redin Revista Facultad de Ingeniería, Universidad de Antioquia, No.109, pp. 35-47, Oct-Dec 2023

Search-based reduction model for unit testing

Modelo de reducción basado en búsqueda para pruebas unitarias

Perla Beatriz Fernández-Oliva [1], Alejandro Miguel Güemes-Esperón [1*], Martha Dunia Delgado-Dapena [1], Alejandro Rosete [1]

[1]Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana "José Antonio Echeverría", CUJAE. Calle 114 # 11901, Ciclovía y Rotonda, Marianao. C. P. 19390. La Habana, Cuba.

CITE THIS ARTICLE AS:

P. B. Fernández-Oliva, A. M. Güemes-Esperón, M. D. Delgado-Dapena and A. Rosete. "Search-based reduction model for unit testing", *Revista Facultad de Ingeniería Universidad de Antioquia*, no. 109, pp. 35-47, Oct-Dec 2023. [Online]. Available: https://www.doi.org/10.17533/udea.redin.20221098

ARTICLE INFO:

Received: March 02, 2022
Accepted: October 18, 2022
Available online: October 18, 2022

KEYWORDS:

Automatic test generation; unit tests; search based tests; software testing

Generación automática de pruebas; pruebas unitarias; pruebas basadas en búsqueda; pruebas de software

ABSTRACT: Software tests are fundamental in the reliability and quality of systems, contributing to their positioning in the market. Generating test data is a critical task, as exhaustive testing is costly in time and effort. An adequate design of the test cases, which contemplates a selection of adequate values, can detect a high number of defects. The effectiveness of the test cases is measured according to the number of errors they managed to detect. However, the proposals that address these issues with the use of heuristic algorithms focus on the reduction of generation time and different coverage criteria. This article presents a search-based optimization model for the generation of unit test suites that integrates different test case design techniques considering the significance of the values generated in the detection of errors. The significance of the paths is also taken into account, with the aim of obtaining test cases with greater potential to detect errors. The optimization model uses heuristic algorithms that maximize the coverage of the paths. The results of the experimentation are presented, which show that the proposal presented generates test suits with a high capacity to detect errors. For this, the effectiveness of the generated test suits to detect errors in the mutated code was evaluated.

RESUMEN: Las pruebas de software son fundamentales en la confiabilidad y calidad de los sistemas, contribuyendo a su posicionamiento en el mercado. La generación de datos de prueba es una tarea crítica, ya que las pruebas exhaustivas son costosas en tiempo y esfuerzo. Un adecuado diseño de los casos de prueba, que contemple una selección de valores adecuados, puede detectar un elevado número de defectos. La significación de los casos de prueba se mide en función del número de errores que consiguen detectar. Sin embargo, las propuestas que abordan estos temas con el uso de algoritmos heurísticos se enfocan en la reducción del tiempo de generación y diferentes criterios de cobertura. Este artículo presenta un modelo de optimización basado en búsqueda para la generación de conjuntos de pruebas unitarias que integra diferentes técnicas de diseño de casos de prueba considerando la significación de los valores generados en la detección de errores. También se tiene en cuenta la significación de los caminos, con el objetivo de obtener casos de prueba con mayor potencial para detectar errores. El modelo de optimización utiliza algoritmos heurísticos que maximizan la cobertura de los caminos. Se presentan los resultados de la experimentación, que muestran que la propuesta presentada genera suit de prueba con una alta capacidad para detectar errores. Para ello se evaluó la efectividad de las suits de prueba generadas para detectar errores en el código mutado.

1. Introduction

At present, it is necessary for software products to obtain a quality certification in order to compete in an increasingly demanding market [1, 2]. Software testing is one of the activities that contributes to quality in the software development process [3]. They allow detecting defects

* Corresponding author: Alejandro Miguel Güemes-Esperón
E-mail: aguemes@tesla.cujae.edu.cu
ISSN 0120-6230
e-ISSN 2422-2844

in the source code during the development of a software product [4, 5]. The tests are left for the last stages of the project and are not carried out with the necessary quality. Due to this, it is necessary and it is of great benefit, the automation of the testing process in order to reduce its costs and increase its effectiveness [6–9].

One of the main tasks related to the design of the test cases is the creation of test data, which represents approximately 40% of the total test costs [10, 11]. Generating test data is a crucial task, as extensive testing is very costly in time and effort [12]. In this context, the appropriate design of the test suit is important, including a selection of the appropriate values, to detect a large number of defects [13], which is the objective of the software test.

Software testing continues to occupy space in the scientific papers of multiple researchers. In particular, the generation of test paths and values to support the design of test cases [14–16], as well as the processes related to software testing [17–19].

There are some proposals that focus on planning and calculating the indispensable means to carry out the tests [20], as well as on the automatic generation of scenarios [21] and test values [11] in addition to other proposals that introduce metaheuristics for the automated generation of tests that address their solution as an optimization problem [22–24]. The main objectives of these proposals are the reduction of the time associated with this process, the simplification of its execution by developers and testers, and the obtaining of wide degrees of coverage, reducing the time used to carry it out. An exception is the mutation test because the test cases are selected from the defects they detect, but for this, the test suit is run on the mutated code. This requires additional effort in running the mutant tests.

Therefore, it is necessary to work on proposals that consider those values that can identify a greater number of defects, as well as the paths that have the greatest significance. The integration of different test case design techniques [25] can allow the generation of values that can determine a greater number of defects. The values that respond to different techniques must have a greater capacity to detect errors. Then, with these values, combinations of values are generated in reduced form. One way to consider the effectiveness of the tests is to introduce in the heuristics and in the objective function, some criteria of the significance of the values in terms of the number of errors that they can detect.

This work presents an optimization model focused on error detection, to achieve a reduction of the unit test suite from the moment of its generation, using heuristic algorithms. The proposal has been implemented in a tool that allows the generation of a unit test suite in production environments, considering the significance of values and paths for error detection.

