An Evolutionary Algorithm and a Clustering Technique to Select Good Subsets of Test for Finite State Machines

Miguel Benito-Parejo(✉), Manuel Méndez, and Mercedes G. Merayo

Design and Testing of Reliable Systems Research Group,
Universidad Complutense de Madrid, Madrid, Spain
{mibeni01,manumend}@ucm.es, mgmerayo@fdi.ucm.es

Abstract. Testing is the technique most widely used to validate the correct behaviour of systems. Essentially, a test consists of applying an input to the system and decide whether it returns the expected output. Unfortunately, budget and temporal constraints limit the amount of testing that can be applied to the system. Therefore, a *good* selection of tests will reduce the resources devoted to testing while keeping an effective validation process. In this paper, we tackle this problem by using mutation testing, which effectively simulates the possible faults that the system under test may have and suggest which tests are best in finding potential faults. In order to perform test selection, we use a multi-objective genetic algorithm that focuses on two targets: minimising the number of inputs the test suite has to perform and maximising the mutation score. We have performed several experiments and exhaustively compared our proposal with a Machine Learning method, specifically clustering, which groups the tests into classes, from which we select the most suitable test to be applied.

Keywords: Mutation Testing · Genetic Algorithms · Clustering · Test case selection

1 Introduction

Software testing is the most widely used technique to validate the correctness of a system, i.e. to check it for errors. One of the main drawbacks of software testing is that the number of tests needed to check all the system behaviours can be infinite. Therefore, there is a need to select a (finite) number of tests that can detect most of the errors based on a certain criterion. In order to obtain this subset, it is necessary to establish a metric that allows us to determine the quality of each test. The final goal is to filter out and keep only the *best*

This work has been supported by the Spanish MINECO/FEDER project AwESOMe (PID2021-122215NB-C31).

N.-T. Nguyen et al. (Eds.): ICCCI 2024, CCIS 2166, pp. 168–182, 2024.
https://doi.org/10.1007/978-3-031-70259-4_13

ones in the final subset. Therefore, we have to rely on test case prioritisation techniques, which allow us to sort tests based on a desirable property, such as their ability to detect errors [4]. In this line, *mutation testing* [8,9,12,13,22,23] has shown its effectiveness to evaluate the quality of a set of tests. For this purpose, systems similar to the system under test (SUT) are created by applying small modifications to the SUT, such as removing code fragments, swapping arithmetic operators or swapping relational operators. By applying the tests to the mutants and to the original system, it is possible to determine those with the highest error detection capability, that is, those tests that reveal the most differences in the behaviour of the mutants compared to that of the original system. The idea is that if a set of tests distinguishes the system under test from other systems similar to it but with faults, then it can be a *good* set for determining errors in that system. In this way, prioritisation of test cases can be carried out based on this error detection capability. Another aspect to take into account during the selection of the test cases is their length, that is, the number of inputs to be applied for their execution. The shorter the length of the tests, the faster their execution.

In this paper, we propose two techniques for selecting a subset of test from an initial set of test cases that optimise both criteria, their error detection capacity and their diversity when detecting errors, as well as the length of the test cases. The first one corresponds to a genetic algorithm, that applies an advanced multi-objective optimisation algorithm [5]. The second technique that we propose applies a clustering method. Clustering is a traditional unsupervised grouping technique that has been utilised across various fields [20,26]. However, this technique is not widely applied to test selection tasks. Some works in this area examine the effectiveness of test case prioritisation techniques through a requirements-based clustering approach that integrates traditional code analysis information [1], or apply spectral clustering to mutant reduction [25]. This work is an extension of a previous proposal for test case selection in the field of finite state machines [2].

The main objective of this work is to apply and combine mutation testing techniques with genetic algorithms and clustering for the design and implementation of a test case selection framework. This framework should guide the minimisation of the number of inputs to be applied while maximising the number of different mutants that the test cases kill. To this end, we will follow the next steps:

– We will generate a set of *mutants* by injecting small changes in the original SUT, with the goal of simulating faults. We will use the *mutation score* for determining the quality of the tests. This metric is based on the number of *mutants* a set of test cases is able to detect. The more mutants the test cases kill, the better this subset of test cases will be. We will use the *mutation score* together with the number of inputs to be applied by the subset of test cases to determine the quality of this subset.

