Using genetic algorithms to generate test suites for FSMs[⋆]

Miguel Benito-Parejo[1], Inmaculada Medina-Bulo[2][0000−0002−7543−2671], Mercedes G. Merayo[1][0000−0002−4634−4082], and Manuel Núñez[1][0000−0001−9808−6401]

[1] Departamento Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{mibeni01,mgmerayo,manuelnu}@ucm.es
[2] Departmento de Ingeniería Informática, Escuela de Ingeniería
Universidad de Cádiz, Spain
inmaculada.medina@uca.es

Abstract. It is unaffordable to apply all the possible tests to an implementation in order to assess its correctness. Therefore, it is necessary to select relatively small subsets of tests that can detect many errors. In this paper we use different approaches to select these test suites. In order to decide how good a test suite is, we confront it with a set of *mutants*, that is, small variations of the specification of the system to be developed. The goal is that our algorithms build test suites that *kill* as many mutants as possible. We compare the different approaches (consider all the possible subsets up to a given number of inputs, intelligent greedy algorithm and different genetic algorithms) and discuss the obtained results. The whole framework has been fully implemented and the tool is freely available.

Keywords: Genetic algorithms · Testing from FSMs · Mutation testing.

1 Introduction

Testing is one of the main techniques to validate the correctness of software systems [1]. Quite often we find ourselves with a group of properties that should be satisfied by the system under development and we want to reassure that it does. In testing, these properties are encoded as tests and we have to check that the system, usually called System Under Test (SUT), successfully passes them. In practice, this approach is unfeasible because the number of tests may be astronomical (one property will give rise to many tests). In addition, we may have a bound on the number of tests that we can apply (e.g. due to budget or temporal constraints). Therefore, it is important to *wisely* choose among these tests a subset that is able to detect most faults. Clearly, the method to select

[⋆] Research partially supported by the Spanish MINECO/FEDER project DArDOS (TIN2015-65845-C3-1-R and TIN2015-65845-C3-3-R) and the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314).

these tests should rely on a measure of how good a test is. In this line, mutation testing [3, 8, 11] 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 is probably good at discovering faults. The technique 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 analyze different strategies 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 (possibly constructed from the specification and representing most representative faults) 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 *complexity* (we will measure the complexity of a test suite in terms of the number of inputs included in it) that kills as many mutants as possible. If T is the whole set of tests and I is the bound on the number of inputs, then the obvious solution is to compute all the subsets of T with up to I inputs, apply them to the set of mutants and choose the subset killing more mutants. Unfortunately, in this case we have a combinatorial explosion that disallows us to use this approach. A second possibility, based on previous work [2], considers a greedy algorithm where we select the best tests individually (according to the number of mutants that they kill) until we reach the limit of I inputs. This technique will generally provide good results, both in cost and in faults detected, 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 developed a genetic algorithm to find better solutions than the greedy algorithm. The algorithm is versatile and allows users to exercise different variants. We have developed a tool that fully implements all the algorithms, with its variants, presented in the paper. Finally, we have performed several experiments to compare the different methods.

The rest of the paper is structured as follows. In Section 2 we introduce the main concepts used in the paper. In Section 3 we enumerate the considered methods to select test suites. In Section 4 we report on our experiments. Finally, in Section 5 we present our conclusions and some lines for future work.

2 Preliminaries

In this section, we introduce the main background and concepts required for the paper.

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,* $i \in I$ *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*

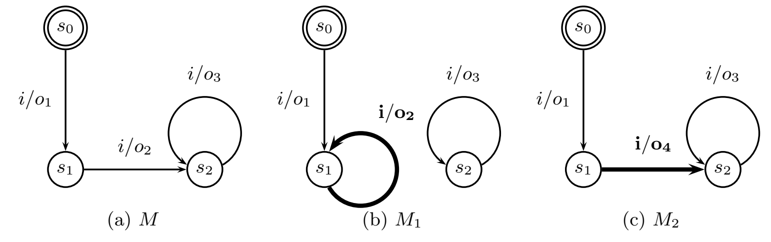


Fig. 1: An FSM and two of its mutants

$i \in I$, 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 paper we will restrict ourselves to input-enabled deterministic FSMs, that is, from each state of the machine, it is possible to perform all the inputs and there will be only one possible evolution. This restriction mimics testing of programs: programs are (usually) deterministic and should react to any received input. FSMs will be used to represent specifications and mutants. Although the FSM formalism looks simple, it has a common base with more complex frameworks to represent black-box systems. Therefore, the framework presented in this paper can be extended as long as we consider systems where we apply a sequence of inputs and decide whether the returned output sequence is expected or nor. Next we introduce the notion of mutant.