2. Related work

There are different works that apply metaheuristic algorithms for the generation of test values. In [26], it is provided a comparative analysis investigating the impact of four common constraint handling techniques (Check, Solve, Tolerate, and Replace) on six widely used combinatorial test set (coverage matrix) generation algorithms: The Automatic Efficient Test Generator (AETG), Deterministic Density Algorithm (DDA), In-Parameter-Order (IPO), Particle swarm optimization (PSO), Simulated annealing (SA) and Tabu search (TS). They posit that the constraint controller is a crucial factor influencing the performance of the test suite generation algorithms on which it is developed. The Verify technique implemented with the Minimal Forbidden Tuple (MFT) approach is the fastest option for handling constraints. The replacement technique that resolves constraints as a post-processing phase tends to produce smaller constrained coverage matrices than the currently widely used Verify and Solver techniques, especially for one-test-at-a-time framework test suite generation algorithms. They also show that it is important to choose a constraint controller specifically suitable for the algorithm and the specific goal (size of the test suite, computational cost, or fault-revealing ability). For example, to generate the smallest constrained coverage matrices, *Replace* is the best choice for AETG, DDA, and PSO; while *Tolerate* might be more promising for SA and TS. This study only provides information for the choice of constraint controller, so that the performance of existing and newly designed test suite generation algorithms can be improved. It also offers a better understanding of the strengths and weaknesses of constraint management techniques.

In [27], two metaheuristic algorithms are merged to improve the efficiency of the generated test cases. It uses a genetic algorithm to generate good test data from previously generated candidates and the tabu search is added to the mutation step of the genetic algorithm to reduce search time.

In [28], it is introduced a Particle Swarm Optimization (PSO) based test data generation method that can generate a test data set to cover multiple target paths in a single run. In the work, a new training function is designed that can guide the data collection to achieve multipath coverage and avoid premature convergence. For the multipath coverage problem, different fitness functions

are applied to evaluate the best individual position and the best overall position. There are several concepts covered in the proposal. Route fitness: the sum of the weighted branches, distance function. Fitness for the best individual position: it is the minimum of the single-path fitness functions; it guides the particles so that they do not converge on a specific path. Fitness function for the best global position: the sum of those single-route fitness functions guides the population to achieve multi-route coverage and avoid premature convergence.

In [29], genetic algorithms that start from the domain are used to generate suitable values that satisfy previously defined test criteria. The established criteria are: obtaining complete coverage of the branch, controlling the number of iterations, and obtaining a set of test data for structural tests. It is applied to several problems whose complexity varies, from a solver of quadratic equations to a generic ordering module that includes several procedures. In these cases, full branch coverage was obtained. To obtain better quality in the generated test data, they design a fitness function that generates data near to subdomain boundary.

In [30], an alternative based on the combination of population metaheuristics with a Tabu List for the generation of test cases is presented. The performance of the solution is tested with a set of programs of varying complexity. It takes into account the coverage in the function by using the particle swarm algorithm to compare the value it gets with a pre-established coverage criterion.

However, there are still some limitations in these proposals. For example: algorithm parameters may affect the efficiency of test data generation, and manual construction and instrumentation of training functions are time-consuming, especially for complicated programs. Furthermore, these proposals do not take into account the criterion of the significance of the values they generate in terms of the number of errors that they can detect. Its validations include comparisons of generation time and a number of combinations of values generated, but do not analyze the effectiveness of the combinations of values generated.

3. Background of this work

The fundamental objective of the previous proposals is to reduce the time associated with this process, to simplify its execution by developers and testers, and to obtain wide degrees of coverage by reducing the time used to carry it out. Although coverage criteria associated with Software Engineering techniques are used, there is no emphasis on the use of Software Engineering (IS) design techniques that can provide additional information to generate test suites with high levels of error detection. This model is part of the framework, MTest.search. This framework integrates test execution workflows, optimization models for test case reduction, and integrated software tools that support workflow execution [31]. In this paper, the proposed heuristic function $fh\left(\bar{\gamma}_j\right)$ is expressed in terms of knowing how much the new combination contributes to the equivalence partitions that were already identified in the combinations of previous values generated. Therefore, this is valid only for functional tests and does not consider the significance of input values, scenarios, or paths.

The software tools that support MTest.search have been developed following a component-based architecture shown in Figure 1. They are organized in three layers: test suite generation, extension, and user environment. "User environment" layer: It contains the tools customized to the specific production environments of the user. There is currently a plugin for the Eclipse development environment (IDE), starting with version 3.5. It contains the client application that captures the input domain information and displays the reduced language test suite or corresponding output artifact.

"Extension" layer: It contains the extensions developed for various input domains and output formats. This layer provides mechanisms to extend to other input domains and output formats. It contains the extensions of the CFG_ANTLRGeneration component of the "Test suite generation" layer:

- *CFG_ReqExt* for descriptions of requirements such as input domain and
- *CFG_ANTLRGenerationExt* for source code in Java language.
- *GeCodP* that allows advanced users to use the extension mechanisms provided for the domain model, the test model and the execution model. It is the one in charge of communication with the rest of the components and the user environment.
- *TestCodeGeneration* is an extension of the "Test suite generation" layer component that incorporates the generation of test code in new output formats.

"Test suite generation" layer: It is responsible for generating unit test cases regardless of input domain and output format. Contains the components:

- *CFG_ANTLRGeneration* for the generation of the control flow graph from the input domain.
- *GeCaP* for the generation of the reduced test suite using the GeVaUn, GeVaF and GeVaU components.
- *TestCodeGeneration* for the transformation of the test suite in the different output codes.

Algorithms for generating independent paths from Java source code and for generating JUnit code from test cases were developed in [32]. These algorithms are included in the CFG_ANTLRGeneration and TestCodeGeneration

