necessary to test these developed programs, and mutation testing joined with heuristics could be a valuable starting point to detect probable flaws in our developed software products based on this new paradigm.

VI. Conclusions

We have presented a set of proofs for the implementation or generation of mutants that can be applied for examining test suites, oriented to functional programs by using the Scheme programming language. We were able to determine that as long as the lines of codes increase, the task of generating mutants by using heuristics, the generation of mutants is achievable such that test suites could not detect them. The use of mutation testing employing heuristics, such as Genetic Algorithms, allows the tester to cover a more extensive broad of possibilities in the generation of mutants that help in detecting flaws into the test suites.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*, 1st ed. USA: Cambridge University Press, 2008.

[2] Konstantinos Adamopoulos, Mark Harman and Robert M. Hierons. *How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution* Proceedings of the 2004 Conference on Genetic and Evolutionary Computation GECCO 04, Springer. Seattle, USA, 2004.

[3] Timothy A. Budd and Dana Angluin. *Two notions of correctness and their relation to testing*. Acta Informatica, vol. 18, pp.31-45, Mar. 1982.

[4] A. Jefferson Offutt and Jie Pan. *Automatically detecting equivalent mutants and infeasible paths* . Software Testing, Verification and Reliability, vol.7, pp. 165-192, 1997.

[5] Leonardo Bottaci. *A Genetic Algorithm Fitness Function for Mutation Testing*. SEMINAL: Software engineering using metaheuristic inovative algorithms, workshop, 2001.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[7] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1996.

[8] S. N Sivanandam and S. N. Deepal. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 2007.

[9] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[10] D. Le, M. Amin Alipour, R. Gopinath, and A. Groce, *MuCheck: an extensible tool for mutation testing of haskell programs*. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). ACM, New York, NY, USA, pp. 429-432, 2014. DOI: https://doi.org/10.1145/2610384.2628052

[11] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, *Mutation Testing Advances: An Analysis and Survey*. Advances in Computers, 2018.

[12] M. Felleisen, R. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, *A programmable programming language*. Commun. ACM 61, 3 , pp. 62-71, February 2018. DOI: https://doi.org/10.1145/3127323