GPTSG: A Genetic Programming test suite generator using Information Theory measures ⋆

Alfredo Ibias[1], David Griñán[2], and Manuel Núñez[1][0000−0001−9808−6401]

[1] Complutense University of Madrid, Madrid 28040, Spain
{aibias,manuelnu}@ucm.es
[2] Polytechnic University of Madrid, Madrid 28223, Spain
david.grinanm@alumnos.upm.es

Abstract. The automatic generation of test suites that get the best score with respect to a given measure is costly in terms of computational power. In this paper we present a genetic programming approach for generating test suites that get a good enough score for a given measure. We consider a black-box scenario and include different Information Theory measures. Our approach is supported by a tool that will actually generate test suites according to different parameters. We present the results of a small experiment where we used our tool to compare the goodness of different measures.

Keywords: Testing; Genetic Programming; Test generation; Information Theory

1 Introduction

Software testing [2,20] is an important research topic because it helps to ensure that the systems work as expected. Specially important have been the efforts to define testing in a formal way, which is still an active research area [3]. There are two orthogonal approaches to test a system: Consider that the system is a white-box or consider that it is a black-box. In this paper, we will focus on the latter as it poses a higher challenge because we have to rely on a model of the system, but we cannot see how the implementation of the system works. One of the main components of black-box testing consists in generating test suites that find the maximum number of faults. Actually, this is a critical part when we are talking about systems that require a lot of time to process an input or that have a small time window where you can test them. As it is a critical task, a lot of work around generating and selecting test suites and constructing tools supporting the theoretical frameworks has been performed [23].

Genetic programming has been a successful technique to find *good* enough solutions to NP-hard problems. Specifically, genetic programming was developed to solve the restrictions of genetic algorithms [15]. Instead of representing each

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

solution as a vector, each element of the solutions space is coded as a tree, with no size limitations. In order to ensure that these trees always represent feasible solutions, the *grammar-based genetic programming* paradigm was proposed [18]. Trees according to this paradigm are produced as derivations from a grammar that is specifically designed so that every solution is feasible. This approach has been used to search for neural net structures [4], Bayesian network structures [21] and rule-based systems [17]. Another field that has been applied to find *good* test suites is Information Theory [6]. There are several approaches proposing different measures to select the *better* test suites from a given set and comparing them [9,11] and the ones that show better performance are measures based on Information Theory.

In order to find the best test suite, up to a given size, to test a certain system we confront the classical combinatorial explosion: we have to check how good are all the subsets (up to a given size) of the set of available test cases. Previous work has automatically generated test suites (or test cases) using genetic algorithms [7,8,16,22]. Most of these approaches usually consider the basic conception of a genetic algorithm, with the risk of losing the correctness of the test suite, or needing to introduce some special items (as a *do not care element* [10]) in order to preserve it. Finally, although there are some work using genetic programming, we are not aware of any work that uses genetic programming ensuring at the same time the correctness and length of the test suite.

In this paper we propose a genetic programming algorithm to generate test suites, guided by a grammar that ensures their correctness. The algorithm generates test suites that get a *good* score for a given measure, and with a fixed length, in order to avoid the generation of extremely long and computationally heavy (or short and useless) test suites. This algorithm is supported by a tool that implements it, using already proved Information Theory based measures. This tool also allows users to compare the test suites generated by the algorithm using two different measures and see how well each of them performs. Finally, the tool allows to include measures defined by the user.

The rest of the paper is organized as follows. In Section 2 we introduce the main concepts that we will use along the paper. In Section 3 we introduce the core of our genetic programming algorithm. In Section 4 we present the main features of our tool and the results of an experiment. Finally, in Section 5 we give the conclusions of our work.

2 Preliminaries

In this section we present the main definitions and concepts that we use throughout this paper. Most of the concepts are based on the classical notions while some notation is adapted to facilitate the formulation of subsequent definitions.

