An evolutionary algorithm for selection of test cases

Miguel Benito-Parejo
Dept. of Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain
mibeni01@ucm.es

Mercedes G. Merayo
Dept. of Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain
mgmerayo@ucm.es

***Abstract*—Applying tests to an implementation to check its correctness is often expensive and may require an excessive amount of time. Hence, it is necessary to find a relatively small subset of tests able to detect as many errors as possible. In this paper, we study several approaches to choose such subsets of tests based on their capacity to detect faults. These faults are defined as *mutation operators* that are applied to the specification of the systems with the goal of simulating faulty versions called *mutants*. The different methods are evaluated to determine the ones that provide the *best* test suite according to the relevance of the tests, their capacity to detect fault and the number of inputs involved. All the algorithms proposed have been implemented in a tool freely available.**

***Index Terms*—Genetic algorithms, Testing from FSMs, Mutation testing, Selection of test cases**

I. Introduction

Testing is one of the main techniques to validate the correctness of software systems [1]. Usually, the expected behaviour of the systems is given in the form of tests, that are used to determine if the System Under Test (SUT) fulfills the specification. However, the number of tests required to encode all the behaviours of a system may be astronomical. It makes exhaustive testing unfeasible. Therefore, we need to establish a bound on the number of tests that we will apply on the basis of some criteria (e.g. the budget or the available amount of time). Then, we need to select a subset of test cases to detect most faults. The methodology to filter these tests should rely on a measure of how good a test is. In this line, mutation testing [7], [12], [15] is a useful tool. The idea behind mutation testing is that if a test suite distinguishes the SUT from other similar, but faulty, systems, then it might be good at discovering faults. Mutation testing introduces small changes in the SUT, one at a time, by applying *mutation operators*, to generate a set of *mutants*. Intuitively, good test suites are the ones *killing* most of the mutants.

In this paper we propose and evaluate different algorithms to select *good* sets of tests. We assume that we have a formal representation of the SUT (its specification) and that we are provided with a set of mutants and a set of tests (usually huge) that we might apply to the SUT. Our goal is to select a subset of tests up to a certain number of inputs included in it that kills as many mutants as possible. Given a set of tests and a maximum number of inputs to be applied, then the simplest way to obtain the best subset of test cases consists in computing all the possible subsets with up to the established maximum number of inputs, apply them to the set of mutants and choose the subset killing more mutants. Unfortunately, it leads to a combinatorial explosion that disallows us to use this approach. A second option, based on previous work [2], considers a greedy algorithm where we select the *best* tests until we reach the limit of inputs previously established. We determine how *good* is a test on the basis of a value that represent the relevance of the test, the number of mutants it is able to kill and the number of inputs required to kill the mutants. This technique will generally provide good results, both in cost and in faults detection, but it may not always yield the best result. For instance, there could be a combination of individually worse elements that are able to cover more faults. In order to solve this problem, and this is the main contribution of this paper, we have developed a genetic algorithm to find better solutions than the greedy algorithm. The algorithm allows testers to exercise different variants. We have developed a tool that fully implements all the algorithms presented in this paper and their variants. In addition, we also report on the results of the experiments performed to compare the different proposals.

The rest of the paper is structured as follows. In Section II we introduce the main concepts used in the paper. In Section III we enumerate the proposed methods to select the subset of test cases. In Section IV we report on our experiments. In Section V we evaluate the main validity threats. Finally, in Section VI we present our conclusions and some lines for future work.

II. Preliminaries

In this section, we present the basic concepts used in this paper. First, we introduce the formalism we will consider as specification in our framework. Then, we define mutants and tests and, finally, we present the genetic algorithms we propose.

Definition 1: A *Finite State Machine*, in the following FSM, is a tuple $M = (S, I, O, Tr, s_{in})$ where S is a finite set of *states*, I is the set of *input actions*, O is the set of *output actions*, Tr is the set of *transitions* and $s_{in} \in S$ is the *initial state*. A transition belonging to Tr is a tuple (s, s', i, o) where $s, s' \in S$ are the initial and final states of the transition, $i \in I$

Research partially supported by the Spanish MINECO-FEDER (grant number FAME RTI2018-093608-B-C31) and the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314).