Definition 2. *Let $M = (S, I, O, Tr, s_{in})$ be an FSM. We say that an 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 $(s, s', i, o') \in Tr'$, with $o \neq o'$, or by $(s, s'', i, o) \in Tr'$, with $s' \neq s''$.*

Note that mutants are still deterministic and input-enabled. For example, consider the FSM given in Figure 1a, being s_0 the initial state. Two possible mutants are shown in Figures 1b and 1c: the first one represents the change of final state of a transition while the second one represents a change in the output. The next concept that we need is the notion of test.

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

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

Consider again M, M_1 and M_2. For example, we have that $t_1 = (i, o_1)$, $t_2 = (ii, o_1o_2)$ and $t_3 = (iii, o_1o_2o_3)$ are tests for M. Moreover, 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 next concept

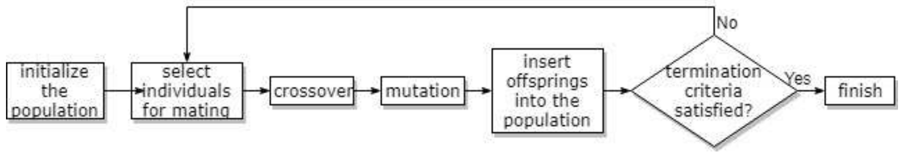

initialize the population
select individuals for mating
crossover
mutation
insert offsprings into the population
termination criteria satisfied?
No
Yes
finish

Fig. 2: GA flowchart

allows us to represent in a matrix whether and when a test kills a mutant. It will be important to record the first input showing a faulty behavior because we might be able to discard the rest of the test.

Definition 4. *Let $M = (S, I, O, Tr, s_{in})$ be an FSM, $\{t_i\}_{i=1}^{n}$ be a set of tests for M and $\{M_j\}_{j=1}^{m}$ be a set of mutants of M. We define a* results table *as a matrix $(a_{ij})_{i=1,j=1}^{n,m}$, where a_{ij} is the length of the shortest prefix of test t_i that kills mutant M_j. In the case that such mutant passes the test, this distance will be equal to infinity.*

A *Genetic Algorithm* (GA) [5, 15] is a heuristic optimization technique, which it is inspired in a metaphor of the processes of evolution in nature. GAs and other meta-heuristic algorithms have been used to automate software testing [4, 6, 12, 13]. 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 Figure 2.

3 Selection methods

In this section we present each of the considered approaches to solve the problem of finding good sets of tests. Each of them will be based on how good a test is, which is represented by the number of mutants that it kills and its length (number of inputs). This shows that two tests killing the same mutants might not be equally good, as one detecting them sooner will have a better score[3] because it will save resources.

[3] The score is computed by dividing the fitness of an individual between the sum of all fitness.

$$\begin{pmatrix} 4 & 7 & 5 & \infty & 12 & \infty \\ 13 & \infty & \infty & 6 & 1 & \infty \\ \infty & 9 & \infty & 8 & \infty & \infty \\ 5 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

(a) Original matrix

$$\begin{pmatrix} 6 & \infty \\ 8 & \infty \\ \infty & \infty \\ \infty & 7 \\ \infty & \infty \end{pmatrix}$$

(b) Reduced matrix

Fig. 3: Matrix simplification

3.1 Global search

The global search approach looks through all the possible combinations of the initial set of tests having less inputs than the given bound. This approach always provides the best solution because it explores all the possible subsets. Therefore, it is useful because it helps to compare the results with the forthcoming algorithms. The negative part is that for non-trivial systems, it gets impossible to apply it because it suffers of a combinatorial explosion. In fact, we were able to compute it only for the smallest of our experiments.

3.2 Greedy algorithm

Our greedy algorithm is based on the matrix including information about tests and mutants (see Definition 4). 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. Then, we add the first test to the subset that we are constructing. Afterwards, we remove that test from the matrix, as well as all the mutants that it has killed. After reducing the matrix, we iterate the process until either we have killed all the mutants or until we reach the upper bound on the number of inputs. We can see an example in Figure 3a: a matrix where rows represent tests and columns represent mutants. As such, the first row is the best test, which will be selected, and will kill the mutants 1, 2, 3 and 5. As a result, we obtain the reduced matrix given in Figure 3b, where one test and four mutants have being removed. The resulting matrix needs to be resorted, and equivalent rows do not necessarily represent the same test, since several entries (comparing with the dead mutants) have been omitted. In addition, we have to take into account that the selected set of tests has length 12 (the length needed to kill all the mutants).