Given a set A, we let:

- A^* denote the set of finite sequences of elements of A.
- $\epsilon \in A^*$ denote the empty sequence.
- A^+ denote the set of non-empty sequences of elements of A.

– $|A|$ denote the cardinal of set A.

Given a sequence $\sigma \in A^*$, we have that $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a.

Throughout this paper we let I be the set of input actions and O be the set of output actions. In our context an input of a system will be a non-empty sequence of input actions, that is, an element of I^+ (similarly for outputs and output actions).

A *Finite State Machine* is a (finite) labelled transition system in which transitions are labelled by an input/output pair. We use this formalism to define specifications.

Definition 1. *We say that $M = (Q, q_{in}, I, O, T)$ is a* Finite State Machine *(`FSM`), where Q is a finite set of states, $q_{in} \in Q$ is the initial state, I is a finite set of inputs, O is a finite set of outputs, and $T \subseteq Q \times (I \times O) \times Q$ is the transition relation. A transition $(q, (i, o), q') \in T$, also denoted by $q \xrightarrow{i/o} q'$ or by $(q, i/o, q')$, means that from state q after receiving input i it is possible to move to state q' and produce output o.*

We say that M is deterministic *if for all $q \in Q$ and $i \in I$ there exists at most one pair $(q', o) \in Q \times O$ such that $(q, i/o, q') \in T$.*

We say that M is input-enabled *if for all $q \in Q$ and $i \in I$ there exists $(q', o) \in Q \times O$ such that $(q, i/o, q') \in T$.*

We let `FSM`(I, O) denote the set of finite state machines with input set I and output set O.

In this paper we assume that `FSM`s are deterministic. We make this assumption because most Information Theory measures are applied to code and code is usually deterministic. We do not impose that `FSM`s are input-enabled. We will assume the *test hypothesis* [14]: the *System Under Test* (`SUT`) can be modelled as an object described in the same formalism as the specification (in our case, an `FSM`). Note that we do not need to have access to this description; we are indeed in a black-box testing framework because we only assume the existence of such `FSM`. Actually, it would be enough to assume that each time that the `SUT` receives a sequence of input actions, it returns a sequence of output actions. As usual, we do need access to the specification.

Our main goal while testing is to decide whether the behaviour of an `SUT` conforms to the specification of the system that we would like to build. In order to detect differences between specifications and `SUT`s, we need to compare the behaviours of specifications and `SUT`s and the main notion to define such behaviours is given by the concept of *trace*.

Definition 2. *Let $M = (Q, q_{in}, I, O, T)$ be an `FSM`, $q \in Q$ be a state and $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^*$ be a sequence of pairs. We say that M can perform σ from q if there exist states $q_1 \dots q_k \in Q$ such that for all $1 \leq j \leq k$ we have $(q_{j-1}, i_j/o_j, q_j) \in T$, where $q_0 = q$. We denote this by either $q \overset{\sigma}{\Longrightarrow} q_k$ or $q \overset{\sigma}{\Longrightarrow}$. If $q = q_{in}$ then we say that σ is a* trace *of M. We denote by `traces`(M) the set of* *traces of M. Note that for every state q we have that $q \overset{\epsilon}{\Longrightarrow} q$ holds. Therefore, $\epsilon \in \texttt{traces}(M)$ for every FSM M.*

Using the notion of trace, we can introduce the notion of test: a test is a sequence of (input action, output action) pairs. A test suite will be a set of tests.