- We will design a genetic algorithm in which we apply the multi-objective algorithm NSGA-II [5] during the replacement phase. This would allow us to order the partial solutions in each generation based on our criteria.
- We will apply a clustering method to group the tests depending on which mutants they kill, with the goal of selecting the shortest test from each cluster.
- Finally, we compare the effectiveness of these methods to obtain the best subset of test cases.

The rest of the paper is organised as follows. In Sect. 2 we present the framework and the main problem to solve with the formalism and objectives to optimise as well as the methods that we use to solve this problem. In Sect. 3 we present the results of our experiments. Finally, in Sect. 4 we present our conclusions and some lines for future work.

2 Description of the Problem and Methods

In this section, we present our framework. We consider a formalism able to represent systems and the mutants that we will deal with. We also introduce tests and how they interact with systems. Finally, we include an introduction to genetic algorithms, multi-objective optimisation and clustering.

2.1 The Formalism

In order to represent the SUT and mutants, we use FSMs. This formalism is simple enough to represent the basic behaviour of systems, but our work can be adapted to deal with other more complex structures or formalisms. This allows a more progressive extension of this work, while maintaining the clarity of the execution and the performance of the experiments.

Definition 1. *A* Finite State Machine *(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,* $i \in I$ *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*.*

In this work, we consider that our system will be input-enabled and deterministic, representing the usual behaviour of most systems, where all the states of the FSM can receive any possible input and perform only one change of state (always the same).

Example 1. Let us consider Fig. 1 (left). This FSM is both input-enabled and deterministic. The FSM shown in Fig. 1 (center) is not deterministic due to the

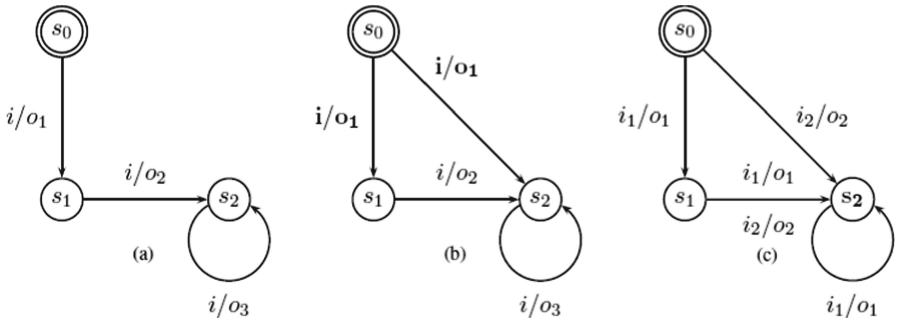

Fig. 1. FSMs with different properties

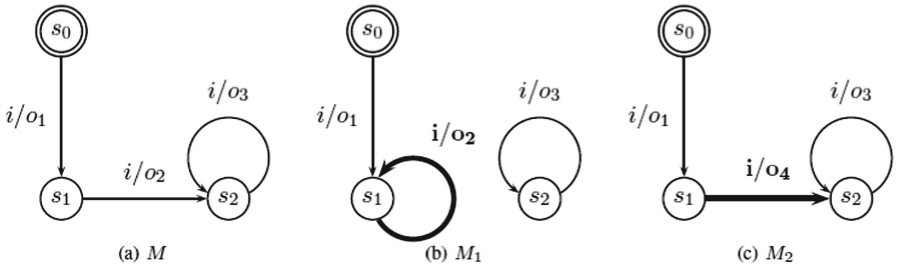

Fig. 2. System and two mutations

fact that there are two outgoing transitions labelled with the input i from state s_0. The machine depicted in Fig. 1 (right) is not input-enabled since s_2 does not have any outgoing transition labelled by the input i_2.

Definition 2. *Let* $M = (S, I, O, Tr, s_{in})$ *be a FSM, 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 original FSM, that is, for each sequence of inputs both produce the same sequence of outputs.* Duplicated *mutants are a special form of equivalent mutants. They are equivalent to each other, but not to the original FSM.*

It is important to note that the mutations we are performing do not change the existence nor the uniqueness of the transitions associated to each input. As such, the mutants that we obtain are deterministic and input-enabled.

Example 2. Consider the FSM given in Fig. 2 (left), being s_0 the initial state. Two possible mutants are shown in Fig. 2 (center) and 2 (right): the first one represents the change of the final state of a transition while the second one represents a change of an output.