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

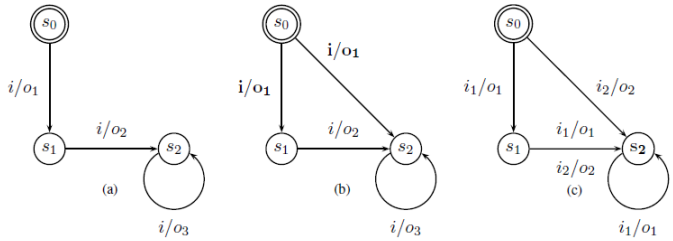


Fig. 1: FSMs with different properties

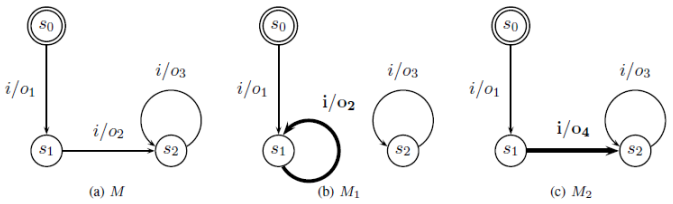


Fig. 2: FSMs with different properties

is the input action and $o \in O$ is the output action. We say that M is *input-enabled* if for each $s \in S$ and input $i \in I$, there exist $s' \in S$ and $o \in O$ such that $(s, s', i, o) \in Tr$. We say that M is *deterministic* if for each $s \in S$ and $i \in I$, there exists at most one transition (s, s', i, o) belonging to Tr.

Throughout this work, we will only consider input-enabled deterministic FSMs, that is, from each state of the machine, all the inputs are accepted and there will be only one possible evolution. This restriction represents the usual behaviour of programs: they are (usually) deterministic and react to any input.

Example 1: Let us consider Figure 1a. The FSM is both input-enabled and deterministic. However, Figure 1b is not deterministic due to the fact that there are two outgoing transitions labelled with the input i from state s_0. The machine depicted in Figure 1c is not input-enabled since s_2 does not have any outgoing transition labelled by the input i_2.

We will use FSMs to represent both specifications and mutants. Despite the simplicity of the FSM formalism, it has a common base with more complex structures to represent black-box systems. Therefore, the framework presented in this paper can be extended whenever we consider systems that receive a sequence of inputs and we can determine the expected output sequence. Next, we introduce the notions of mutant and test.

Definition 2: Let $M = (S, I, O, Tr, s_{in})$ be a FSM. We say that a FSM $M' = (S, I, O, Tr', s_{in})$ is a *mutant* of M if Tr' differs from Tr in only one transition. This *mutation* can be produced by choosing one transition $(s, s', i, o) \in Tr$ and replacing it by either $(s, s', i, o') \in Tr'$, where $o' \in O$ and $o \neq o'$, or $(s, s'', i, o) \in Tr'$, where $s'' \in S$ and $s' \neq s''$. A mutant is called *equivalent* when it is semantically identical to the specification. *Duplicated* mutants are a special form of equivalent mutants. They are equivalent to each other, but not to the specification.

Note that mutants are still deterministic and input-enabled.

Example 2: Consider the FSM given in Figure 2a, being s_0 the initial state. Two possible mutants are shown in Figures 2b and 2c: the first one represents the change of the final state of a transition while the second one represents a change of an output.

Following, we introduce the notion of test.

Definition 3: Let $M = (S, I, O, Tr, s_{in})$ be a FSM. A *test* for M is a tuple $\sigma = (\sigma_{in}, \sigma_{out}, w)$ where $\sigma_{in} \in I^*$ is a sequence of inputs, $\sigma_{out} \in O^*$ is the sequence of outputs that M produces when applying σ_{in} and $w \in \mathbb{Z}$ is the weight associated to the test.

Let $t = (\sigma_{in}, \sigma_{out}, w)$ be a test for M and M' be a mutant of M. We say that M' passes t if the application of σ_{in} to M' produces σ_{out}; otherwise, we say that the M' fails t.

The weight associated with a test represents the *relevance* of such test. In that sense, a higher weight indicates a higher relevance of such test.

Example 3: Let us consider again the mutants depicted in Figures 2 and 2c and the tests $t_1 = (i, o_1, 1)$, $t_2 = (ii, o_1o_2, 2)$ and $t_3 = (iii, o_1o_2o_3, 4)$ for M. We have that M_1 passes t_1 and t_2 and fails t_3 while M_2 passes t_1 and fails t_2 and t_3. The weights of t_1, t_2 and t_3 are 1, 2 and 4, respectively. Intuitively, the test t_3 is 4 times more relevant than the test t_1, and the test t_2 is twice as relevant as t_1.

The main goal of our proposal is to design different techniques to select a *good* subset of test cases from a test suite. Therefore, we need to establish a criterion to determine how much good a specific set of tests is. With this aim we will represent, by means of numerical values, information about each of the tests and their capacity to detect faulty behaviors when they are applied to the mutants. These values will be stored in either a *multiplicative results table*, or an *additive results table*. The rows represent tests and the columns represent mutants. Each element of the table store a value for each test and each mutant whenever the mutant is killed by the test. These values are based on both the number of inputs of the tests that are required to be applied for killing the mutants and the weight associated to the tests. Despite the similarities, both tables are considered in the selection methods that we propose, showing significant differences in the results of the experiments.