Definition 3. *Let $M = (Q, q_{in}, I, O, T)$ be an FSM. We say that a sequence $t = (i_1, o_1) \dots (i_k, o_k) \in (I \times O)^+$ is a* test *for M if $t \in \texttt{traces}(M)$. We define the* length *of t as the length of the sequence, that is, $|t| = k$. We define the sequence of inputs of t as $\alpha = i_1 \dots i_k$ and the sequence of outputs of t as $\beta = o_1 \dots o_k$ (we will sometimes use the notation $t = (\alpha, \beta) \in (I^+ \times O^+)$). A* test suite *for M is a set of tests for M. Given a test suite $\mathcal{T} = \{t_1, \dots, t_n\}$, we define the* length *of the test suite as the sum of the lengths of its tests, that is, $|\mathcal{T}| = \sum_{i=1,\dots,n} |t_i|$.*

Let $t = (\alpha, \beta)$ be a test for M. We say that the application of t to an FSM M' fails if there exists β' such that $(\alpha, \beta') \in \texttt{traces}(M')$ and $\beta \neq \beta'$. Similarly, let $\mathcal{T}$ be a test suite for M. We say that the application of $\mathcal{T}$ to an FSM M' fails if there exists $t \in \mathcal{T}$ such that the application of t to M' fails.

Intuitively, a test (α, β) for M denotes that the application of the sequence of input actions α to a correct system (with respect to M) should show the sequence of output actions β. Note that if we would allow non-determinism, then the previous inequality must be appropriately replaced to express that the behaviours of the SUT must be a subset of those of the specification. For now, we will assume the determinism of the FSMs.

In order to select the tests that can detect the higher amount of fails in the program, it is useful to have a *measure* on the goodness of a test suite. Let us emphasize that measures will be, in general, heuristics to find good solutions and that each measure should be validated with experiments. Usually, higher values of a measure will be associated with better solutions, but this relation need not be monotonic. The measures that we use in this paper have been introduced in previous work and it has been shown that they are useful to find good test suites. We introduce a general notion of measure.

Definition 4. *A* measure *is a function*

$$f : \mathit{FSM}(I, O) \times \mathcal{P}(I^+ \times O^+) \to \mathbb{R}^+ \cup \{0\}$$

Intuitively, a measure is a function that receives an FSM and a test suite and returns a real number representing how good the measure considers that this test suite is to detect fails in an SUT. This notion of measure allows us to use information both from the specification and the test suite that we are evaluating, although it not necessarily has to use information from both, that is, a measure could work only with the information from the test suite and not use the specification at all. Finding the best test suite according to a measure (that is, the test suite that gets the best score) is usually an NP-hard problem (due to the combinatorial explosion). Therefore, we decided to rely on genetic programming in order to obtain *relatively good* test suites. A genetic algorithm is composed by:

```
Initialize population;
Evaluate population;
while termination criterion not reached do
    Select next population;
    Perform crossover;
    Perform mutation;
    Evaluate population;
end
```

Algorithm 1: Genetic algorithm: general scheme

- An encoding of the population in *genes*.
- An *initial population*, that is, randomly generated individuals expressed in the selected codification.
- A *fitness function* to evaluate the population.
- A *stopping criterion*.
- A *next population selection method*, which usually keeps the best individuals and discards the worst ones (with respect to the fitness function values).
- A *crossover method* that generates new individuals from the mixture of the genes of the existing ones.
- A *mutation method* that can modify some individuals in order to obtain new genes that might have not been present before.

The structure of a genetic algorithm is given in Algorithm 1. A basic genetic programming algorithm is a genetic algorithm where the codification of the population in genes does not use a linear structure (as a vector) but a tree-like structure [15]. Most of the work using genetic algorithms to generate test suites rely on a linear structure to represent the test suite. Specially, they use to rely on a vector of the inputs of the test suite [7,8,16,22]. This encoding of a test suite presents a problem: if the `FSM` is not input-enabled, then the algorithm could generate invalid tests that will always fail when applied to the `SUT`, even if this is totally equivalent to the `FSM`. As we are working with deterministic but not necessarily input-enabled `FSM`s, we have to face this problem and using a grammar-guided genetic programming algorithm allows us to ensure the correctness of the generated test suites. This approach also allows us to use the information from the output that each input generates in each state of the `FSM` (as the inputs do not have to generate the same output in all the states).

3 The Genetic Programming algorithm

In this section, we will present all the components of our genetic algorithm.