A good property of this algorithm is that it works in low polynomial order over an space, that is the matrix, that reduces its size after each iteration. This is the less costly, in terms of time needed to compute the solution, algorithm that we present in this paper. Our greedy method shows great results and will also help bounding the number of generations and the global cost of our next algorithm.

3.3 Genetic algorithm

As we know, GAs shine 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 3.1. Therefore, a genetic algorithm is a sensible approach to compete with our greedy algorithm, in particular, taking into account that our greedy algorithm computes relatively good solutions using a short period of time.

In our population, 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 or we stop the execution because we have reached the time limit. Even though we will spend more time than the greedy algorithm in the computation of the solution, we would like to have a reasonable bound.

We have implemented our algorithms and the tool is freely available at https://github.com/miguelbpsg/IWANN19. Our tool needs different parameters that can be received in a dynamic way. We have an interface where we can provide how many inputs we expect in the solution, the type of selection of the population, the type and rate of crossover, the rate of mutation (of the genetic algorithm, not to be confused with the mutants of the specification), and more concrete fields in specific cases. Next, we briefly describe these parameters.

Initialization method. The type of initialization that we have designed is *incremental initialization*. It provides a variety of chromosomes, with a different amount of test and inputs each, which generates more 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. As some individuals might be better than others, the transition from one generation to the next one has to ensure that the representatives of the foremost have to be selected. The idea is to reward the best ones with more appearances and the worst ones with even no appearances at all. We allow the user of our tool to use alternative selection models: *Linear rank*, *remains*, *roulette wheel*, *tournament*, *truncation* and *stochastic universal.* Since these selection models are quite common [5, 14], we do not explain the details about how each of them is applied to our populations.

Crossover methods. Concerning crossover, combining elements of the population to produce a new generation after an iteration of the algorithm, we have designed two methods. First, *standard crossover* takes two chromosomes of the population and chooses a random point at each of their respective list of tests. Then, the left part of each list is preserved and the right part is exchanged. If such modification generates a test suite with more inputs than the fixed bound, then the last tests are discarded until the bound is reached. Second, *continuous* *crossover* takes two chromosomes and randomly exchanges the test at each position for both lists. Therefore, more possible combinations of swaps can happen.

Mutation of population methods. As the initial seeds for the population might not be complete, it is sensible to refresh the population with some slight changes that could renew some stale state. In our case, we designed two different techniques. The first one considers *adding mutation*, that is, it 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. Despite increasing the number of tests on the whole population, using the appropriate crossover method, it is possible to generate an exchange of tests where some of them have to be discarded. As a consequence, this method complements the possible loss of these tests. The second method is based on *replacing mutation*. This method allows an individual to change one of its test by another one coming from the initial set of all provided tests. 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.

The last step of each generation on the genetic algorithm is to replace the population by the new one. Again, we have two possibilities. On one hand, the trivial option would be to simply substitute the last population by the new one, even if it could be worse. In this way, less operations are performed at this stage and, as a result, the execution will be faster. On the other hand, we could replace a high percentage of the new generation on the previous one, where the best individuals could still stay. This approach will always allow the population to keep the best partial solution until it is improved. As a counterpart, more calculations have to be made and that cost might decelerate the program so that the GA will perform less iterations in the same amount of time.

Finally, the heuristics that we use to define the fitness function enhances the individuals that locally improve the expected results of testing because it takes into account how many mutants are killed. Specifically, the fitness is calibrated by adding the minimum number of inputs required to kill a mutant considering the subset of tests that the chromosome has at a moment, through all the mutants. This value does punish leaving mutants alive, as a penalty is added to the final value for each living mutant. Therefore, the more mutants a subset kills, the lower the score will be, and a lesser number of inputs required to kill more mutants will also reduce the value, leading us to a minimization problem (a lower value of fitness denotes a better population).

In Figure 4 we show the results of one experiment concerning how fitness varies along generations. The results are as expected. In short, there is a relatively big variance concerning the worse individual of each generation (that is, the highest value of fitness), this variance is smaller for average fitness (and the value notoriously improves after only 10 generations), generational best stabilizes (although there are small variations), while absolute best quickly converges.