Definition 4: Let $M = (S, I, O, Tr, s_{in})$ be a FSM, $\mathcal{T} = \{t_i | 1 \leq i \leq n\}$ be a set of tests for M and $\mathcal{M} = \{M_j | 1 \leq j \leq m\}$ be a set of mutants of M. We define a *multiplicative results table* for $\mathcal{T}$ and $\mathcal{M}$ as a matrix $A \in \mathbb{R}^{nxm}$. Given $t_i = (\sigma_{in}, \sigma_{out}, w) \in \mathcal{T}$ and $M_j \in \mathcal{M}$, if M_j fails t_i then A_{ij} is the length of the shortest prefix of σ_{in} that kills M_j multiplied by $\frac{1}{w}$. In the case that M_j passes t_i, A_{ij} is infinity.

Definition 5: Let $M = (S, I, O, Tr, s_{in})$ be a FSM, $\mathcal{T} = \{t_i | 1 \leq i \leq n\}$ be a set of tests for M and $\mathcal{M} = \{M_j | 1 \leq j \leq m\}$ be a set of mutants of M. We define

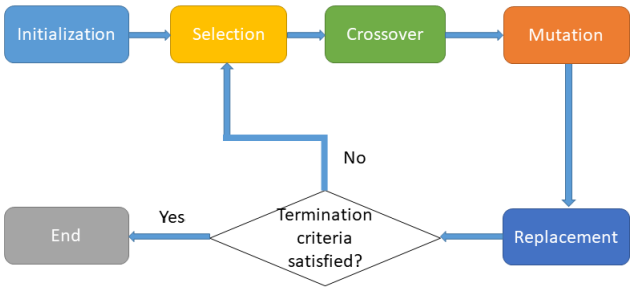


Fig. 3: GA flowchart

an *additive results table* for $\mathcal{T}$ and $\mathcal{M}$ as a matrix $A \in \mathbb{R}^{nxm}$. Given $t_i = (\sigma_{in}, \sigma_{out}, w) \in \mathcal{T}$ and $M_j \in \mathcal{M}$, if M_j fails t_i then A_{ij} is the length of the shortest prefix of σ_{in} that kills M_j minus w. In the case that M_j passes t_i, A_{ij} is infinity.

The information stored in the result tables is used in two different *fitness functions* based on a *multiplicative heuristic* and an *additive heuristic*, respectively. In addition to the results tables, the number of mutants killed by the set of tests is also taking into account, penalizing those ones that kill less mutants overall.

Definition 6: Let M be a FSM, $\mathcal{T} = \{t_i | 1 \leq i \leq n\}$ be a set of tests for M, $\mathcal{S} = \{t'_i | 1 \leq i \leq n'\}$ be a subset of $\mathcal{T}$, $\mathcal{M} = \{M_i | 1 \leq i \leq m\}$ be a set of mutants of M, A be the multiplicative (respectively additive) results table for $\mathcal{T}$ and $\mathcal{M}$, and B be the multiplicative (respectively additive) results table for $\mathcal{S}$ and $\mathcal{M}$. We define the multiplicative (respectively additive) fitness function of $\mathcal{S}$ for $\mathcal{M}$ as:

$$f(\mathcal{S}, \mathcal{M}) = \sum_{k=1}^{m} \min\left(\alpha(M_k, \mathcal{S}), P\right)$$

where $\alpha(M_k, \mathcal{S}) = \min(B_{ik} : 1 \leq i \leq n')$, r is the penalty ratio and $P = r * \max(A_{ij} : 1 \leq i \leq n, 1 \leq j \leq m \wedge A_{ij} \neq \infty)$ is the penalty value.

The fitness function does punish the sets of tests that do not kill all the mutants, as a penalty is added for each alive mutant. Therefore, the more mutants a subset kills, the lower the score will be. Moreover, a lesser number of inputs required to kill the mutants will also reduce this value, leading us to a minimization problem (a lower value of fitness denotes a better set of tests).

In our framework we consider *Genetic Algorithms* (GA) [10], [21], a heuristic optimization technique, which it is based on evolutionary processes in nature. GAs and other optimization algorithms have been used in software testing [9], [11], [16], [17], where finding the optimal solution is, in practice, unfeasible. Nevertheless, good enough approximations with *low* complexity are usually accepted by users for such problems. Generally, a GA consists of a group of individuals (population of genomes), each representing a potential solution to the problem in hand. An initial population with such individuals is usually selected at random. Then, a parent selection process is used to pick a few of these individuals. New offspring individuals are produced using *crossover*, keeping some of the characteristics of their parents, and *mutation*, which introduces some new genetic material. The quality of each individual is measured by a fitness function, defined for the particular search problem. Crossover exchanges information between two or more individuals. The mutation process randomly modifies offspring individuals. The population is iteratively recombined and mutated to evolve successive populations, known as generations. When the termination criterion specified is satisfied, the algorithm terminates. A flowchart for a simple GA is presented in Fig. 3.