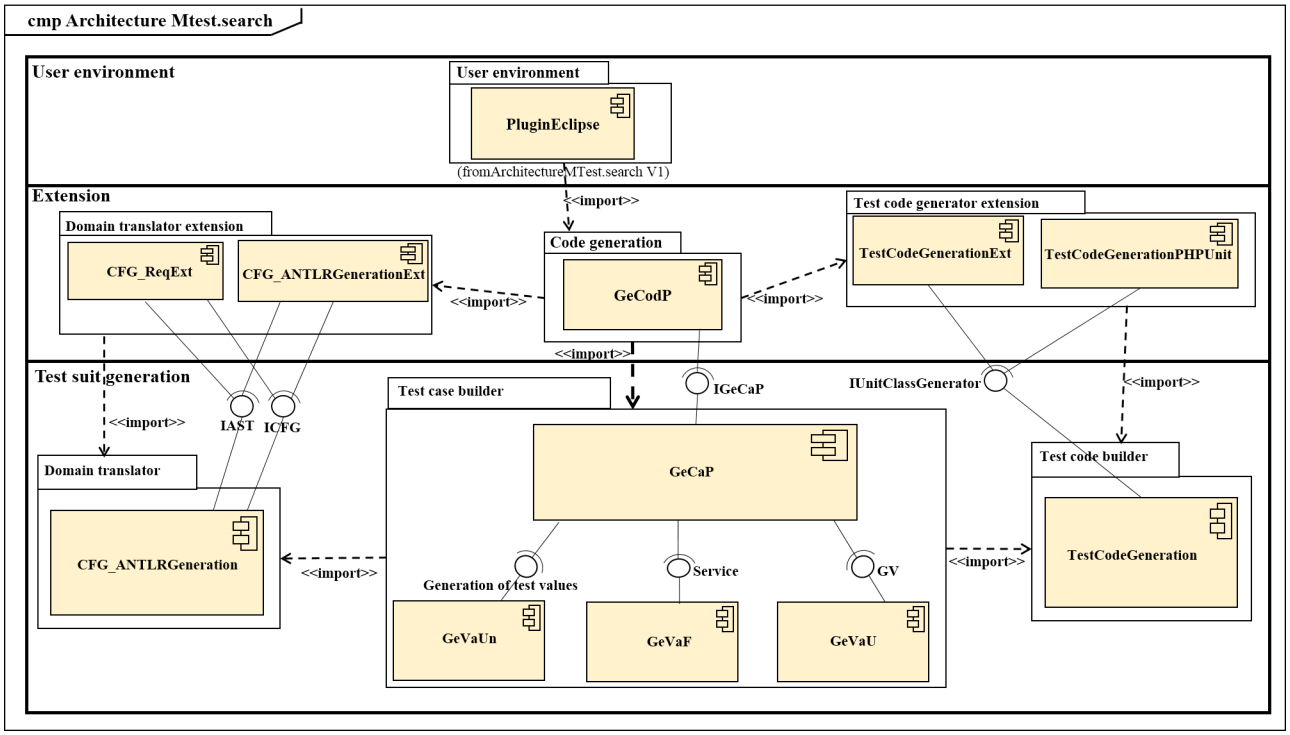

Figure 1 Architecture of the model support tools. Modified from [30]

components, respectively. GeCaP encapsulates the interaction with the other components of the extension layer. The user environment layer integrates through interaction with *GeCaP* all the functionalities contemplated in the developed components. In this paper, it can be seen that the proposal implemented in the support tool generates a reduced set of values and, in addition, obtains the results in less time than the proposals consulted in the previous bibliographies.

The reduction model is in the central core, but connected to source code translators that allow transforming it to the reduction model, and the inputs of the reduction model can also be arranged for the generation of test code. This tool focuses on productive environments through client applications that can be developed. In particular, [33] presents a tool developed for the Eclipse environment that generates code in JUnit from source code in Java language and that contains translators to analyze this language.

MTest.search includes a specific optimization model for functional test reduction (MOPF) presented in [31] and extension mechanisms to incorporate new reduction models. In [34], an optimization model for unit test reduction (MOPU) is defined that includes only path coverage, without taking into account the importance of values and paths. The extension mechanisms to add new input and output languages are detailed in [33].

4. Materials and methods. Search-based reduction model for unit tests

The fundamental contribution of this work is the definition of an optimization model that allows generating reduced unit test suites, incorporating in [34] the significance of paths and input values. The GeVaU component presented in [35] implements the model proposed in this paper. This component was developed to generate combinations of values to perform unit tests (GeVaU) where the generated combinations take into account the independent paths of the unit to be tested.

The search process was defined, which is the generation of the initial solution, the operators, and the stop criterion. To generate the initial solution, it was decided to use a random construction of the initial state. Therefore, a vector of length equal to the number of variables in the problem is generated. The new solutions to the problem are generated from the use of operators. The mutation operator is used at one point. This operator consists of randomly selecting one, or two or three positions of the coded solution and changing its value for another randomly within the possible values that this variable can take. Unlike other proposals, this one takes into account white box design techniques.

The unit test search-based reduction model (MOPU) assumes the hypothesis that, if combinations of values are selected that have been chosen from the application of test case design techniques considering the significance of paths and input values, then the test suite will detect more errors. Therefore, a model is proposed that maximizes path coverage with a reduced number of combinations of test values, considering the significance of the values in terms of error detection. The optimization aims to find a reduced set of test values that maximize the coverage of the route and thus provide a possible solution to the combinatorial explosion problem. From the search-based reduction model for tests in the article [31], the variables described below are maintained starting from:

- $\bar{\alpha} = (X_1, X_2, \dots X_n)$, vector containing the input variables or attributes but for the code to be tested;
- $\bar{\beta} = (y_1, y_2, \dots y_n)$, vector containing the domain description of each attribute that belongs to $\bar{\alpha}$;
- C, level of coverage to be achieved;
- n, number of input variables but for the code to test;

But since the proposal of this article works for unit tests with white box techniques, it is generated from the source code. Therefore, it is necessary to define the model with the incorporation of variables, functions, and a penalty mechanism as shown below:

- $\bar{\rho} = (l_1, l_2, \dots l_m)$, vector containing the conditional statements present in the code to be tested;
- N_{pxn}, matrix with the truth values of each conditional for each independent path of the code to be tested;
- m, number of conditionals of the code to be tested;
- p, number of independent paths of the code to be tested;

Then Equation (1) is redefined to add the significance of input values and paths:

$$MOPU\left(\left(\bar{\alpha}, \bar{\beta}, c, f_o(\bar{\varepsilon}, \bar{\gamma}, \bar{\omega})\right)\right) \rightarrow M_{lxn} \tag{1}$$

as the model by which the matrix is obtained M_{1xn} containing a test suite that satisfies the independent paths, with coverage criterion C.

- l : is the number of combinations to obtain in the generation of the test suit that satisfies the independent paths, with coverage criteria C.
- M_{1xn}, is the matrix that contains the generated value combinations and where each row j corresponds to a vector $\bar{\gamma}$.

To include the path and input value significance criteria, the objective function was defined in Equation (2):

$$f_o(\bar{\varepsilon}, \bar{\gamma}, \bar{\omega}) = \text{Max}\left[\frac{cs * \frac{\sum_{j=1}^{l} f'h(\bar{\gamma}j)}{l} + \frac{\sum_{j=1}^{l} f''h(\bar{\gamma}j)}{l}}{2}\right] \tag{2}$$

Where:

- l: is the number of test cases of the suit.
- $\bar{\omega} = (w_1, w_2, \dots, w_p)$: is the vector with the significance of p independent paths.
- $\bar{\gamma} = (\varphi_1, \varphi_2, \dots, \varphi_j)$: is the vector containing the test suit.
- cs: is the path coverage of the test suite.