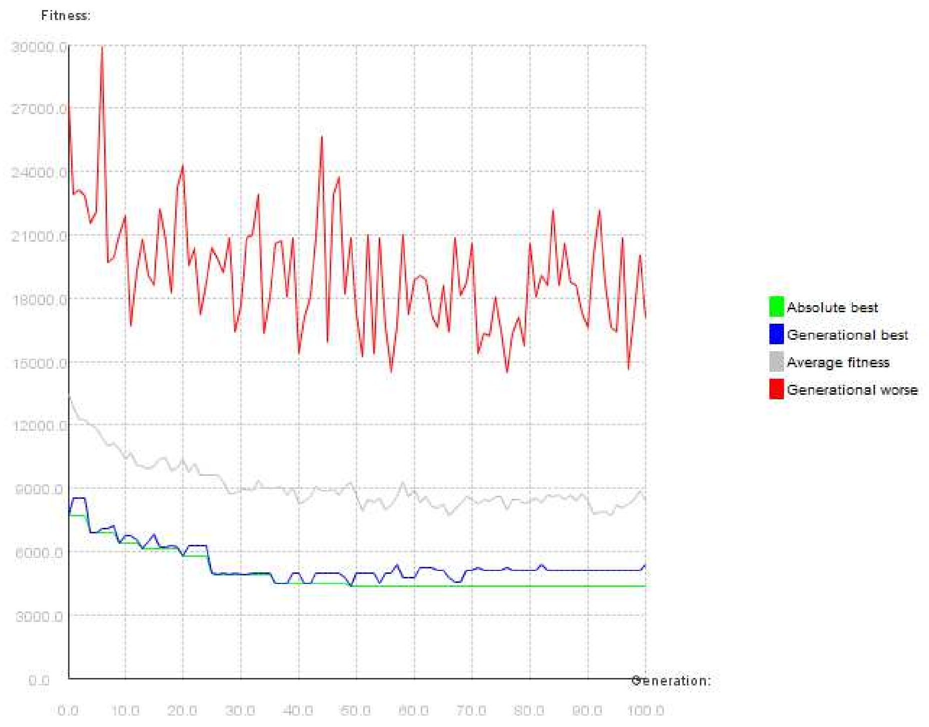


Fig. 4: Genetic evolution

4 Experiments

In this section we comment on the experiments that we have performed, on why we chose to make them, the results obtained in each of them provided, and the main differences between experiments. We also discuss the results of trying our three previous methods into a specification, with different bounds on the number of inputs that a solution could have. We have analyzed the time that the different approaches need to compute their solutions.

4.1 Description of the experiments

Our experiments consisted in the execution of the three described algorithms over the same specification, mutants, initial tests and maximum number of allowed inputs, for several combinations of them. Afterwards, we compare the results both in time needed to compute the solution and in the goodness of the solution. Note that the initial set of tests should not be confused with the set of tests produced in the initialization of the GA. The former is provided as a precondition of the problem. This is the set of tests that we should aim to exercise but if our resources do not allow us to try all of them (for example, because they have more inputs than the considered bound), then we should apply a *good* subset of them (computing this final subset is the goal of our approaches). The latter is computed as the first step of the solution of the problem.

It is clear that the smaller the FSM is, the lesser number of mutants we will have. We have considered a fixed specification with 10 states, 3 different inputs and 5 different outputs generating near 300 mutants. We also considered three possible bounds on the number of inputs, and two possible initial sets of tests, resulting in six representative cases. Next we give the details of each of them.

1. The first experiment consisted of allowing a maximum of 30 inputs in the solution and starting with a set of 99 tests. All algorithms yield a very good (if not the best) solution on a reasonable amount of time as there are few possible results. This experiment is a good baseline that all three algorithms provide good solutions.
2. Next, we increased the bound on the number of inputs, from 30 to 80, maintaining the initial set of tests. This experiment should show us the evolution of the algorithms depending on the number of inputs.
3. To conclude the 99 initial tests experiments, we increased the bound on the number of inputs up to 150 inputs.
4. The next variation considered the smallest bound on the number of inputs (that is, 30 inputs) on a bigger set of tests (we consider a set of 957 tests).
5. The next experiment considers the intermediate bound (that is, 80) and the big set of tests.
6. Finally, the last experiment consider the biggest bound on the number of inputs and the bigger set of tests (150 inputs and 957 tests, respectively).