Table 1. Tests killing mutants

	m_1	m_2	m_3	m_4
t_1	1	1	1	1
t_2	0	1	1	0
t_3	0	0	1	1
t_4	1	0	1	0

Table 2. Differences in mutation score

test sets	*killed mutants*	*ms*
$T_{S_1} = \{t_4\}$	m_1, m_3	0.5
$T_{S_2} = \{t_3, t_4\}$	m_1, m_3, m_4	0.75
$T_{S_3} = \{t_1\}$	m_1, m_2, m_3, m_4	1
$T_{S_4} = \{t_2, t_3, t_4\}$	m_1, m_2, m_3, m_4	1

Definition 3. *Let $M = (S, I, O, Tr, s_{in})$ be a FSM. A* test *for M is a tuple $\sigma = (\sigma_{in}, \sigma_{out})$ 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}.*

Let $t = (\sigma_{in}, \sigma_{out})$ 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 M' fails t.

Example 3. Let us consider again the mutants depicted in Fig. 2 and the tests $t_1 = (i, o_1)$, $t_2 = (ii, o_1o_2)$ and $t_3 = (iii, o_1o_2o_3)$ 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.

Note that the set of input actions is fixed for a given FSM. In this work we want to minimise the number of inputs that the tests apply to the SUT. In other words, we want to reduce the length of the sequences of inputs (respectively outputs) of the tests that we select to be performed on the SUT. This is because the execution of each input of the tests is associated to a cost, either in terms of budget or time.

The classical approach to assess the goodness of a test suite is to compute its mutation score, which indicates the ratio of killed mutants.

Definition 4. *We define the mutation score of the set of tests $\mathcal{T}$ for the set of mutants $\mathcal{M}$ as*

$$ms(\mathcal{M}, \mathcal{T}) = \frac{\#\text{mutants killed}}{\#\text{nonequivalent mutants}}$$

It is important to note that groups of duplicated mutants are only considered once, or otherwise this metric would be biased towards tests that kill many duplicated mutants.

Example 4. Let us consider Table 1, where rows represent different tests and columns different mutants. Each cell indicates whether the test kills the mutant (1) or not (0). In this case, we have that tests t_2, t_3 and t_4 all kill two mutants. They all detect m_3 and then individually kill m_2, m_4 and m_1 respectively. Also, we have that test t_1 kills all four mutants.

Now, let us consider Table 2, which uses different subsets of tests from Table 1. We observe T_{S_1} only kills two mutants, having $ms = 0.5$. If we look at T_{S_4}, we are able to kill all mutants, having $ms = 1$, but several tests are required to achieve that goal. We can also achieve $ms = 1$ with T_{S_3}, that only needs one test to detect all the mutants.

2.2 Multi-objective Genetic Algorithm

We address the problem of selecting a quality subset of tests focused on optimising two objectives: the total number of inputs to be applied and the mutation score of the subset corresponding to a set of mutants. Since these two objectives are interrelated in the sense that fewer inputs are likely to reduce the mutation score, we are faced with a classic multi-objective optimisation problem.

Genetic Algorithms (GAs) are stochastic methods for heuristic adaptive search that are based on the natural selection and in the genetic evolution in nature [6,11,24]. These methods are generally used when looking for a good enough approximation for problems where the optimal solution is unreachable or when it is too expensive to obtain it, generally with an exponential cost. The general idea is to have a population of individuals or chromosomes that evolve over a number of iterations, making changes that eventually yield a solution close to the optimum [16].

Figure 3 presents the flowchart of a GA: Initially, an initialisation of the population takes place, usually yielding a diverse spectrum of initial solutions that could easily find a good approximation. Each iteration of a GA is called *generation*, and consists of a *selection* of individuals, usually representing the best solutions so far, which are *crossed* with each other producing an offspring where some of the new individuals are *mutated* for further exploration of the solutions. Finally, during the replacement phase, the parents and offspring compete to continue into the next generation.

Usually, the use of a fitness function is required to measure how good a solution is. However, we are aiming to optimise two objectives at the same time, and it is often the case that two solutions are incomparable. That is, one is better on one objective but worse on the other. This introduces the concepts of domination and Pareto front, defined as follows.