This function evaluates the test suite considering two heuristics: $f'h\,(\bar{\gamma}_j)$ shown in Equation (3) y $f''h\,(\bar{\gamma}_j)$ shown in Equation (4), for the significance of the paths and the input values respectively such that $0 \leq fh' \leq 1$.

$$fh'(\bar{\gamma}j) = \frac{\sum_{s=1}^{p} W_s * \delta\,(\bar{\gamma}_j)}{p} \tag{3}$$

$$fh''(\bar{\gamma}j) = \frac{\sum_{n=1}^{CT} \frac{\text{potential variable value } h}{\text{maximum power to detect errors}}}{CT} \tag{4}$$

Where:

- p: Total number of independent paths.
- W_s: Path significance s, value between 0 and 1 such that $\sum_{s=1}^{p} W_s = 1, W_s = \frac{\text{number of conditions}}{\text{total number of conditions}}$
- CT: Total number of variables in the problem.

Equation (5) represents the number of conditionals within the path that are satisfied by combining values j , which was just generated.

$$\delta\,(\bar{\gamma}_{ji}) = \frac{\sum_{i=1}^{m} C_{ji}}{m}$$

$$C_{ij} = \begin{cases} 1, \text{ if the combination of the value } j \text{ satisfies} \\ 0, i.a.c \quad \text{the condition } i \end{cases} \tag{5}$$

To generate the initial test suite, a penalty mechanism is used that guarantees that in this first suit all paths are satisfied with at least one test case. To do this, a penalty mechanism has been incorporated as part of the model, which is shown in Equation (6). If the new combination of values is such that:

$$fh\,(\bar{\gamma}_j) = \sum_{s=1}^{S} \frac{\delta_s\,(\bar{\gamma}_{ji})}{p} = 1 \tag{6}$$

then $p = p - 1$. The proposal presented in this paper reduces the input domains to discrete values using the transformation vector $\bar{\varepsilon} = (e_1, e_2, \dots e_k)$, similarly to [31]. The difference is that the transformation vector adds the condition and loop techniques to the existing ones.

In addition to the existing components, a new GeVaUExt component was implemented that incorporates these new elements and extends GeVaU. In this new method, the change between the method that only considers the coverage of paths and the one that includes the significance of input values and paths can be observed.

5. Results. Application of the model by using the tools developed

The validation of the proposal is aimed at showing that the generated test suite has high levels of path coverage, that the test cases included in the generated suite contain the input values and paths with the highest levels of significance, and that the generated test suite is effective in detecting errors.

Two case studies and one experiment were designed. In the case studies, the quality of the generated test suite is evaluated in terms of the importance of the input values and paths in two applications of different complexity. The experiment evaluates the effectiveness in error detection of the proposal made in this article. The case studies that were selected have different complexities and functionalities, in which an analysis is carried out considering the effectiveness of the implemented algorithms and the quality of the test set. For each case study, several suites are generated with GeVaUExt. In the first case, the inclusion of the most significant values is analyzed and in the second case, mutants are implemented that are executed with the generated suite to know the effectiveness in detecting errors. The results of the extension are compared against the results of the previous component.

Case study 1: Buy online

The case study is based on the fact that several customers can make purchases in online stores where they must register on the platform and then place several orders according to the quantity that exists in the store. For this case study, two functionalities were chosen. The description of each of the functionalities is as follows:

- Reserving product request quantity, where the user may or may not successfully reserve a requested quantity depending on the actual quantity in the store.
- Reporting the products in stock limits according to the minimum and maximum quantities.

The system generates significant values for each value of the variables by applying test case design techniques, depending on the equivalence class, according to the domain it belongs to. Some generated values are in the numeric domain and others are of type string.

Tables 1 and 2 show the conditions that describe the scenarios in the functionalities of reserving the number of products and reporting the products, respectively. Table 3 and 4 show the paths for each of the functionalities with their respective values (T, F, -) by condition, tsignificance and expected result for each of the paths.

Table 1 Conditions to reserve the number of products requested by the user

Id Conditions	Conditions
1	RequestQuantity < 0
2	RequestQuantity > 99999
3	RealQuantity < 0
4	RealQuantity > 99999

The case study has the following objectives:

1. To determine the effectiveness of the implemented algorithms to select the test cases with the most significant input values. For this, the following question is proposed. Do the generated test suites contain the test cases with the most significant values?
2. To determine the effectiveness of the implemented algorithms to select the test cases for the most significant paths. For this, the following question is proposed: Do the generated test suites contain a greater number of test cases associated with the most significant paths?

Case study 2: Classification of triangles

The functionality of the case study consists basically in classifying a triangle as scalene, isosceles, or equilateral, in addition to verifying whether or not the sides form a triangle. Tables 5 and 6 show the conditions and the paths with their values per condition. The case study has the following objectives:

1. To determine the quality of the test suites generated by the implemented algorithms, in terms of the number of errors detected. For this, the following question is proposed: What percentage of mutants are killed by each suite generated by the implemented algorithms? Five test suites were generated in five executions and with 2000 iterations for both functionalities using the implementations

To answer the first question of the buy online case study: do the generated test suites contain the test cases with the most significant values?

The significant values of the paths were fixed with a value equal to 1.0 for both functionalities. Table 7 shows the number of times the values of the variables are repeated in each of the generated suits. In this case, the values of greatest significance are 0 and 99999 with a significance of 1.0 for both variables that have a numerical domain. These values are contained in each of the test suites, having a higher frequency in most of the suits, and being very useful for error detection.

Table 2 Conditions for reporting products in stock limits

Id Conditions	Conditions
1	MinimumQuantity < 0
2	MinimumQuantity > 10000
3	MaximumQuantity < 0
4	MaximumQuantity > 10000
5	RealQuantity < 0
6	RealQuantity > 10000
7	QuantityCompared < 0
8	QuantityCompared > 10000
9	Product != ""
10	Product = ""
11	MaximumQuantity + QuantityCompared >= RealQuantity
12	MaximumQuantity - QuantityCompared >= RealQuantity
13	MaximumQuantity + QuantityCompared <= RealQuantity
14	MaximumQuantity - QuantityCompared <= RealQuantity

Table 3 Paths to reserve the quantity of products requested by the user

Id Paths	Conditions/Paths	Significance	Expected result
1	T - - - - - -	0.1429	Quantity of requests out of range
2	F T - - - - -	0.2857	Quantity of requests out of range
3	F F T - - - -	0.4286	Real quantity out of range
4	F F F T - - -	0.5714	Real quantity out of range
5	F F F F T - -	0.7143	The quantity is reserved satisfactorily
6	F F F F - T -	0.7143	The quantity is not reserved satisfactorily
7	F F F F - - T	0.7143	The quantity is not reserved satisfactorily