3.1 Encoding

The first and most important choice of a genetic approach is to select a good encoding. As we are working with test suites generated from an `FSM`, we need to preserve the structure of the FSM in order to generate correct tests for it. Therefore, we decided to use a tree structure as an encoding of our tests and we use a genetic programming algorithm. Specifically, we decided to use a grammar-guided genetic programming approach, which solves the correctness issues from just using genetic programming. This implies that the first step of our genetic programming algorithm will be to generate the grammar that the FSM produces. We have the following components:

- A start non-terminal symbol S that starts the grammar.
- A non-terminal symbol T that introduces each test of the test suite.
- A non-terminal symbol N for each state, where $N \in \mathbb{N}$ is the state number.
- A terminal symbol $'a/b'$ for each input/output pair present on the FSM, where a is the input and b is the output.
- A terminal symbol $'null'$ to represent the end of a test.
- A production rule $S \longrightarrow T$ to generate the initial test.
- A production rule $T \longrightarrow T + T$ to introduce a new test.
- A production rule $T \longrightarrow\ 0$ to start each test in the FSM initial state.
- A production rule $N \longrightarrow\ 'a/b' + M$ for each transition from the state N to a state M with input/output pair (a, b).
- A production rule $N \longrightarrow' null'$ for each state N to a terminal to represent the end of the test.

Given an FSM, the generation of the associated grammar is automatic (and it has been implemented as part of our tool).

3.2 Initial population

As an initial population we randomly generate 100 test suites of the length given by the user using the grammar previously derived from the FSM. Each rule in the grammar has the same probability of being triggered. This allows a uniform random initialization.

3.3 Fitness function

The fitness function of our genetic programming algorithm will be the available measures. As previously defined, they will receive the test suite and the FSM and will return a real value that represents how *good* is this test suite according to the measure. An important remark about fitness functions is that they should be easy to compute, as they will be invoked many times during the execution of the algorithm. Therefore, fitness functions with high computational cost will lead to a higher computational cost of the algorithm.

We decided to give the users the capability to select the fitness function that better suites their problem, along with the decision on whether the score should be maximized or minimized. As we explained before, fitness functions should have similar performances and this is the case for the measures based on Information Theory that we include in our tool. Among them, we can mention, due to the big improvement with respect to previous measures, the *Test Set Diameter* (TSDm) based measures [9]. We implemented the Input-TSDm, the Output-TSDm and the InputOutput-TSDm. Also, we implemented a measure that we have developed in our research group and that it is called *Biased Mutual Information*. Note that users of our tool can add their own measures. So, our tool can be use to evaluate the usefulness of new proposals because they can be compared with existing ones.

3.4 Stopping criterion

The algorithm performs at most 100 epochs and at least 20 epochs. Once we have passed the 20 epochs, the stop criterion will be fulfilled if the best test suite is the same along $0.2 \times \mathit{NumberOfPassedEpochs}$ epochs.

3.5 Selection method

We use a variant of elitist reduction [19]. First, the test suites that got a fitness score over the mean (or under the mean if we want to minimize) go directly to the next epoch. In addition, the ones that are under the mean can pass to the next epoch if their score is higher than the mean minus a random number modulo the distance between the mean and the best score.

3.6 Crossover method

The choice of crossover method depends on our encoding and the characteristics we want the produced test suites to have. As we use a grammatical encoding, we need to use a grammatical crossover. We have considered a mixture between the Whigham crossover [18] and the standard grammatical crossover [5]. Also, as we want all our test suites to have the same length (as previously defined), we need to slightly modify crossovers in order to achieve a crossover that keeps the length fixed. Algoritm 2 shows how crossover is performed.

Finally, we need to set the probability of producing the crossover. In our case, giving how hard is to perform a crossover, we decided to set this probability to 90%, so that we favour the mixture between test suites.