Definition 5. *We say that a solution x* dominates *another one y, denoted by $x \prec y$, if no objective of the latter improves with respect to the former and at least one objective of the former improves compared to the latter. A Pareto front, or* front *is a set of solutions that are not dominated among them.*

Although GAs have previously used for software testing [2,3,7,10,15,19], in our approach, we use a multi-objective genetic algorithm, the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [5], that has not been considered yet for this specific problem. The idea is not to use a fitness function that guides the execution of the GA, indicating how good each solution is, but to compare and classify the solutions in the different *fronts* that select those that are non dominated [5,21]. Note that, in general, NSGA-II does not work well on many-objective problems. This is due to the fact that dominated solutions are rare when the number of objectives grows, where the front has a higher dimension and the dominated region is smaller. However, our previous experience with NSGA-II in two-objective problems has shown very good results because given a point, half of the solution space is dominated or being dominated by such point. The other half are incomparable points, as it can be seen in Fig. 4, where

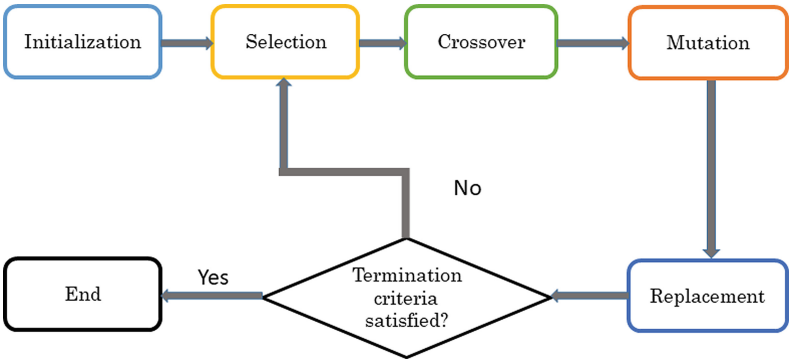

Fig. 3. Flowchart for a standard GA

region 1 is dominated by x, region 3 dominates x and the regions 2 and 4 are not dominated and do not dominate x.

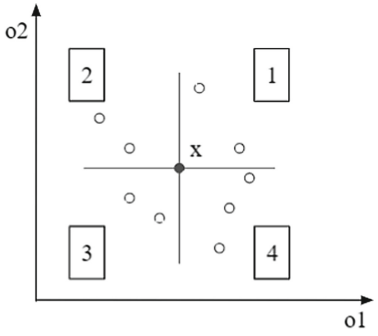

Fig. 4. Dominance regions with 2 objectives

It is worth noting that the mutation score ms is a maximisation objective. Therefore, in order to adequate it for the standard use of the NSGA-II. Instead of using such value we will minimise $1 - ms$, as ms takes values from 0 to 1.

Next, we will describe the specific structure of the phases of the GA and the methods we propose to solve the considered problem.

Population. The chromosomes or individuals represent a possible solution to the problem to solve. In our case, such problem is to obtain a good subset of tests. Thus, each chromosome is a subset of the initial set of tests, which we have implemented as a list.

Initialisation. We have used *random initialisation*, as there is not any special criterion to bias the initialisation, and a diverse population might help to achieve a better solution faster. The number of tests and inputs each solution

may initially have should vary, having chromosomes that specialise in each of the objectives.

Selection. In order to obtain the best results at the end of each iteration, without exponentially increasing the size of the population, we have to determine which individuals are selected for the rest of the process. In this case, we focus on the individuals that are able to dominate in both objectives over the rest, forming *fronts*.

Although there exist several methods to perform the selection, most of them require a percentage of relevance for each individual, highly related to a fitness function [11] (roulette wheel, stochastic universal). Since we are targeting this multi-objective problem, we focus on two methods that do not require such percentage. These methods are *truncation* and *tournament*:

- *Truncation* is an elitist method that chooses a percentage of the population that fits better the objectives we are minimising. The number of individuals that should be selected should correspond to a range varying from 10% to 50% of the actual population. Then, the selected individuals are duplicated as many times as required to maintain the size of the population. In our case, we consider a 25% of individuals, where each of them appears 4 times.
- *Tournament* is a selection technique that individually selects each chromosome from small groups of, usually, two or three participants. The way of deciding who wins each small tournament can vary. The deterministic approach selects the individual that dominates the others. Another option is a probabilistic tournament, where a random number between 0 and 1 is chosen. If the number is smaller than an initial parameter, then the first chromosome is the one that wins the tournament and is selected. Otherwise, the other participants have a chance to compete without this rejected competitor. In this way, we increase the diversity, and a high fixed parameter would still produce a high fitted population. In our case, we consider 3 participants with a 0.8 ratio of victory for the better suited participant.