III. The Proposal: Selection Methods

In this section we propose different methods to address the problem of obtaining a *good* subset of tests. All the approaches use the fitness functions previously introduce to determine the *quality* of a subset of tests.

A. Global search

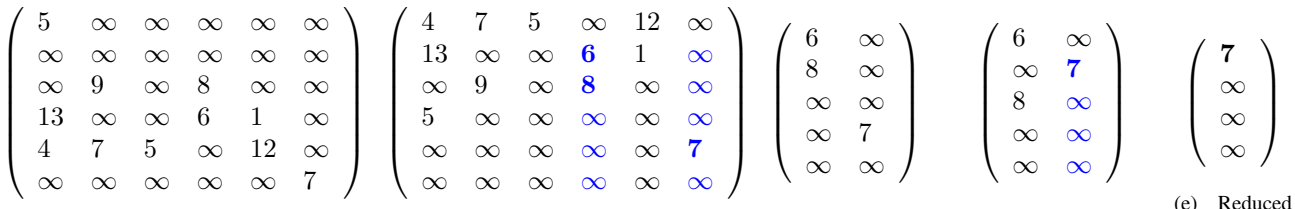
The global search approach looks through all the possible combinations of the initial set of tests having less inputs than a given bound. This approach always provides the best solution because it explores all the possible subsets. Therefore, we consider it to compare the results with the forthcoming algorithms. Despite obtaining the best result, the main reason this technique is not usually applied to a general problem is the combinatorial explosion it suffers from non-trivial systems. Since so many possibilities arise from the combinations, only really small systems are subject to this technique. In fact, we were able to compute it only for the smallest of our experiments.

B. Greedy algorithms

We have designed two different greedy methods to show that some intuitive approaches are not always as good as they seem.

Our first greedy algorithm builds a set of selected tests by choosing the test case with the highest weight from the original test suite. If several test cases have the same weight, the test that kills more mutants is selected. In the case that many tests kill the same number of mutants the test that requires the lowest number of inputs to kill them is selected. We iterate this process avoiding to exceed the bound of inputs established initially. Since this method does not need the fitness function to generate the subset of tests, the approach will be used as a control method to be compared with other approaches. However, the mere weight information does not represent how good a test is, and the experiments do not show any remarkable result. Such fact moves the attention towards other methods that require the fitness functions to operate.

The second greedy algorithm that we propose is based on the results tables. It is worth pointing out that the solution depends on the kind of results table we consider. For the same set of tests and mutants, the multiplicative results table and the additive results table are different. For example, let us consider the test t_i whose associated weight is 3. If the number of inputs



(a) Original matrix (b) Ordered matrix (c) Reduced matrix (d) Ordered matrix (e) Reduced matrix

Fig. 4: Matrix simplification

of t required to kill the mutant M_j is 9, then the value that will be stored in the position A_{ij} of the multiplicative (respectively additive) results table will be 3 (respectively 6). Essentially, the algorithm sorts the matrix in a way such that the first test is the best, the second is the second best and so on. The test corresponding to the first row of the matrix is included into the set that we are generating. Then, the row corresponding to that test and all the columns of the mutants that it is able to kill are removed from the matrix. After the reduction of the matrix, the process is iterated until either all the mutants are killed by the set of selected tests or the maximum number of inputs established is reached.

Example 4: Figure 4a depicts a matrix that represents a results table. In this case, 6 tests cases and 6 mutants are considered. Once the matrix is ordered, see Figure 4b, the first row corresponds to the best test, which is selected to be included in the new test suite. Following, the first row and the columns corresponding to the mutants killed by the test case selected (1, 2, 3 and 5) are removed, resulting in the reduced matrix depicted in Figure 4c. This process is iterated until all the columns are removed, that is, the set of selected tests are able to kill all the mutants.

A good property of this algorithm is that its space complexity is in low polynomial order over the size of the table, that reduces its size after each iteration. In terms of time, this algorithm offers the best solution for computing the solution. This greedy method shows great results and will help to bound the number of generations and the global cost of our next algorithm.

C. Genetic algorithm

As we know, GAs excel when we seek for a good enough approximation of the solution of problems whose optimal solution needs a combinatorial approach to compute all the potential candidates. This is the case of our problem and its solution, as discussed in Section III-A. Therefore, a genetic algorithm is a sensible approach to compete with our greedy algorithms, in particular, taking into account that our second greedy algorithm computes relatively good solutions spending a small amount of time. Next, we explain the specifics of our GA from our previous work [3].