Table 4 Paths to report the products in limits of existence

Id Paths	Conditions/Paths	Significance	Expected result
1	T - - - - - - - - - - - -	0.0714	Minimum quantity out of range
2	F T - - - - - - - - - - - -	0.1429	Minimum quantity out of range
3	F F T - - - - - - - - - - -	0.2143	Maximum quantity out of range
4	F F F T - - - - - - - - - -	0.2857	Maximum quantity out of range
5	F F F F T - - - - - - - - -	0.3571	Real quantity out of range
6	F F F F F T - - - - - - - -	0.4286	Real quantity out of range
7	F F F F F F T - - - - - - -	0.5	Quantity to compare out of range
8	F F F F F F F T - - - - - -	0.5714	Quantity to compare out of range
9	F F F F F F F F T T - - - -	0.7143	Is displayed the stock limit report satisfactorily
10	F F F F F F F F - - T - - -	0.6429	Is displayed the stock limit report satisfactorily
11	F F F F F F F F - - - T T -	0.7143	Is not displayed the stock limit report satisfactorily
12	F F F F F F F F - - - - - T	0.6429	Is not displayed the stock limit report satisfactorily

The values 99998 and 1 tend to be repeated more in the first and fourth suit, respectively; however, their significance is high with a value of 0.8 so it can also detect errors.

The value 49999.5, which has less significance, tends to have a higher frequency than -1 and 100,000 since both values can only be contained in test cases that cover the first four paths or scenarios because they are outside

Table 5 Conditions for classifying a triangle

Id Conditions	Conditions
1	SideA>0&&sideB>0&& sideC>0
2	SideA<(sideA+sideB+sideC)/2 && SideB<(sideA+sideB+sideC)/2 SideC<(sideA+sideB+sideC)/2
3	sideA==sideB
4	sideB==sideC
5	sideA==sideC
6	sideB==sideC

Table 6 Paths for the classification of a triangle

Id Paths	Conditions/Paths	Significance	Expected result
1	F - - - - -	0.1667	The sides do not form a triangle. Sides less than or equal to 0
2	T F - - - -	0.1250	The sides do not form a triangle. Sides with a wrong sum
3	T T F – F F	0.8333	Scalene triangle
4	T T F – F T	0.8333	Isosceles triangle
5	T T F – T -	0.6667	Isosceles triangle
6	T T T F - -	0.6667	Isosceles triangle
7	T T T T - -	0.6667	Scalene triangle

Table 7 Number of times the values are repeated in each test suit, setting a single significance value of the paths

Test suit	Value 0	Value 99999	Value -1	Value 100000	Value 1	Value 99998	Value 49999.5
SP1	6	6	2	2	6	7	4
SP2	6	6	3	2	6	4	5
SP3	5	8	2	2	5	6	4
SP4	4	6	2	2	7	5	5
SP5	7	5	3	3	5	6	3

the range of 0 and 99999 by what the significant value of these values is not the correct one to obtain the test suit.

Table 8 shows the number of times the values of the variables are repeated in each test case. The values of the variables with the numerical domain are shown, where the 0 and 10,000 that have the highest significant value equal to 1.0 are contained in the test cases of each of the test sets, with the highest number of repetitions. While the values of the string domain variable with a significant value equal to 1.0 are also contained in the test cases more frequently than the value with a significance equal to 0.7.

Answering the second question of the online purchase case study: Do the generated test suites contain a greater number of test cases associated with the most significant paths?

The significant values of all the values of variables with a value equal to 1.0 for the two functionalities were set. In the Figure 2 shows the number of test cases covering each of the paths to reserve the number of products.

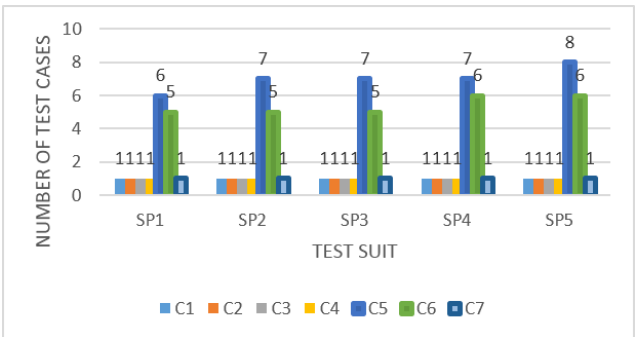

Figure 2 Number of test cases covered by each path to reserve the number of products by setting a single significance value of the input values

The most significant paths are 5, 6, and 7 whereas paths 5 and 6 are the ones that contain the most test cases.

Table 8 Number of times that the values are repeated to report the products in limits of existence, setting a single significance value of the paths

Test suit	Value 0	Value 10000	Value -1	Value 10001	Value 1	Value 9999	Value 5000
SP1	49	47	14	19	5	11	10
SP2	51	54	17	16	12	13	5
SP3	54	47	16	14	11	11	6
SP4	53	52	19	16	11	7	6
SP5	52	54	15	16	10	13	10

However, in path 7 it can only be covered by a single test case since the request quantity variable has to have a value less than or equal to 0, and in this case, the value it can always take is 0, otherwise, it is -1 so this is not in the range of 0 and 99999.

Figure 3 shows the number of test cases that cover each of the paths to report products at stock limits. In the graph, it can be seen that the number of test cases in each path varies in each of the test suites. Path 9, which is more significant, contains more test cases in each of the generated suits. However, path 11, which also has the highest significant value, is one of the paths that contain the fewest test cases. This is because it is a path that can be covered by a few combinations given the conditions it must satisfy.

The experiment was designed, with five executions of the triangle classification algorithm, to answer the following question: What percentage of mutants kills each suite generated by the implemented algorithms?

The effectiveness of each test case was measured by applying the mutation technique. For this, 65 mutants were designed with the aim of executing the test suite generated with the proposed algorithms and determining their effectiveness in detecting errors in the mutated code.

The mutation operators used were: AOR (Substitution of an Arithmetic Operator) 27 mutants, LCR (Replacement of a Logical Connector) 4 mutants, UOI (Unique Operator Insertion) 23 mutants, ROR (Replacement of a Relational Operator) 10 mutants and SDL 1 mutant [36].

