Balera and Santiago Júnior *Journal of Software Engineering Research and Development* (2017) 5:10
DOI 10.1186/s40411-017-0043-z

Journal of Software Engineering Research and Development

RESEARCH Open Access

An algorithm for combinatorial interaction testing: definitions and rigorous evaluations

Juliana M. Balera* and Valdivino A. de Santiago Júnior

*Correspondence:
juliana.balera@inpe.br
Laboratório Associado de Computação e Matemática Aplicada, Instituto Nacional de Pesquisas Espaciais (INPE), Av. dos Astronautas, 1758, São José dos Campos, SP, Brazil

Abstract

Background: Combinatorial Interaction Testing (CIT) approaches have drawn attention of the software testing community to generate sets of smaller, efficient, and effective test cases where they have been successful in detecting faults due to the interaction of several input parameters. Recent empirical studies show that greedy algorithms are still competitive for CIT. It is thus interesting to investigate new approaches to address CIT test case generation via greedy solutions and to perform rigorous evaluations within the greedy context.

Methods: We present a new greedy algorithm for unconstrained CIT, T-Tuple Reallocation (TTR), to generate CIT test suites specifically via the Mixed-value Covering Array (MCA) technique. The main reasoning behind TTR is to generate an MCA *M* by creating and reallocating t-tuples into this matrix *M*, considering a variable called *goal* (ζ). We performed two controlled experiments addressing cost-efficiency and only cost. Considering both experiments, we did 3200 executions related to 8 solutions. In the first controlled experiment, we compared versions 1.1 and 1.2 of TTR in order to check whether there is significant difference between both versions of our algorithm. In such experiment, we jointly considered cost (size of test suites) and efficiency (time to generate the test suites) in a multi-objective perspective. In the second controlled experiment we confronted TTR 1.2 with five other greedy algorithms/tools for unconstrained CIT: IPOG-F, jenny, IPO-TConfig, PICT, and ACTS. We performed two different evaluations within this second experiment where in the first one we addressed cost-efficiency (multi-objective) and in the second only cost (single objective).

Results: Results of the first controlled experiment indicate that TTR 1.2 is more adequate than TTR 1.1 especially for higher strengths (5, 6). In the second controlled experiment, TTR 1.2 also presents better performance for higher strengths (5, 6) where only in one case it is not superior (in the comparison with IPOG-F). We can explain this better performance of TTR 1.2 due to the fact that it no longer generates, at the beginning, the matrix of t-tuples but rather the algorithm works on a t-tuple by t-tuple creation and reallocation into *M*.

Conclusion: Considering the metrics we defined in this work and based on both controlled experiments, TTR 1.2 is a better option if we need to consider higher strengths (5, 6). For lower strengths, other solutions, like IPOG-F, may be better alternatives.

Keywords: Software testing, Combinatorial interaction testing, Combinatorial testing, Mixed-value covering array, T-Tuple reallocation, Controlled experiment

1 Introduction

The academic community has been making efforts to reduce the cost of the software testing process by decreasing the size of test suites while at the same time aiming at maintaining the effectiveness (ability to detect defects) of such sets of test cases. Hence, several contributions exist for test suite/case minimization (Yoo and Harman 2012; Ahmed 2016; Huang et al. 2016; Khan et al. 2016) where the goal is to decrease the size of a test suite by eliminating redundant test cases, and hence demanding less effort to execute the test cases (Yoo and Harman 2012). One of the approaches to reduce the number of test cases is *Combinatorial Interaction Testing* (CIT) (Petke et al. 2015), also known as *Combinatorial Testing* (CT) (Kuhn et al. 2013; Schroeder and Korel 2000), *Combinatorial Test Design* (CTD) (Tzoref-Brill et al. 2016), or *Combinatorial Designs* (CD) (Mathur 2008). CIT relates to combinatorial analysis whose objective is to answer whether it is possible to organize elements of a finite set into subsets so that certain balance or symmetry properties are satisfied (Stinson 2004).

There are reports which claim the success of CIT (Dalal et al. 1999; Tai and Lei 2002; Kuhn et al. 2004; Yilmaz et al. 2014; Qu et al. 2007; Petke et al. 2015). Such approaches have drawn attention of the software testing community to generate sets of smaller (lower cost to run) and effective (greater ability to find faults in the software) test cases where they have been successful in detecting faults due to the interaction of several input parameters (factors).

CIT approaches to generate test cases can be divided in four main classes: Binary Decision Diagrams (BDDs) (Segall et al. 2011), Satisfiability (SAT) solving (Cohen et al. 1997; Yamada et al. 2015; Yamada et al. 2016), meta-heuristics (Garvin et al. 2011; Shiba et al. 2004; Hernandez et al. 2010), and greedy algorithms (Lei and Tai 1998; Lei et al. 2007)[1]. Recent CIT test case generation methods based on BDD and SAT are interesting to constrained (there are restrictions related to parameter interactions) problems but they perform worse compared with greedy algorithms/tools in the context of unconstrained (there are no restrictions at all) problems.