3.7 Mutation method

A mutation consists in generating a new test with the same length. The probability of performing a mutation will be, as usual [19], equal to 5% for each test of each test suite of the population.

4 GPTSG

We have implemented a tool[3] supporting our framework. The tool has two main uses: generate a test suite with a giving length according to a selected measure

[3] The tool can be downloaded from https://github.com/Colosu/gptsg.

```
Data: $TS1$, $TS2$ test suites
Result: Crossover of $TS1$ and $TS2$
$match = false$;
while $!match$ do
    Select a random node $t1$ from $TS1$;
    for each node $t2$ of $TS2$ do
        if $t2$ non-terminal $==$ $t1$ non-terminal and $t2$ length $==$ $t1$ length then
            Set $t2$ as valid node.
        end
    end
    if valid nodes $> 0$ then
        $match = true$;
    end
end
Select a random valid node $t2$;
Get parent $p1$ of $t1$;
Get parent $p2$ of $t2$;
Set $t2$ as child of $p1$;
Set $t1$ as child of $p2$;
```

Algorithm 2: Crossover algorithm

and compare different measures. In order to develop the tool, we looked for libraries dealing with FSMs and we decided to use the OpenFST library [1]. Therefore, input files must be in OpenFST format, with the .fst extension. The tool will have two kind of calls *generate* and *compare*. The syntax of the two calls is:

```
gptsg generate inputFile length {max|min} fitness
gptsg compare length {max|min} fitness {max|min} fitness
```

and two examples of calls are:

```
gptsg generate ./test/binary.fst 50 max ITSDm
gptsg compare 50 max ITSDm min OTSDm
```

Currently, our tool supports the following fitness functions:

– BMI: Biased Mutual Information.
– ITSDm: Input Test Set Diameter.
– OTSDm: Output Test Set Diameter.
– IOTSDm: Input-Output Test Set Diameter.
– Own: For your own developed measure.
– random: generates a totally random test suite.

Let us emphasize that an important feature of our tool is that it is possible to define new measures, so that they can be compared with the already existing ones. The user only needs to open the `src/Measures.cpp` file and modify the `OwnFunction` method. Once the code is compiled, the inserted measure can be called as the *Own* fitness function.

4.1 Test suite generation

In order to generate a *good* enough test suite, we implemented the genetic programming algorithm explained in the previous section, giving some configuration to the user. The tool needs that the input `FSM` is in OpenFST format (in a .fst file). This format is easy to use and can be learned quickly. Also, the tool needs to know the length of the expected test suite, in terms of input actions, and the measure to use as a fitness function. Then, the user will receive a .txt file with the generated test suite, with each test conformed by a succession of input/output pairs.

4.2 Test suite comparison

The tool allows users to compare two measures. It needs to know the length of the desired test suite, the two measures to be compared and if it should maximize or minimize each measure. Essentially, the tool takes the set of 100 `FSM`s that are shipped with the tool, representing different and diverse scenarios and characteristics, and for each one of them it generates two test suites according to the corresponding measures. Then, the tool produces 1000 mutants of the corresponding `FSM` and checks which test suite kills more mutants. With the results for each `FSM`, the tool produces an output telling the percentage of cases where each test suite has killed more mutants, along with a percentage of how many mutants where killed by each test suite. This process is repeated 50 times, getting 50 results, and at the end, the program gives a mean of all the results obtained for the 50 repetitions. This process is given in Algorithm 3.