Subsequently, the test cases were generated, and coded in Junit and the tests were run on each of the mutants to analyze the number of dead mutants. When carrying out the analysis, it was found that all the test cases generated in the mutation kill at least one mutant. Therefore, it was considered to make a modification in the proposed algorithm by introducing other elements of coverage such as coverage of conditions, loops, and equivalence partition. After including the new coverage criteria with the extension, test suits were generated using the hill climber and genetic algorithms. The comparison gave the result that with the first proposal effectiveness greater than 80% was obtained and with the other two algorithms the effectiveness is greater than 95%. Table 9 shows the number of test values generated under different criteria.

Table 9 Number of different values generated by the component with the three implementations

Variables	C1	C2	C3	C4	C5	C6
Side 1	13	6	6	12	11	13
Side 2	13	5	11	9	11	12
Side 3	15	6	9	11	12	14

Being:

C1: Total entry of values.
C2: Number of different values generated by the previous implementation.
C3: Number of different values with the hill climber with 20 test cases.
C4: Number of different values with the genetic algorithm with 20 test cases.
C5: Number of different values with the hill climber with 60 test cases.
C6: Number of different values with the genetic algorithm with 60 test cases.

As can be seen in Table 9, the improvements generate more combinations of test values than the previous version. This is due to the inclusion of coverage techniques that contain values contributed from the application of test case design techniques that had not been taken into account and are more effective for detecting errors. Therefore, it would be coherent to think about improving the proposed optimization model to include the significance of values in the detection of errors. More than one test case is generated for each path as the coverage criterion is expanded.

Figure 4 shows the original code that implements the triangle classification algorithm.

Figure 5 shows the mutated code, from the original code of the triangle classification algorithm, for the AOR

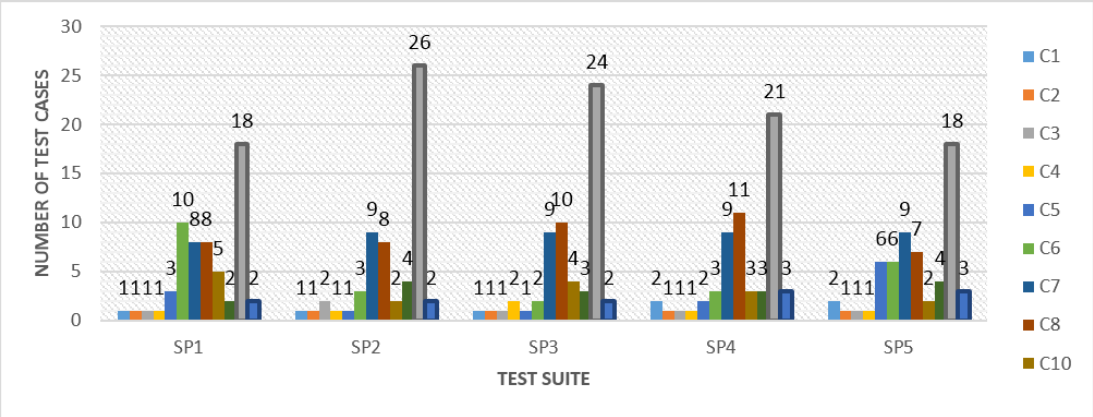

Figure 3 Number of test cases covered by each path to report products at limits of existence by setting a single significance value of the input values

```
public String triangleClassification(int side1, int side2, int side3)  {
    String result = "Los lados " + side1 + " , " + side2 + " y " + side3 + " no forman un triángulo. Lados menores o iguales que cero ";
    if (side1 > 0 && side2 > 0 && side3 > 0) {
        if (side1 < (side1 + side2 + side3) / 2 && side2 < (side1 + side2 + side3) / 2 && side3 < (side1 + side2 + side3) / 2) {
            if (side1 == side2) {
                if (side2 == side3) {
                    result = "El triángulo es equilátero.";
                }
                else {
                    result = "El triángulo es isósceles.";
                }
            }
            else {
                if (side1 == side3) {
                    result = "El triángulo es isósceles.";
                }
                else {
                    if (side2 == side3) {
                        result = "El triángulo es isósceles.";
                    }
                    else {
                        result = "El triángulo es escaleno.";
                    }
                }
                }
            }
        }
        else
            result = "Los lados " + side1 + " , " + side2 + " y " + side3 + " no forman un triángulo. Lados con suma errónea ";
    }
    return result;
}
```

Figure 4 Original code implementing the triangle classification algorithm

operator. Where it can be seen that the mutation is a change of the arithmetic operator of addition by division indicated in a circle.

Figure 6 shows the last mutant. Test cases from each of the suites were then run on the mutants. Table 10 shows the effectiveness of each of the suits.

The results show that the last test suit is better compared to the rest, with 100% effectiveness. However, despite the fact that the others are not 100% effective, they are still good at detecting errors since they managed to cover more than 95% of the errors inserted by the mutants.

Table 10 Result of the execution of the test suites on the mutants with the implemented algorithms

ID SP	Dead mutants	Effectiveness
SP_1	64	98.46
SP_2	63	96.92
SP_3	63	96.92
SP_4	64	98.46
SP_5	65	100

```
public String triangleClassification(int side1, int side2, int side3)  {
    String result = "Los lados " + side1 + " , " + side2 + " y " + side3 + " no forman un triángulo. Lados menores o iguales que cero ";
    if (side1 > 0 && side2 > 0 && side3 > 0) {
        if (side1 < (side1 + side2 + side3) / 2 && side2 < (side1 + side2 / side3) / 2 && side3 < (side1 + side2 + side3) / 2) {
            if (side1 == side2) {
                if (side2 == side3) {
                    result = "El triángulo es equilátero.";
                }
                else {
                    result = "El triángulo es isósceles.";
                }
            }
            else {
                if (side1 == side3) {
                    result = "El triángulo es isósceles.";
                }
                else {
                    if (side2 == side3) {
                        result = "El triángulo es isósceles.";
                    }
                    else {
                        result = "El triángulo es escaleno.";
                    }
                }
            }
        }
        else
            result = "Los lados " + side1 + " , " + side2 + " y " + side3 + " no forman un triángulo. Lados con suma errónea ";
    }
    return result;
}
```

Figure 5 Mutated code using the AOR operator

```
public String triangleClassification(int side1, int side2, int side3)  {
    String result = "Los lados " + side1 + " , " + side2 + " y " + side3 + " no forman un triángulo. Lados menores o iguales que cero ";
    if (side1 > 0 && side2 > 0 && side3 > 0) {
        if (side1 < (side1 + side2 + side3) / 2 && side2 < (side1 + side2 + side3) / 2 && side3 < (side1 + side2 + side3) / 2) {
            if (side1 == side2) {
                if (side2 == side3) {
                    result = "El triángulo es equilátero.";
                }
                else {
                    result = "El triángulo es isósceles.";
                }
            }
            else {
                if (side1 == side3) {
                    result = "El triángulo es isósceles.";
                }
                else {
                    if (side2 == side3) {
                        result = "El triángulo es isósceles.";
                    }
                    else {
                        result = "El triángulo es escaleno.";
                    }
                }
            }
        }
        Missing else statement
    }
    return result;
}
```

Figure 6 Mutated code removing code snippets related to error messages

The above results were compared with the execution of the GeVaU shown in Table 11. As can be seen, the test suites generated by GeVaUExt are much more effective than those generated by GeVaU.