These experiments were performed with our default options, that is, tournament selection over 3 participants with a ratio of 80%, continuous crossover with a probability of 0.6, the extra test mutation option with a 0.02 coefficient and a direct replacement. It is worth to mention that most of the combinations of our GAs gave very similar results. It should be also noted that the elitist replacement always produced better fitness results at the cost of a longer execution time while roulette wheel and stochastic universal selection methods produced slightly worse results.

We also tested what bounds and initial sets of tests the full search was able to compute. We were not able to exceed a bound of 60 inputs over 99 initial tests, taking over a week to compute the solution.

4.2 Evaluation

As expected, as soon as the systems were sizable enough and we had a nontrivial set of tests to choose start with, generating all the possible subsets was unfeasible due to the combinatory explosion on the number of subsets. For example, if we had 40 initial tests with an average of 10 inputs, and we could choose tests up to 150 inputs to finally execute them, we would have more than 40 billion possible

	Time Exp. 1	Time Exp. 2	Time Exp. 3	Time Exp. 4	Time Exp. 5	Time Exp. 6
Genetic	91	190	220	208	296	462
Greedy	22	23	26	147	178	263
Full search	1.355	–	–	–	–	–

Table 1: Time results (in milliseconds)

	Fitness Exp. 1	Fitness Exp. 2	Fitness Exp. 3	Fitness Exp. 4	Fitness Exp. 5	Fitness Exp. 6
Genetic	11.887	5.866	2.877	12.225	3.212	1.988
Greedy	16.166	7.687	5.415	22.746	13.543	7.429
Full search	11.716	–	–	–	–	–

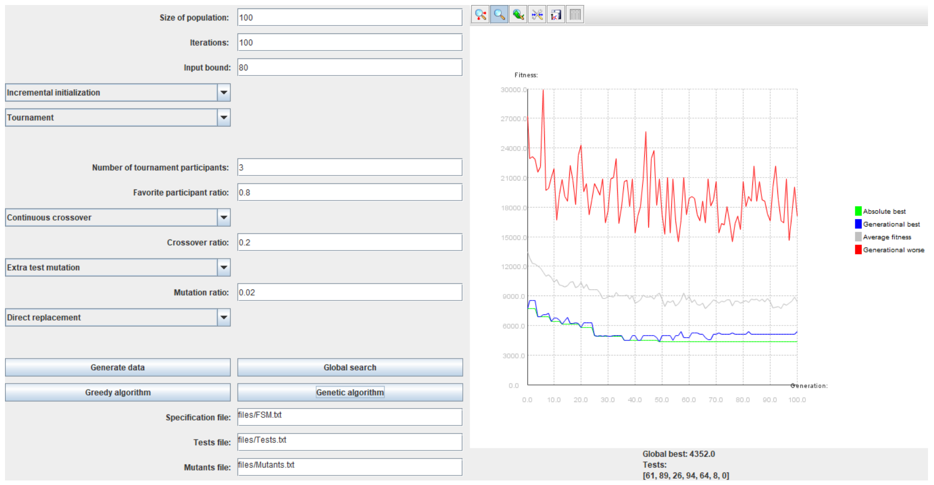
Table 2: Fitness results (higher values denote worse results)

subsets, with 150 inputs, of the original set of tests. Nevertheless, whenever we find ourselves with small bounds, it is possible to compute an optimal result, guaranteeing that we obtain the best subset of tests to use on the SUT.

In terms of relative cost, we have that the greedy algorithm was always the fastest and we can see this in Table 1. The time needed to compute the solution mainly depended on the size of the given set of tests but it also had a small dependence on the number of inputs allowed, since this determines how many times the matrix has to be sorted. Considering its efficiency, our GA was able to provide very good results, almost equivalent to the optimal in the only experiment where we were able to compute all the combinations, as Table 2 shows (11.761 fitness of the full search vs. 11.887 of the GA). However, the solution was computed in a much faster time, showing evidence of the usefulness of this technique.

Focusing now on Tables 1 and 2, we have that a higher bound on the number of inputs always increases the execution time, as more possibilities are available. In contrast, the fitness of the obtained solution improves. Also, comparing all the experiments, we observe that a bigger initial set of tests induces a higher execution time, but better results are obtained for the same bound of inputs.

These experiments show that the GA can adequately compete, depending on the resources, and complement the results of the greedy algorithm. It is true that the GA requires some more time to evaluate, but considering the obtained results we find it worth to use this extra computing power. Finally, let us note that the whole implementation and some additional examples are freely available at https://github.com/miguelbpsg/IWANN19. In Figure 5 we give a screenshot of our tool.