The population: In our implementation, an individual only has one chromosome. Our GA has a population of initial test subsets that will evolve to generate better subsets until either we reach the optimum solution or an established maximum number of iterations.

The user must provide some parameters associated to the different phases of the GAs execution. Next, we briefly describe them.

Initialization method: We have designed an *incremental initialization*. It provides a variety of chromosomes, each of them with a different amount of tests and inputs, which generates a high level of diversity. Such initialization follows the idea of minimizing the number of inputs to apply. As some chromosomes may have too few inputs and others too many, the execution of the algorithm will mix them at some point and improve the general result.

Selection of population methods: The transition from a generation to the next one has to ensure that the representatives of the foremost individuals have to be selected. The idea is to reward the best ones with more appearances and penalise the worst ones even with no appearances at all. Our tool allows the user to choose alternative selection models: *Linear rank*, *remains*, *roulette wheel*, *tournament*, *truncation* and *stochastic universal*. Since these selection models are quite common [10], [18], we do not explain the details about how each of them is applied to the population.

Crossover methods: Concerning crossover we have designed two methods. The first one is the *standard crossover* (Fig. 5) that takes two chromosomes of the population and chooses a random point at each of their respective list of tests. Then, the left part of both lists are preserved and the right parts are exchanged. If such modification generates a test suite with more inputs than the established bound, then the last tests are discarded to fulfill this limitation. The second one is the *continuous crossover* (Fig. 6) that takes two chromosomes and exchanges tests of both lists at randomly positions.

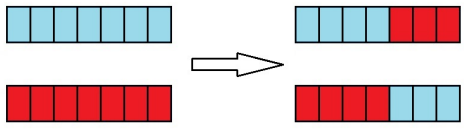


Fig. 5: Standard Crossover

Mutation of population methods: As the initial seeds for the population might not be complete, it is sensible to refresh

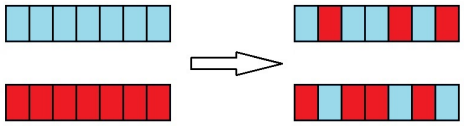


Fig. 6: Continuous Crossover

the population with some slight changes that could renew some stale state. In our case, we have designed two different techniques. The first one considers *adding mutation* (Fig. 7) that is oriented to not-complete subsets. In these cases, it is possible to introduce an extra test to an individual without exceeding the bound of inputs. As a consequence of the crossover step, it is possible that some of them have to be discarded. This mutation method complements the possible loss of tests. The second method bases on *replacing mutation* (Fig. 8). This method changes one test of an individual by another one coming from the original test suite. This technique will include some slight changes to specific individuals that might either increase or decrease the relevance of a subset of tests into the population.

Fig. 7: Adding Mutation

Fig. 8: Replacing Mutation

Replacement of population methods: The last step of the genetic algorithm consists in replacing the population by the new one. Again, we have two possibilities. On one hand, we have the trivial option, that substitutes the last population by the new one, even if it is worse. In this way, less operations are performed at this stage and, as a result, the execution will be faster. On the other hand, a high percentage of the new generation can replace the previous one. This approach called *elitist replacement* (Fig. 9) will allow the population to keep the best partial solution. As a counterpart, more calculations have to be made and that cost might decelerate the program.

In Figure 10, we show the graphs of an experiment concerning how fitness varies along generations. The results are as expected. In short, there is a relatively big variance related to the worse individual of each generation, that is, the highest value of fitness. This variance is smaller for *average fitness*, *generational best* stabilizes (although there are small variations), while *absolute best* quickly converges.

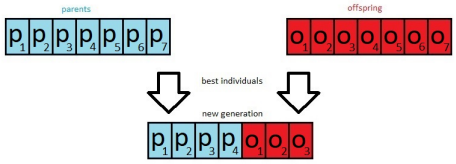


Fig. 9: Elitist Replacement

D. Random selection

Our last approach is used as a control method to determine the improvement obtained from the application of the proposed approaches over a *random selection* of the tests. The idea is that if we perform a random selection of tests several times, the mean result will represent the average values expected from the whole test suite. As such, this method relies on a number of random choices of tests until we reach the bound on the inputs we can apply. This method also takes information of both results tables so that each heuristic has its own comparison criterion.

IV. Experiments

In this section we report on the experiments that we have performed and the results obtained. First, we analyze the closeness of the solutions provided by the proposed algorithms to the best solution given by the full search algorithm. This analysis aims to answer the following research question:

RQ1: How good are the solutions produced by the different proposed algorithms?