4.3 Experiment

Next we show the results of a small experiment to evaluate our genetic programming algorithm and tool. First, we compared the Input Test Set Diameter measure, used as fitness function, and a random test suite generation. We observed that the genetically generated test suite killed more mutants than the randomly generated test suite in a 75.3% of the cases, killing an average of 47.1% of the mutants, while the randomly generated test suite killed more mutants the 24.7% of the cases, killing an average of 43.9% of the mutants. We can see the full comparison in Figure 1 (left), where each of the first 50 rows shows the result of an iteration of the experiment. In order to see how two measures are compared, we rerun the comparison algorithm to compare the maximization of the Input Test Set Diameter and the maximization of the Output Test Set Diameter. The results can be seen in Figure 1 (right). As expected, they obtain similar results, getting better results the Output TSDm due to the randomization involved in the genetic algorithm. On average, the Input TSDm killed more mutants the 49% of the cases, killing 47.3% of the mutants, while Output TSDm killed more mutants the 51% of the cases, killing 47.5% of the mutants.

```
Data: $length, measure1, measure2$
Result: .txt file with the values
$REP = 50$;
FSM $= 100$;
for each $REP$ do
    Set control values to 0;
    for each FSM $F$ do
        Generate $TS1$ genetic test suite using measure $measure1$;
        Generate $TS2$ genetic test suite using measure $measure2$;
        Generate 1000 mutants of $F$;
        Check which test suite kills more mutants;
    end
    Output the percentage of runs $TS1$ killed more mutants;
    Output the percentage of runs $TS2$ killed more mutants;
    Output the percentage of mutants killed by $TS1$;
    Output the percentage of mutants killed by $TS2$;
end
Output the average percentage of runs $TS1$ killed more mutants;
Output the average percentage of runs $TS2$ killed more mutants;
Output the average percentage of mutants killed by $TS1$;
Output the average percentage of mutants killed by $TS2$;
```

Algorithm 3: Test suite comparison algorithm

5 Conclusions

The automatic generation of good test suites is a fundamental task when limitations in testing complex systems come into play. In this paper we have presented a genetic programming algorithm to generate these test suites. We have implemented a tool to support our algorithm so that any potential user can apply it. The tool allows users to compare genetically generated test suites that outperform different measures, so that we can compare the performance of each measure. We have relied on Information Theory to define the measures that will work as fitness functions of our genetic programming algorithm. Finally, we have performed several experiments with our tool and report on two of them. There are some lines for future work.

We are considering several lines of future work. First, it would be interesting to find and define new measures so that we can extend the catalogue of our tool. Second, we are working on extending our algorithm to deal with the generation of test suites to test systems with distributed interfaces [12,13].

References

1. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: A general and efficient weighted finite-state transducer library. In: 9th Int. Conf. on Implementation and Application of Automata, CIAA'07, LNCS 4783. vol. 4783, pp. 11–23. Springer (2007)

Iteration Number	% wins ITSDm	% wins random	% mutants killed by ITSDm	% mutants killed by random
1	0.757576	0.242424	0.475081	0.439909
2	0.744898	0.255102	0.476306	0.443786
3	0.75	0.25	0.46496	0.43275
4	0.8	0.2	0.47095	0.43229
5	0.75	0.25	0.46347	0.43407
6	0.69	0.31	0.46947	0.4419
7	0.72449	0.27551	0.475051	0.447755
8	0.72449	0.27551	0.475051	0.447857
9	0.74	0.26	0.46035	0.43142
10	0.767677	0.232323	0.471525	0.441646
11	0.72449	0.27551	0.473204	0.443663
12	0.767677	0.232323	0.470525	0.441
13	0.646465	0.353535	0.465929	0.440162
14	0.717172	0.282828	0.468394	0.437384
15	0.81	0.19	0.47143	0.43202
16	0.686869	0.313131	0.469606	0.441101
17	0.76	0.24	0.46867	0.43234
18	0.81	0.19	0.47033	0.43285
19	0.79798	0.20202	0.468172	0.438222
20	0.83	0.17	0.46732	0.43214
21	0.77	0.23	0.46903	0.43535
22	0.757576	0.242424	0.473949	0.440232
23	0.78	0.22	0.46716	0.43258
24	0.744898	0.255102	0.475673	0.442582
25	0.79798	0.20202	0.475343	0.446455
26	0.79	0.21	0.4678	0.43391
27	0.68	0.32	0.46184	0.43662
28	0.79	0.21	0.46923	0.43317
29	0.75	0.25	0.4663	0.43457
30	0.79	0.21	0.46861	0.43607
31	0.747475	0.252525	0.474848	0.436535
32	0.663265	0.336735	0.469214	0.445235
33	0.73	0.27	0.46731	0.43186
34	0.721649	0.278351	0.478278	0.449763
35	0.72	0.28	0.46624	0.43619
36	0.79798	0.20202	0.472141	0.439152
37	0.783505	0.216495	0.48668	0.450495
38	0.75	0.25	0.46744	0.43613
39	0.767677	0.232323	0.465687	0.432909
40	0.676768	0.323232	0.469	0.443343
41	0.742268	0.257732	0.484216	0.450979
42	0.75	0.25	0.46626	0.43473
43	0.77551	0.22449	0.477082	0.443643
44	0.77	0.23	0.46771	0.43251
45	0.76	0.24	0.47459	0.43538
46	0.757576	0.242424	0.471333	0.439051
47	0.744898	0.255102	0.479857	0.444378
48	0.806122	0.193878	0.478143	0.437531
49	0.777778	0.222222	0.473505	0.440222
50	0.74	0.26	0.46422	0.43529
Mean	0.752723	0.247277	0.470851	0.438578