Table 11 Comparison of the effectiveness of the test suites generated with GeVaUExt and GeVaU

ID SP	GeVaU (%)	GeVaUExt (%)
SP_1	75.38	98.46
SP_2	78.46	96.92
SP_3	78.46	96.92
SP_4	78.46	98.46
SP_5	76.92	100

Having the highest percentage of effectiveness, it allows detecting a greater number of errors.

With the incorporation of the significance of the input values and the paths proposed in this work, it can be seen that the error coverage of the test cases of each of the suits is effective. This reveals a high ability to find errors. A suit of tests is obtained with the most significant values of the variables and a greater number of test cases are generated for the paths with the greatest significance.

6. Conclusions

This work presents a search-based optimization model for the generation of unit tests integrating different test case design techniques. The transformations of the test model to the extended model are exposed. The optimization model maximizes the coverage of the path with a reduced number of combinations of test values, considering the importance of the values in terms of error detection. Regarding coverage, compared to the results presented in the other bibliographies, this proposal improves the previous proposals since it guarantees that at least one test case is generated for each path. Better results are obtained in effectiveness, compared to this same proposal since there were no data from other bibliographies because the significance of the values is not taken into account. By including more test cases, a better chance of detecting more errors is guaranteed. If combinations of values are selected that have been chosen from the application of test case design techniques considering the significance of the input values and the paths, the test suite will detect more errors. For future work, it is recommended to continue evaluating the effectiveness of the generated suites and to make adjustments to the significance of the input values and the significance of the paths.

7. Declaration of competing interest

We declare that we have no significant competing interests, including financial or non-financial, professional, or personal interests interfering with the full and objective presentation of the work described in this manuscript.

8. Acknowledgments

We thank the referees for their comments to improve this work.

9. Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

10. Author contributions

Perla Beatriz Fernández Oliva: Proposal of the model, experimental design and analysis of results. Alejandro Miguel Güemes Esperón: Implementation of the model. Martha Dunia Delgado Dapena: Experimental design and project coordination. Alejandro Rosete: Contribution of the heuristics model.

11. Data availability statement

The authors confirm that the data supporting the findings of this study are available within the article and its supplementary materials.

References

[1] M. Chaudhary and A. Chopra, *CMMI for Development: Implementation Guide*. New York, NY: Apress, 2016.

[2] B. Castro and S. Javier, "Paradigmas en la construcción de software," *Ingeniería*, vol. 6, no. 1, 2001. [Online]. Available: https://dialnet.unirioja.es/servlet/articulo?codigo=4797401

[3] L. M. J. M. Zhang, M. Harman and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, Jan. 01, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2019.2962027

[4] D. Spinellis, "State-of-the-art software testing," *IEEE Software*, vol. 34, no. 5, Sep. 22, 2017. [Online]. Available: https://doi.org/10.1109/MS.2017.3571564

[5] J. M. Belman-Flores, J. M. Barroso-Maldonado, S. Méndez-Díaz, and S. Martínez-Martínez, "Virtual test bench as a complement to study thermal area: application in vapor compression systems," *Revista Facultad de Ingeniería, Universidad de Antioquia*, vol. 77, Oct-Dec 2015. [Online]. Available: https://doi.org/10.17533/udea.redin.n77a08

[6] S. M. Edgar, M. M. Raquel, and T. O. Paula, "A review of reality of software test automation," *Computación y Sistemas*, vol. 23, no. 1, Feb. 26, 2021. [Online]. Available: https://doi.org/10.13053/cys-23-1-2782

[7] B. Oliinyk and V. Oleksiuk, "Automation in software testing, can we automate anything we want?" in *Proceedings of the 2nd Student Workshop on Computer Science & Software Engineering*, Kryvyi Rih, Ukraine, 2019, pp. 224–234.

[8] D. Atesogullari and A. Mishra, "Automation testing tools: a comparative view," *International Journal on Information Technologies & Security*, vol. 12, no. 4, 2020. [Online]. Available: http://ijits-bg.com/contents/IJITS-N4-2020/2020-N4-06.pdf

[9] G. Alor-Hernández, V. Y. Rosales-Morales, J. L. G. Alcaráz, R. Z. cabada, and M. L. B. Estrada, "An analysis of tools for automatic software development and automatic code generation," *Revista Facultad de Ingeniería, Universidad de Antioquia*, vol. 77, Oct-Dec 2015. [Online]. Available: https://doi.org/10.17533/udea.redin.n77a10

[10] M. Xiao, M. El-Attar, M. Reformat, and J. Miller, "Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques," *Empirical Software Engineering*, vol. 12, Nov. 08, 2006. [Online]. Available: https://doi.org/10.1007/s10664-006-9026-0

[11] H. Felbinger, F. Wotawa, and M. Nica, "Adapting unit tests by generating combinatorial test data," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Vasteras, Sweden, 2018, pp. 352–355.

[12] A. Spillner and T. Linz, *Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant*. Heidelberg, DEU: dpunkt.verlag, 2021.

[13] P. R. S, *Software engineering: a practitioner's approach*. New york, NY: McGraw-Hill Education, 2015.

[14] R. Sharma and A. Sahar, "Optimal test sequence generation in state based testing using moth flame optimization algorithm," *Journal of Intelligent & Fuzzy Systems*, vol. 35, no. 5, 2018. [Online]. Available: https://doi.org/10.3233/JIFS-169804

[15] B. C. F. Silva, G. Carvalho, and A. Sampaio, "Cpn simulation-based test case generation from controlled natural-language requirements," *Science of Computer Programming*, vol. 181, Jul. 15, 2019. [Online]. Available: https://doi.org/10.1016/j.scico.2019.04.001

[16] D. B. Mishra, R. Mishra, K. N. Das, and A. A. Acharya, "Test case generation and optimization for critical path testing using genetic algorithm," in *Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing*, J. Bansal, K. Das, A. Nagar, K. Deep, and A. Ojha, Eds. Singapore: Springer, 2018, pp. 67–80.