Second, we evaluate the time that the different approaches need to compute the solutions. The goal of this analysis is to answer the following research question:

RQ2: What is the relative impact of the size of the specifications, sets of tests and number of inputs allowed on algorithms scalability?

A. Description of the experiments

We divided our experiments in two phases in order to solve the research questions. Nevertheless, the experiments were held over the same specifications and sets of mutants. We worked with 50 specifications comprised between 10 and 50 states and between 3 and 10 different inputs and outputs. Concerning the mutants, for each specification we have a set of mutants according to the size of the specification. For the smallest specification, with 10 states, 3 inputs and 4 outputs, we had 63 mutants, whereas for the biggest one, with 50 states, 10 inputs and 8 outputs we had 2514 mutants. Before the experiments, we removed both the duplicate and the equivalent mutants.

We have applied all the proposed algorithms, considering three categories. The first category includes the full search algorithm, providing the best subset of tests, the first greedy algorithm, focusing on the weights of the tests, and the random selection, representing the average values of a subset of tests.

TABLE I: Comparison of fitness values

Fitness funtion	Best GA	Avg GA	Worst GA	Greedy Weights	Greedy Table	Random
A	0.3%	3.2%	4.8%	25.5%	9.6%	28%
M	3.1%	6.7%	10.4%	24.8%	9.8%	31.6%

The second category considers the second greedy algorithm proposed. This approach is an alternative to the GAs, so it will be used to compare the execution times. Finally, the third category deals with all the versions (48) of the genetic algorithm proposed, that is, considering the different heuristics, selection, crossover, mutation and replacement methods. In these experiments, we used the default values for the parameters, that is, a population of 100 individuals, 3 players with a ratio of 0.8 for the tournament selection, 0.125 ratio for the truncation selection, a probability of 0.6 for the crossover methods, a probability of 0.08 for the mutation methods and a ratio of 0.02 for the elitist replacement. Finally, it is worth adding that the execution of the algorithms that include a random component were repeated. Then, we calculated the average of both the time execution and the fitness values obtained.

The goal of the first group of experiments was to compare the results produced by the proposed algorithms with the "best solution" provided by the full search approach and the random selection. In this case, we used small sets of test cases and low upper bounds for the number of inputs accepted in the solution. The reason for establishing these limits is that with greater values we were not able to complete the execution of the full search algorithm due to its exponential growth. The initial sets of tests have between 10 and 50 test cases, having each test case between 10 and 80 inputs. For each test suite, the bound on the number of inputs allowed in the solution ranges from 40 to 150.

In the second group of experiments we increased the size of the initial sets of test cases as well as the bound on the inputs of the sets of tests generated by the algorithms. Specifically, the initial sets of tests were comprising between 100 to 250 test cases with a number of inputs varying from 80 to 150. We established the upper bound of the number of inputs that sum all the tests included in a solution between 150 to 1000. The goal of these experiments was to evaluate the scalability, in terms of execution time, of each of the proposed algorithms. Therefore, due to the fact that the execution time of the full search algorithm for these sets of tests is unavoidably unpractical, we did not consider it in this phase. We only analyse the performance of the greedy algorithm, the random search and the GAs.

The workbench that we have used for performing the experiments has an Intel i5-8250U processor of 3.4 GHz and 8GB of RAM. Let us note that the implementation of the approach and some additional examples are freely available at https://github.com/miguelbpsg/WCCI20.

B. Evaluation

The first group of experiments allowed us to obtain the best solution provided by the full search algorithm in order to be used for determining properly how good other algorithms are producing solutions.

Table I shows the deviation of the fitness of the solutions provided by each of the proposed algorithms from the best solution (full search algorithm). The information of the first and second row (A and M), corresponds to the use of the additive and multiplicative fitness functions, respectively. The columns correspond to the different algorithms that have been analyzed, that is, the random search, the greedy algorithms and the GAs. In the case of the GAs, we have only include the values associated to the algorithms that present the best, average and worst results with respect to the full search approach.

The first greedy algorithm (greedy weights) presents a scatter behavior. This is due to the fact that it bases on the weights associated to the tests, which do not ensure their effectiveness. However, it is worth noting that it often beats the random solution, but not consistently neither in terms of frequency nor in terms of improvement. The solution provided by the second greedy algorithm (greedy table) is substantially better than the one obtained from random selection, and the set of tests selected is close to the solution produced by the full search approach $\backsimeq 9\%$. This suggests that the greedy algorithm is worth to being applied, since it produces a good approximation to the best solution to the problem. Mostly, considering the improvement with respect to the random algorithm. All the GA algorithms yielded closest solutions to the best one. Table I shows that all the versions of the GAs find the closest solutions to the best one. It is worth noting that the results corresponding to the additive an multiplicative heuristics, in the case of the GAs, present solutions of different quality due to the fact that the multiplicative fitness applies a product which may highly enlarger the distances, and also the penalties. Then, we can conclude from the experiments, that the use of the additive version of the GAs provides better solutions. Overall, the experiments show that the GAs can adequately compete with the rest of the algorithms considered, providing a better solution with the proper choice of parameters. There is not a *best* combination of parameters in terms of obtaining the best solution. Only the combination of Tournament selection, Standard crossover, Adding mutation and Elitist replacement