Size of population: 100
Iterations: 100
Input bound: 80
Incremental initialization
Tournament
Number of tournament participants: 3
Favorite participant ratio: 0.8
Continuous crossover
Crossover ratio: 0.2
Extra test mutation
Mutation ratio: 0.02
Direct replacement
Specification file: files/FSM.txt
Tests file: files/Tests.txt
Mutants file: files/Mutants.txt

Global best: 4352.0
Tests:
[61, 89, 26, 94, 64, 8, 0]

Fig. 5: GUI of our tool

5 Conclusions and future work

In this paper we present different solutions to the problem of obtaining good sets of tests out of big sets. Ideally, if a tester is provided with a whole set of tests then the tester should apply all of them to the SUT. However, the time and resources devoted to testing are usually limited and the tester can apply only a subset of these tests. If we are working within a framework where the tester applies inputs and receive outputs, then this bound is given by the number of inputs that the tester can apply. This is an important problem in testing and in addition to provide a sound theoretical framework, it is a must to develop tools supporting the frameworks. We have a tool implementing all the algorithms presented in the paper, that is, our tool is able to, given an initial set of tests and the maximum number of inputs that we can really apply, compute a subset of the initial set with any of the algorithm, and the different variants of the GA, discussed in this paper. In addition, the tool supports the process of generating mutants from a specification of the SUT.

The results show that our GA usually finds an excellent solution. In general, the GA beats the greedy algorithm needing a slightly higher amount of time to compute the result. For smaller experiments, where full search could be effectively computed, the differences between the best solution and the one obtained through the GA were very small: the fitness of the full search approach was around $1,5\%$ better but it needed 14 times longer to compute it. Therefore, we can be satisfied with the results considering the complexity of the problem.

As future work, we plan to extend the framework to deal with other FSM-like formalisms. A first line of work is to consider probabilistic FSMs, where nondeterminism is probabilistically quantified. We will take as initial step our previous work on mutation testing of probabilistic FSMs [7] complemented with our recent work on conformance relations for probabilistic systems[10]. An orthogonal line or work that we would like to pursue is to adapt our framework to test in the distributed architecture [9], where several users interact over the same data but cannot observe what the others are doing. Finally, we would like to improve the usability and report features of our GUI so that the whole interaction with the algorithms and its extensions could be followed and such that more complex graphs could be shown and compared.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, 2nd edn. (2017)
2. Andrés, C., Merayo, M.G., Núñez, M.: Formal passive testing of timed systems: Theory and tools. Software Testing, Verification and Reliability **22**(6), 365–405 (2012)
3. Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J.: Mutation testing. In: Encyclopedia of Information Science and Technology, pp. 7212–7221. IGI Global, 3rd edn. (2014)
4. Derderian, K., Merayo, M.G., Hierons, R.M., Núñez, M.: 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. pp. 396–403. Springer (2011)
5. Goldberg, D.: Genetic Algorithms in Search, Optimisation and Machine Learning. Addison-Wesley (1989)
6. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Transactions on Software Engineering **36**(2), 226–247 (2010)
7. Hierons, R.M., Merayo, M.G.: Mutation testing from probabilistic and stochastic finite state machines. Journal of Systems and Software **82**(11), 1804–1818 (2009)
8. Hierons, R.M., Merayo, M.G., Núñez, M.: Mutation testing. In: Laplante, P.A. (ed.) Encyclopedia of Software Engineering, pp. 594–602. Taylor & Francis (2010)
9. Hierons, R.M., Merayo, M.G., Núñez, M.: Bounded reordering in the distributed test architecture. IEEE Transactions on Reliability **67**(2), 522–537 (2018)
10. Hierons, R.M., Núñez, M.: Implementation relations and probabilistic schedulers in the distributed test architecture. Journal of Systems and Software **132**, 319–335 (2017)
11. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011)
12. Jones, B.F., Eyres, D.E., Sthamer, H.H.: A strategy for using genetic algorithms to automate branch and fault-based testing. The Computer Journal **41**(2), 98–107 (1998)
13. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Transactions on Software Engineering **27**(12), 1085–1110 (2001)
14. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. Springer, 3rd, revised and extended edn. (1996)
15. Srinivas, M., Patnaik, L.M.: Genetic algorithms: A survey. IEEE Computer **27**, 17–27 (1994)