Crossover. The crossover phase focuses on the exploitation of the search space, as it aims to find new solutions that are combinations of already existing ones. The crossover operators that we use exchange tests in several pairs of individuals. We have considered the following two methods:

- *Standard crossover*: we select a point in the list of tests of an individual that we will use to split it into two parts. Then, we produce two individuals with the first half of one of the parents and the second half of the other parent.
- *Continuous crossover*: for each test in each parent individual, there is a chance to be swapped for the respective test in the other parent's list. In this way, the children generated have a more mixed configuration of the tests.

Mutation. The mutation phase aims at fully explore the search space, adding new solutions that were not considered before, trying to increment the diversity of the population. Even if a bad solution arises from this method, the small

ratio of mutation operations does not have a negative impact, as such solution would be discarded soon. However, if a promising solution is obtained, it is likely to stay and improve the overall set of solutions. The mutation usually makes small modifications on an individual to avoid local optima. Here we propose two methods:

- The *adding mutation* approach introduces a new test to the subset that was not being considered before. Since this increases the size of the solution, it is ideal for small solutions that are not complete.
- The *replacement mutation* approach substitutes a test that was being considered by a chromosome by a new one, where more changes could be produced. This mutation is oriented to big sets of tests, where there are redundant tests or partial solutions that do not generate a substantial increase in mutation score nor reduces the inputs to apply.

Replacement. The last step of a GA is to fix the population for the next generation. For that, we have to decide which parents and children continue, as the size of the population should stay constant. We have only considered a replacement method oriented to solve multi-objective problems.

In this NSGA-II replacement we combine the parent and children population, having twice the desired size. Then, we generate fronts of solutions that do not dominate each other, having different levels of fronts. This ensure elitism, producing an increase of the performance of the solution after each generation. Then, we include the best fronts in the new population until we reach the specified size. If at any point it is impossible to add a complete front, we add the chromosomes that are more spread in such front, as they are more likely to have important information rather than a group of solutions really close to each other.

2.3 Cluster Algorithm

In this section we briefly describe the fundamentals of the cluster technique that we use. First, it is important to note that clustering is a machine learning algorithm used to group similar objects based on certain criteria and features, without any prior knowledge of the groups. In our experiments, we use a method known as Agglomerative Clustering [17]. This clustering algorithm begins by considering each test case as a separate cluster. Then, the similarity between each pair of clusters is calculated using the Euclidean distance. Subsequently, the two most similar clusters are merged into a single cluster. At this stage, it is necessary to employ a linkage criterion to establish how similarity between clusters is measured. In this case, we use the Ward criterion [18]. The process of calculating the similarity between each pair of clusters and merging the two most similar clusters into a single cluster iterates until the number of clusters reaches a predetermined value established beforehand. For our experiments, we use clustering on the tests only knowing what mutants each of them kills. Also we perform this clustering to obtain from 2 to 30 clusters, each of them yielding

Algorithm 1. Agglomerative Clustering

Input: Dataset $\{t_1, t_2, \ldots, t_n\}$, distance metric d, number of clusters T
Output: Final set of T clusters

1: Initialise each test as a separate cluster: $C \leftarrow \{\{t_1\}, \{t_2\}, \ldots, \{t_n\}\}$
2: **while** $T < |C|$ **do**
3: Compute pairwise distances between clusters in C using distance metric d.
4: Find the two closest clusters, (C_i, C_j)
5: Merge clusters C_i and C_j: $M \leftarrow C_i \cup C_j$
6: $C \leftarrow (C \cup M) \setminus \{C_i, C_j\}$
7: **end while**
8: **return** Final set of T clusters C

one solution. In particular, the subset of tests made of the shortest test of each cluster.