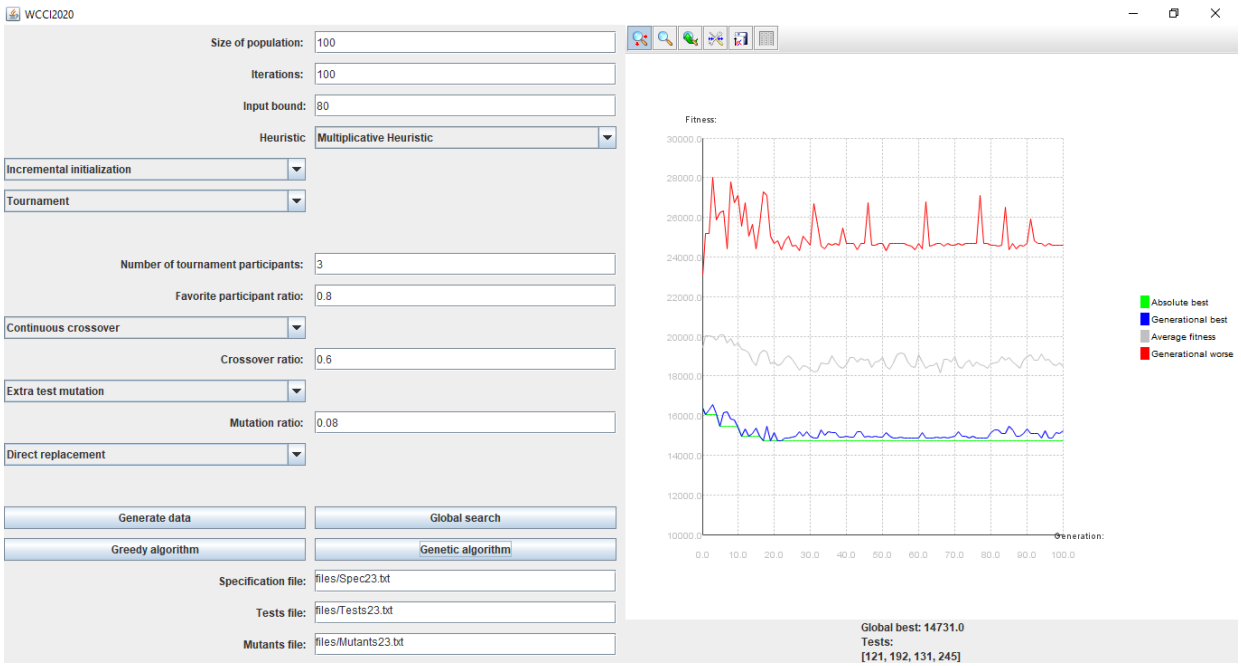


Fig. 10: GUI of our tool

provided the best solution in the 5% of the experiments, barely making a difference from a uniform distribution. However, we cannot assure such configuration will always yield the best solution among all possible GAs.

In general terms, the GAs got to beat the rest of algorithms with a substantial margin. Nevertheless, the greedy algorithm also performs an acceptable approximation.

Altogether, we can answer *RQ1* positively: The algorithms proposed, in particular the GAs, are very close to the best solution.

The goal of the second group of experiments is to determine the scalability in terms of time of our approaches. Since we cannot perform the full search with big sets of tests, we will consider the random search algorithm, to determine how much extra time is required for generating the solutions.

Table II shows the results of the experiments. In this case we focus on the second greedy algorithm and the GAs. The greedy algorithm and the random selection were the fastest methods. Nevertheless, the greedy algorithm requires about 60% more time than the random algorithm to yield a solution.

Regarding the GAs, there is a great difference between the fastest and the slowest, due to the number of sorting operations required in some of the versions of this proposal. In particular, the GAs that use the elitist replacement and truncation selection operators show the worst execution time, because these parameters slow the execution at the cost of partially better results, although the required time grows up to more than 300% extra time.

In this case, the results obtained from both heuristics (additive and multiplicative) are very similar, since the execution time of the algorithms is not affected by the information of each results table.

Overall, we can only answer *RQ2* negatively. In terms of time, the GAs are slower than all the other approaches with the exception of the full search algorithm.

V. Threats to Validity

In this section we briefly discuss the main possible threats to the validity of our experiments and its results.