Iteration Number	% wins ITSDm	% wins OTSDm	% mutants killed by ITSDm	% mutants killed by OTSDm
1	0.494845	0.505155	0.484216	0.48366
2	0.546392	0.453608	0.481206	0.479309
3	0.44	0.56	0.46812	0.47277
4	0.525253	0.474747	0.473485	0.468192
5	0.443299	0.556701	0.481979	0.489619
6	0.515152	0.484848	0.474323	0.473404
7	0.525253	0.474747	0.477051	0.474596
8	0.56	0.44	0.4725	0.4662
9	0.48	0.52	0.46631	0.47339
10	0.45	0.55	0.46801	0.47551
11	0.515152	0.484848	0.469535	0.472646
12	0.489796	0.510204	0.476612	0.478694
13	0.51	0.49	0.46966	0.46947
14	0.40404	0.59596	0.468949	0.478323
15	0.47	0.53	0.46272	0.46797
16	0.52	0.48	0.46723	0.46533
17	0.367347	0.632653	0.468765	0.480704
18	0.535354	0.464646	0.471131	0.471232
19	0.428571	0.571429	0.47351	0.48201
20	0.51	0.49	0.46843	0.47081
21	0.561224	0.438776	0.483551	0.479612
22	0.43	0.57	0.46509	0.46947
23	0.414141	0.585859	0.470253	0.475808
24	0.555556	0.444444	0.473596	0.469566
25	0.494949	0.505051	0.474384	0.475111
26	0.52	0.48	0.46671	0.46436
27	0.505155	0.494845	0.480866	0.481639
28	0.515152	0.484848	0.472889	0.473586
29	0.469388	0.530612	0.474306	0.476204
30	0.545455	0.454545	0.477	0.473242
31	0.510204	0.489796	0.476571	0.477184
32	0.51	0.49	0.46423	0.46837
33	0.469388	0.530612	0.477316	0.479204
34	0.408163	0.591837	0.475296	0.484643
35	0.545455	0.454545	0.468455	0.471949
36	0.494949	0.505051	0.475091	0.475495
37	0.5	0.5	0.46762	0.46903
38	0.46	0.54	0.46477	0.47045
39	0.474227	0.525773	0.48034	0.484113
40	0.45	0.55	0.46602	0.47369
41	0.489796	0.510204	0.480306	0.480286
42	0.546392	0.453608	0.484423	0.484804
43	0.51	0.49	0.46951	0.46428
44	0.47	0.53	0.46927	0.47261
45	0.469388	0.530612	0.478684	0.477918
46	0.43	0.57	0.46678	0.47126
47	0.438776	0.561224	0.479082	0.480398
48	0.515464	0.484536	0.479969	0.483546
49	0.56	0.44	0.4709	0.46138
50	0.484848	0.515152	0.472162	0.471455
Mean	0.489581	0.510419	0.47293	0.474633

Fig. 1. Results of ITSDm vs random (left) and ITSDm vs OTSDm (right).