[17] C. V. Jordan, F. Maurer, S. Lowenberg, and J. Provost, "Framework for flexible, adaptive support of test management by means of software agents," *IEEE Robotics and Automation Letters*, vol. 4, no. 3, Jul. 2019. [Online]. Available: https://doi.org/10.1109/LRA.2019.2918486

[18] S. Ashritha and T. Padmashree, "Machine learning for automation software testing challenges, use cases advantages & disadvantages," *International Journal of Innovative Science and Research Technology*, vol. 5, no. 9, Sep. 2020. [Online]. Available: https://ijisrt.com/assets/upload/files/IJISRT20SEP344.pdf

[19] A. Pandey and S. Banerjee, "Test suite optimization using chaotic firefly algorithm in software testing," *International Journal of Applied Metaheuristic Computing*, vol. 8, no. 4, 2017. [Online]. Available: https://doi.org/10.4018/IJAMC.2017100103

[20] I. A. Qureshi and A. Nadeem, "Gui testing techniques: A survey," *International Journal of Future Computer and Communication*, vol. 2, no. 2, Apr. 2013. [Online]. Available: http://www.ijfcc.org/papers/139-B077.pdf

[21] F. Din and K. Z. Zamli, "Pairwise test suite generation using adaptive teaching learning-based optimization algorithm with remedial operator," in *Recent Trends in Data Science and Soft Computing. IRICT 2018. Advances in Intelligent Systems and Computing*, F. Saeed, N. Gazem, F. Mohammed, and A. Busalim, Eds. Switzerland, A. G: Springer Cham., 2019, pp. 187–195.

[22] S. K. Harikarthik, V. Palanisamy, and P. Ramanathan, "Optimal test suite selection in regression testing with test case prioritization using modified ann and whale optimization algorithm," *Cluster Computing*, vol. 21, Nov. 30, 2017. [Online]. Available: https://doi.org/10.1007/s10586-017-1401-7

[23] H. N. N. Al-Sammarraie and D. N. Jawawi, "Multiple black hole inspired meta-heuristic searching optimization for combinatorial testing," *IEEE Access*, vol. 8, Feb. 03, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.2973696

[24] M. Khari, A. Sinha, E. Herrerra-Viedma, and R. G. Crespo, "On the use of meta-heuristic algorithms for automated test suite generation in software testing," in *Toward Humanoid Robots: The Role of Fuzzy Sets. Studies in Systems, Decision and Control*, C. Kahraman and E. Bolturk, Eds. Switzerland, A. G: Springer Cham., 2021, pp. 149–197.

[25] E. Serna and F. Arango, "Desafíos y estrategias prácticas de los estudios empíricos sobre las técnicas de prueba del software," *Ingeniería y Competitividad*, vol. 13, no. 1, Jun. 14, 2011. [Online]. Available: https://www.redalyc.org/pdf/2913/291323660007.pdf

[26] S. Ruland, M. Lochau, O. Fehse, and A. Schurr, "Cpa/tiger-mgp: test-goal set partitioning for efficient multi-goal test-suite generation," *International Journal on Software Tools for Technology Transfer*, vol. 23, Jun. 03, 2020. [Online]. Available: https://doi.org/10.1007/s10009-020-00574-z

[27] T. B. Miranda, M. Dhinya, and K. Sathyamoorthy, "Test-case optimization using genetic and tabu search algorithm in structural testing," *International Journal of Computer Applications Technology and Research*, vol. 4, no. 5, 2015. [Online]. Available: http://ijcatr.com/archives/volume4/issue5/ijcatr04051005.pdf

[28] X. Han, H. Lei, and Y. S. Wang, "Multiple paths test data generation based on particle swarm optimisation," *IET Software*, vol. 11, no. 2, Apr. 2017. [Online]. Available: https://doi.org/10.1049/iet-sen.2016.0066

[29] B. F. Jones, H. H. Sthamer, and D. E, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, Sep. 1996. [Online]. Available: https://doi.org/10.1049/sej.1996.0040

[30] L. Lanzarini and J. P. L. Battaglia, "Dynamic generation of test cases with metaheuristics," *Journal of Computer Science and Technology*, vol. 10, no. 2, Jun. 2010. [Online]. Available: https://www.redalyc.org/pdf/6380/638067314004.pdf

[31] M. D. D. Dapena, A. M. Rojas, D. L. Uribazo, S. V. Marcos, and P. B. F. Oliva, "Modelo para la generación automática de pruebas tempranas basadas en búsquedas," *Computación y Sistemas*, vol. 21, no. 3, 2017. [Online]. Available: https://doi.org/10.13053/CyS-21-3-2716

[32] A. M. Güemes-Esperón, M. D. Delgado-Dapena, and D. Larrosa-Uribazo, "Patrones de implementación para extender la genración de código de pruebas a nuevos lenguajes en gecodp," *Revista Cubana de Ciencias Informáticas*, vol. 15, Oct. 01, 2021. [Online]. Available: https://rcci.uci.cu/?journal=rcci&page=article&op=view&path%5B%5D=2240&path%5B%5D=935

[33] A. M. Güemes-Esperón, M. D. Delgado-Dapena, P. B. Fernández-Oliva, and H. M. Henry-Chibas, "Extensiones de mtest.search para la generación de código de prueba," *Ingeniería Industrial*, vol. 43, no. 1, Feb. 17, 2022. [Online]. Available: http://scielo.sld.cu/scielo.php?pid=S1815-59362022000100049&script=sci_arttext&tlng=pt

[34] P. Fernández, "Modelo para la generación automática de combinaciones de valores de pruebas unitarias," M.S. thesis, Instituto Superior Politécnico José Antonio Echeverría, La Habana, Cuba, 2016.

[35] P. F. Oliva, W. Cantillo-Terrero, M. D. D. Dapena, A. R. Suárez, and C. Y. Márquez, "Generación de combinaciones de valores de pruebas utilizando metaheurística," *Ingeniería Industrial*, vol. 37, no. 2, May-Aug. 2016. [Online]. Available: https://dialnet.unirioja.es/servlet/articulo?codigo=5560665

[36] J. M. Loor-Intriago, M. D. Delgado-Dapena, and P. B. Fernández-Oliva, "Priorización de casos de prueba en entornos de desarrollo ágil," *Ingeniería Industrial*, vol. 41, no. 2, Aug. 01, 2020. [Online]. Available: http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S1815-59362020000200009