To corroborate this claim, in (Segall et al. 2011) a BDD-based approach, implemented in the Focus tool, was better in terms of cost than the greedy solutions Advanced Combinatorial Testing System (ACTS) (Yu et al. 2013), Pairwise Indepedent Combinatorial Testing (PICT) (Czerwonka 2006), and jenny (Jenkins 2016) in the constrained domain. However, their method was worse than such greedy solutions for unconstrained problems.

A recent SAT-based approach (Yamada et al. 2016), implemented in the Calot tool, performed well in terms of efficiency (time to generate the test suites) and cost (test suite sizes) comparing again with the greedy tools ACTS (Yu et al. 2013) and PICT (Czerwonka 2006). Despite the advantages of the SAT-based approach, ACTS was much more faster than Calot for many 3-way test case examples. Moreover, if unconstrained CIT is considered, ACTS again was remarkable faster than Calot for large SUT models and higher-strength test case generation.

In the context of CIT, meta-heuristics such as simulated annealing (Garvin et al. 2011), genetic algorithms (Shiba et al. 2004), and Tabu Search Approach (TSA) (Hernandez et al. 2010) have been used. Recent empirical studies show that meta-heurisitic and greedy algorithms have similar performance (Petke et al. 2015). Hence, early fault detection via a greedy algorithm with constraint handling (implemented in the ACTS tool (Yu et al. 2013)) was no worse than a simulated annealing algorithm (implemented in the CASA

tool (Garvin et al. 2011)). Moreover, there was not enough difference between test suites generated by ACTS and CASA in terms of efficiency (runtime) and t-way coverage. All such previous remarks, some of them based on strong empirical evidences, emphasize that greedy algorithms are still very competitive for CIT.

Even if some authors have argued that CIT resides in the constrained domain in real-world applications (Bryce and Colbourn 2006; Cohen et al. 2008; Petke et al. 2015), it is important to mention that unconstrained CIT may be interesting from a practical point of view, especially for critical applications such as satellites, rockets, airplanes, controllers of an unmanned train metro system, etc. For such types of applications, robustness testing is very important. In the context of software systems, robustness testing aims to verify whether the Software Under Test (SUT) behaves correctly in the presence of invalid inputs. Therefore, even though an unconstrained CIT-derived test case may seem pointless or even somewhat difficult to execute, it may still be interesting to see how the software will behave in the presence of inconsistent inputs.

Let us consider that we need to test a communication protocol implemented in several critical embedded systems. If each field of such a protocol is a parameter, it is interesting to impose no restriction (no constraint) in the parameter interactions so that a certain Protocol Data Unit (PDU) sent from system A to system B may have values not allowed in the combination of the fields (parameters) of the PDU. In other words, if the specification says that when field $f_i = 1$, possible values of field f_j are between 20 and 70 ($20 \leq f_j \leq 70$), and other field $f_k < 5$, then a test case where $f_i = 1$, $1 \leq f_j \leq 4$, and $f_k < 5$ is clearly inconsistent because of the value of f_j. But, this can precisely the goal of the test designer because he/she wants to check how the receiving system (B) will act upon receiving a PDU like that from A. This is an example where unconstrained CIT is relevant. It is important to mention that the argument is not that constraints can not be used for testing critical systems but rather that, for certain types of tests (robustness), constraints are not as relevant.

Based on the context and motivation previously presented, this research relates to greedy algorithms for unconstrained CIT. In (Pairwise 2017), 43 algorithms/tools are presented for CIT and many more not shown there exist. Some of these solutions are variations of the In-Parameter-Order (IPO) algorithm (Lei and Tai 1998) such as IPOG, IPOG-D (Lei et al. 2007), IPOG-F, IPOG-F2 (Forbes et al. 2008), IPOG-C (Yu et al. 2013), IPO-TConfig (Williams 2000), ACTS (where IPOG, IPOG-D, IPOG-F, IPOG-F2 are implemented) (Yu et al. 2013), and CitLab (Cavalgna et al. 2013). All IPO-based proposals have in common the fact that they perform horizontal and vertical growths to construct the final test suite. Moreover, some need two auxiliary matrices which may decrease its performance by demanding more computer memory. Such algorithms accomplish exhaustive comparisons within each horizontal extension which may penalize efficiency.

PICT can be regarded as one baseline tool where other approaches have been done based on it (PictMaster 2017). The algorithm implemented in this tool works in two phases, the first being the construction of all t-tuples to be covered. This can often be a not interesting solution since many t-tuples may require large disk space for storage.