Algorithm 1 presents a generalised pseudo-code for the Agglomerative Clustering method. The required input is a set with n tests and a predefined distance metric, d (in our case, the Euclidean distance), along with the desired number of clusters, T. The output will be the final set containing T clusters. In line 1, the algorithm assigns each test to a separate cluster. Then, the clusters are processed until the number of clusters equals to T. The algorithm computes the distances between all pairs of clusters (line 3). The two closest clusters are then identified (line 4), based on the computed distances, and merged into a single cluster, forming a new cluster (line 5). This new cluster, will replace the merged clusters (line 6).

3 Experiments

We have compared our two techniques with a total of 9 SUTs, ranging from smaller specifications that generated 400 mutants to bigger ones with 12800. The length of the sequence of inputs of our tests has a range that varies from 202 to 6428. We evaluate how good our methods are in these SUTs by obtaining representative results and we show how much we are able to improve the mutation score as the number of inputs grows. Additionally, we have included a random selection of solutions and a bad set of solutions to visualise the range of the possible results. This is important as no matter how you select a group of tests, at a given point adding more and increasing the sequence of inputs will result in detecting more faults, and thus the mutation score will improve as well. Because of that, the different approaches must have similar values near the extreme values, i.e. the empty set and the full set. In order to obtain the bad set of solutions[1], we used the GAs with opposite objectives. In this case, the target was to minimise the mutation score (maximising $1 - ms$) and to maximise the number of inputs at the same time. The implementation of the

[1] Represented in Figs. 6 and 7 as *Worst*. The *Random* set is chosen by randomly selecting tests to have different solutions with different sizes.

described methods, the SUTs, and all the results are available at https://github.com/miguelbpsg/ICCCI24.

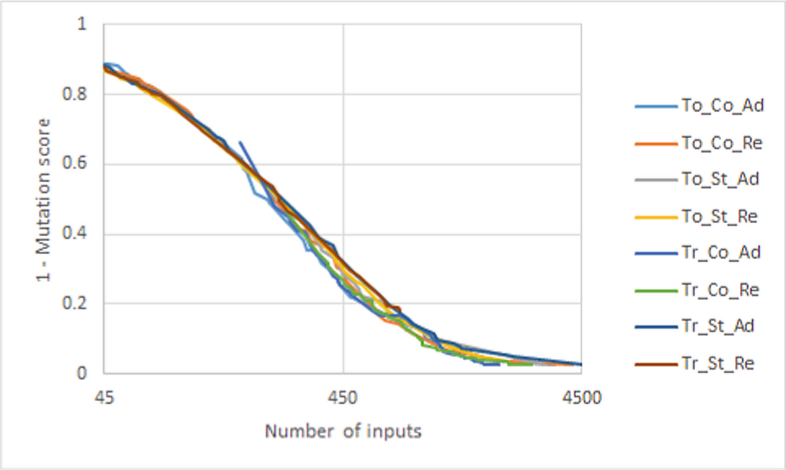

Fig. 5. Comparison of GAs

First, we have performed all 8 configurations of the GAs a total of 20 times. To control the random behaviour and reduce the threat of validity of our experiments we use the median of the 20 runs as the representative value for each configuration. This also means that the GAs may provide better solutions when executed once, but one individual run is not representative, and should be repeated more times. Figure 5 shows that the different configurations of the GA yield similar Pareto fronts. However, an important feature of these algorithms is diversity, that is distributing the solutions in an extensive manner. For that, the truncation method is too tight, as it focuses on a small number of solutions. Although the results of this selection method are comparable to the tournament method, no matter which crossover or mutation method is used, the range is smaller. Such range is skewed towards fewer inputs when using the standard crossover or the replacement mutation, and towards bigger inputs with the continuous crossover or the adding mutation (all the results can be checked in the files and graphs in github). Considering the tournament selection method, the solutions obtained are distributed through all the possible inputs. The four configurations for this selection are of an equivalent quality, but in particular the continuous crossover and replacement mutation yield solutions that are further from one another, thereby covering a wider solution space. For that reason, in order to compare the GAs with the clustering approach and to easily visualise the results, we will only show such configuration of the GAs corresponding to tournament selection, continuous crossover and replacement mutation.