2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, 2nd edn. (2017)
3. Cavalli, A.R., Higashino, T., Núñez, M.: A survey on formal active and passive testing with applications to the cloud. Annales of Telecommunications **70**(3-4), 85–93 (2015)
4. Couchet, J., Manrique, D., Porras, L.: Grammar-guided neural architecture evolution. In: 2nd Int. Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC'07, LNCS 4527. pp. 437–446. Springer (2007)
5. Couchet, J., Manrique, D., Rios, J., Rodríguez-Patón, A.: Crossover and mutation operators for grammar-guided genetic programming. Soft Computing **11**(10), 943–955 (2007)
6. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley, 2nd edn. (2006)

7. Derderian, K., Merayo, M.G., Hierons, R.M., Núñez, M.: Aiding test case generation in temporally constrained state based systems using genetic algorithms. In: 10th Int. Conf. on Artificial Neural Networks, IWANN'09, LNCS 5517. pp. 327–334. Springer (2009)
8. 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)
9. Feldt, R., Poulding, S.M., Clark, D., Yoo, S.: Test set diameter: Quantifying the diversity of sets of test cases. In: 9th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'16. pp. 223–233. IEEE Computer Society (2016)
10. Guo, Q., Hierons, R.M., Harman, M., Derderian, K.: Computing Unique Input/Output sequences using genetic algorithms. In: 3rd Int. Workshop on Formal Approaches to Testing of Software, FATES'03, LNCS 2931. pp. 164–177. Springer (2003)
11. Henard, C., Papadakis, M., Harman, M., Jia, Y., Traon, Y.L.: Comparing white-box and black-box test prioritization. In: 38th Int. Conf. on Software Engineering, ICSE'14. pp. 523–534. ACM Press (2016)
12. Hierons, R.M., Merayo, M.G., Núñez, M.: Bounded reordering in the distributed test architecture. IEEE Transactions on Reliability **67**(2), 522–537 (2018)
13. Hierons, R.M., Núñez, M.: Implementation relations and probabilistic schedulers in the distributed test architecture. Journal of Systems and Software **132**, 319–335 (2017)
14. ISO/IEC JTCI/SC21/WG7, ITU-T SG 10/Q.8: Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T (1996)
15. Koza, J.R.: Genetic programming. MIT Press (1993)
16. Lefticaru, R., Ipate, F.: Automatic state-based test generation using genetic algorithms. In: 9th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC'07. pp. 188–195. IEEE Computer Society (2007)
17. Luna, J.M., Romero, J.R., Ventura, S.: Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules. Knowledge and Information Systems **32**(1), 53–76 (2012)
18. McKay, R.I., Hoai, N.X., Whigham, P.A., S., Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genetic Programming and Evolvable Machines **11**(3-4), 365–396 (2010)
19. Mitchell, M.: An introduction to genetic algorithms. MIT Press (1998)
20. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. John Wiley & Sons, 3rd edn. (2011)
21. Nunes Regolin, E., Ramirez Pozo, A.T.: Bayesian automatic programming. In: 8th European Conference on Genetic Programming, EuroGP'05, LNCS 3447. pp. 38–49. Springer (2005)
22. Samarah, A., Habibi, A., Tahar, S., Kharma, N.N.: Automated coverage directed test generation using a cell-based genetic algorithm. In: 11th Annual IEEE Int. High-Level Design Validation and Test Workshop. pp. 19–26. IEEE Computer Society (2006)
23. Shafique, M., Labiche, Y.: A systematic review of state-based test tools. International Journal on Software Tools for Technology Transfer **17**(1), 59–76 (2015)