First, the main threats to *internal validity*, which considers uncontrolled factors that could alter the obtained results, are the faults introduced in the development of the experiments, since the results could be compromised. To prevent the impact of the internal validity threat, we performed several tests on our implementation. We focused in unit testing to check that the individual methods worked as expected, and integration testing to check that the whole system is adequately connected. We also repeated the experiments of the genetic algorithms to reduce the impact that their random behavior could generate.

Regarding the *external validity* threats, which include conditions that allow us to extend our experiments to other scenarios, the most important one corresponds to the selected specifications, mutants and tests we used. Their structure and size are unknown and cannot be generalized. Thus, we developed several methods trying to widen the choices a specification may have to solve the problem, and we compared the features of each of the methods.

Finally, the threats to *construct validity*, which concern the *reality* of our experiments, that is, whether our experiments are representative enough for real machines, would arise when a large specification with a large amount of tests and mutants would have to be checked. Despite being an important threat,

TABLE II: Comparison time executions

Fitness function	Fastest GA	Average GA	Slowest GA	Greedy Table
A	90%	135%	300%	60%
M	85%	135%	335%	55%

our implementation is able to support considerably large cases with an acceptable workbench. As such, we consider that the threat is not as disturbing as long as the experiments are performed on an adequate system.

VI. Conclusions and future work

In this paper we present a solution to the problem of obtaining good subsets of tests out of an initial test suite. Usually, time and budget subject to testing is usually limited, and only few test cases might end up being applied. In our framework, the tester can order the tests in terms of relevance, and control the bound of tests to apply taking into account a maximum number of inputs. We provide a framework consisting of different algorithms (greedy and GAs) to generate good subsets of tests. The results showed that our GAs find a much better solution than both random selection and greedy algorithms. Also, it reveals that the difference between the GAs and the full search algorithms was rather small. However, in terms of time, the GAs are slower than all the other approaches with the exception of the full search.

As future work, we aim at dealing with different formalisms in the proposed framework, such as fuzzy systems [4], [5]. Another line of work is to support a distributed architecture [13], [14] considering the Petri Nets formalism. Such distributed architecture allows different users to interact with the same system without being connected among them. Other line of work will consider the use of our approach with other metaheuristics [19], [20]. In addition, we would like to use current approaches to mutation testing [6], [8] to efficiently generate and process big amount of mutants representing different faults.

From another perspective, we are willing to consider different approaches to deal with budget and time limitations, such as the number of tests applied, or a combination of the number of tests and the number of inputs. Another interesting option could be to add weight to mutants, indicating the relevance of the errors they represent.

Finally, we would like to improve the usability of the tool to allow the user to provide the parameters in a more user friendly interface. Furthermore, we will improve the information provided by the tool about the results of the application of the different algorithms, including graphs that help to understand them.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

[2] C. Andrés, M. G. Merayo, and M. Núñez. Formal passive testing of timed systems: Theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012.

[3] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.

[4] I. Calvo, M. G. Merayo, and M. Núñez. A methodology to analyze heart data using fuzzy automata. *Journal of Intelligent & Fuzzy Systems (to appear)*, 2019.

[5] I. Calvo, M. G. Merayo, M. Núñez, and F. Palomo-Lozano. Conformance relations for fuzzy automata. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 753–765. Springer, 2019.

[6] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.

[7] P. Delgado-Pérez, I. Medina-Bulo, and J. J. Domínguez-Jiménez. Mutation testing. In *Encyclopedia of Information Science and Technology*, pages 7212–7221. IGI Global, 3rd edition, 2014.

[8] P. Delgado-Pérez, Louis M. Rose, and I. Medina-Bulo. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2):823–859, 2019.

[9] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *11th Int. Conf. on Artificial Neural Networks, IWANN'11, LNCS 6692*, pages 396–403. Springer, 2011.

[10] D.E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.

[11] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[12] R. M. Hierons, M. G. Merayo, and M. Núñez. Mutation testing. In Phillip A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 594–602. Taylor & Francis, 2010.

[13] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.

[14] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.

[15] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[16] B. F. Jones, D. E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[17] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[18] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 3rd, revised and extended edition, 1996.

[19] V. D. Nguyen, H. B. Truong, M. G. Merayo, and N. T. Nguyen. Toward evaluating the level of crowd wisdom using interval estimates. *Journal of Intelligent and Fuzzy Systems*, 37(6):7279–7289, 2019.

[20] Van Du Nguyen, Hai Bang Truong, Mercedes G. Merayo, and Ngoc Thanh Nguyen. An overview on consensus-based approaches to processing collective inconsistency and knowledge integration. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, 9(4), 2019.

[21] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27:17–27, 1994.