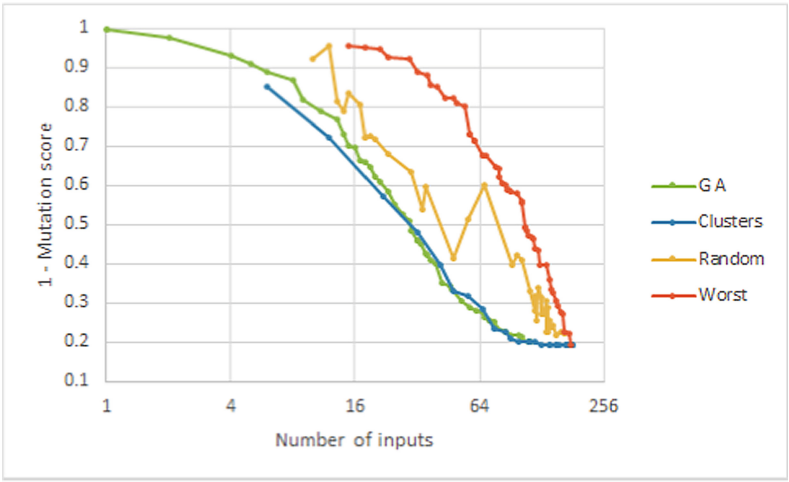

Fig. 6. Small experiment

As the Agglomerative Clustering method is deterministic, only one execution is needed to have a valid result which is later analysed and compared in Figs. 6 and 7. The former corresponds to the smallest experiment we perform with a set of tests with a total of 202 inputs and 400 mutants to detect. In the graph, we observe that the clustering method slightly improves the GA when it comes to selecting small subsets, that is when few clusters are being considered. However, the GA is able to give more uniform solutions, starting from one input to all of them. Besides, the lines for the GA and the clustering method are very similar, having closer solutions among the clustering method than among the GA. Both our methods always dominate considerably a random solution, that simulates selecting a subset without any criteria or knowledge but the number of inputs to perform. The latter corresponds to the biggest experiment we perform with a set of tests with a total of 6428 inputs and 12800 mutants to detect. Here the same trend is maintained. The clustering approach have a higher mutation score (lower $1 - ms$) than GAs for small subsets, but as more tests are selected and the number of inputs grows, both methods get closer while obtaining better solutions than any random one throughout the spectrum that the GA fully covers. It is important to note too that the scale for the number of inputs is logarithmic, so in order to increase the mutation score, the number of inputs has to substantially grow each time.

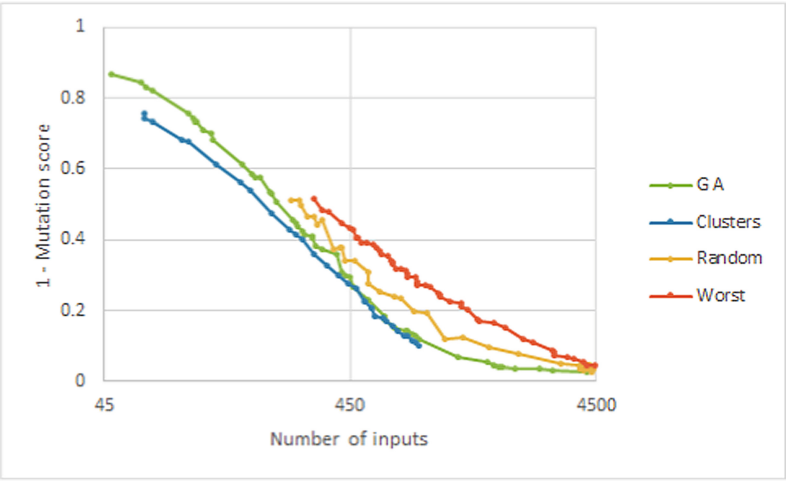

Fig. 7. Big experiment

From these experiments we determine that our proposed methods are able to obtain subsets of tests that minimise the cost while detecting many faults.

4 Conclusions and Future Work

In this paper we have presented a clustering method and a genetic algorithm to select a good subset of tests that detects a big number of mutants applying a small amount of inputs. We have performed experiments on several SUTs to obtain subsets that optimise these objectives, and compare them. We have concluded that both techniques yield great results, being the clustering slightly better but more specific, and the GA much more spread while consistently providing good solutions. Our proposal significantly simplifies the task of selecting *good* tests from a large test suite in order to validate a SUT and provides a diverse group of solutions depending on which objective is more important to prioritise.

Future lines of work include the use of other Machine Learning methods [14] to select good tests, extending the methods for the different phases of the GA, and specially combining the two techniques to obtain better solutions in a more spread space. Additionally, this work could be extended to other formalisms, or to systems coded in popular programming languages like Java, Python or C. Another approach involves considering different metrics to optimise. For example, some inputs may take more time than others to be executed in a real system, or some actions may be more critical and thus more expensive to be tested, changing the objective of minimising the number of inputs to minimise the *actual* cost or time of testing. These objectives do not necessarily have to replace the current ones, but could also be added to the problem, raising it to a many-objective one.

References

1. Arafeen, Md.J., Do, H.: Test case prioritization using requirements-based clustering. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 312–321 (2013)
2. Benito-Parejo, M., Merayo, M.G.: An evolutionary algorithm for selection of test cases. In: 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2020)
3. Benito-Parejo, M., Merayo, M.G.: Using genetic algorithms to select test cases for finite state machines with timeouts. In: 2021 IEEE Congress on Evolutionary Computation (CEC), pp. 2403–2410 (2021)
4. Catal, C., Mishra, D.: Test case prioritization: a systematic mapping study. Softw. Qual. J. **21**(3), 445–478 (2013)
5. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
6. Goldberg, D.E.: Genetic Algorithms in Search, Optimisation and Machine Learning. Addison-Wesley, Boston (1989)
7. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: local, global, and hybrid search. IEEE Trans. Softw. Eng. **36**(2), 226–247 (2010)
8. Hierons, R.M., Merayo, M.G., Núñez, M., Mutation testing. : Laplante, P.A. (ed.) Encyclopedia of Software Engineering, pp. 594–602. Taylor & Francis (2010)
9. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
10. Jones, B.F., Eyres, D.E., Sthamer, H.-H.: A strategy for using genetic algorithms to automate branch and fault-based testing. Comput. J. **41**(2), 98–107 (1998)
11. Katoch, S., Chauhan, S.S., Kumar, V.: A review on genetic algorithm: past, present, and future. Multimedia Tools Appl. **80**, 8091–8126 (2021)
12. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. Softw. Pract. Exp. **21**(7), 685–718 (1991)
13. Lou, Y., Hao, D., Zhang, L., Mutation-based test-case prioritization in software evolution. In: 26th International Symposium on Software Reliability Engineering. ISSRE'15, pp. 46–57. IEEE Computer Society (2015)
14. Méndez, M., Merayo, M.G., Núñez, M.: Machine learning algorithms to forecast air quality: a survey. Artif. Intell. Rev. **56**(9), 10031–10066 (2023)
15. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Trans. Softw. Eng. **27**(12), 1085–1110 (2001)
16. Mirjalili, S.: Evolutionary Algorithms and Neural Networks. SCI, vol. 780. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-93025-1
17. Müllner, D., Modern hierarchical, agglomerative clustering algorithms. arXiv, abs/1109.2378 (2011)
18. Murtagh, F., Legendre, P.: Ward's hierarchical agglomerative clustering method: which algorithms implement ward's criterion? J. Classif. **31**(3), 274–295 (2014)
19. Núñez, A., Merayo, M.G., Hierons, R.M., Núñez, M.: Using genetic algorithms to generate test sequences for complex timed systems. Soft. Comput. **17**(2), 301–315 (2013)
20. Gbeminiyi John Oyewole and George Alex Thopil: Data clustering: application and trends. Artif. Intell. Rev. **56**(7), 6439–6475 (2022)
21. Pareto, V.: Cours d'économie politique, vol. 1. Librairie Droz (1964)

22. Petrović, G., Ivanković, M., Fraser, G., Just, R.: Does mutation testing improve testing practices? In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 910–921 (2021)
23. Shin, D., Yoo, S., Papadakis, M., Bae, D.H.: Empirical evaluation of mutation-based test case prioritization techniques. Softw. Test. Verif. Reliab. **29**(1–2), e1695 (2019)
24. Sivanandam, S.N., Deepa, S.N.: Introduction to Genetic Algorithms. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-73190-0
25. Wei, C., Yao, X., Gong, D., Liu, H.: Spectral clustering based mutant reduction for mutation testing. Inf. Softw. Technol. **132**, 106502 (2021)
26. Xu, R., Wunsch, D.: Survey of clustering algorithms. IEEE Trans. Neural Netw. **16**(3), 645–678 (2005)