Thus, it is interesting to think about a new greedy solution for CIT that does not need, at the beginning, to enumerate all t-tuples (such as PICT) and does not demand many auxiliary matrices to operate (as some IPO-based approaches). Although we have some recent rigorous empirical evaluations comparing greedy algorithms with meta-

heuristics solutions (Petke et al. 2015) and greedy approaches against SAT-based methods (Yamada et al. 2016), there are no rigorous empirical assessments comparing greedy algorithms/tools, representative of the unconstrained CIT domain, among each other.

In this paper, we present a new algorithm, called T-Tuple Reallocation (TTR), to generate CIT test suites specifically via the Mixed-value Covering Array (MCA) technique. The main reasoning behind TTR is to generate an MCA M by creating and reallocating t-tuples into this matrix M, considering a variable called *goal* (ζ). TTR is a greedy algorithm for unconstrained CIT.

Three versions of the TTR algorithm were developed and implemented in Java. Version 1.0 is the original version of TTR (Balera and Santiago Júnior 2015). In version 1.1 (Balera and Santiago Júnior 2016), we made a change where we do not order the input parameters. In the last version, 1.2, the algorithm no longer generates the matrix of t-tuples (Θ) but rather it works on a t-tuple by t-tuple creation and reallocation into M. Moreover, version 1.2 was also implemented in C.

We performed two controlled experiments addressing cost-efficiency and only cost. Considering both experiments, we performed 3,200 executions related to 8 solutions. In the first controlled experiment, our goal was to compare versions 1.1 and 1.2 of TTR (in Java) in order to check whether there is significant difference between both versions of our algorithm. In such experiment, we jointly considered cost (size of test suites) and efficiency (time to generate the test suites) in a multi-objective perspective. We conclude that TTR 1.2 is more adequate than TTR 1.1 especially for higher strengths (5 and 6).

We then carried out a second controlled experiment where we confronted TTR 1.2 with five other greedy algorithms/tools for unconstrained CIT: IPOG-F (Forbes et al. 2008), jenny (Jenkins 2016), IPO-TConfig (Williams 2000), PICT (Czerwonka 2006), and ACTS (Yu et al. 2013). We performed two evaluations where in the first one we compared TTR 1.2 with IPOG-F and jenny since these were the solutions we had the source code (to precisely measure the time). Hence, a cost-efficiency (multi-objective) assessment was accomplished. In order to address a possible evaluation bias in the time measures due to different programming languages, we compared the implementation of TTR 1.2 (in Java) with IPOG-F (in Java), and the implementation of TTR 1.2 (in C) with jenny (in C). In the second assessment, we did a cost (single objective) evaluation where TTR 1.2 (Java) was compared with PICT, IPO-TConfig, and ACTS. The conclusion is the same as before: TTR 1.2 is better for higher strengths (5 and 6).

In this paper, we extend our previous works where we presented version 1.0 of TTR (Balera and Santiago Júnior 2015), and version 1.1 together with another controlled experiment (Balera and Santiago Júnior 2016). The contributions of this work are:

- Even though we considered version 1.1 of TTR in (Balera and Santiago Júnior 2016), we did not detail this version since the focus of this previous paper was this other controlled experiment. Thus, we highlight the key features of TTR 1.1 here;
- We created another version of our algorithm, 1.2, where, at the beginning, TTR does not generate the matrix of t-tuples. Our goal here is trying to avoid an exhaustive combination of t-tuples as might happen with other classical greedy approaches. Moreover, we rely on just one auxiliary matrix which is different from other greedy solutions which require two auxiliary matrices;

- We performed two controlled experiments in the unconstrained CIT domain (TTR 1.1 × TTR 1.2; TTR 1.2 × IPOG-F, jenny, IPO-TConfig, PICT, ACTS) with almost three times more participants, in each experiment, than in the previous one (Balera and Santiago Júnior 2016). In addition, we run each participant (instance) 5 times with different input orders of parameters and values to address the nondeterminism of the solutions. To the best of our knowledge, no previous research presented rigorous empirical evaluations for greedy solutions within the unconstrained CIT domain;
- We really accomplished a multi-objective (cost-efficiency) evaluation in both controlled experiments (in the second one, we did it in the first assessment). Previously (Balera and Santiago Júnior 2016), we analyzed cost and efficiency in isolation.

This paper is structured as follows. Section 2 presents an overview of the main concepts related to CIT. In Section 3, we show the main definitions and procedures of versions 1.1 and 1.2 of our algorithm. Section 4 shows all the details of the first controlled experiment when we compare TTR 1.1 against TTR 1.2. In Section 6, the second controlled experiment is presented where TTR is confronted with the other 5 greedy tools. Section 7 presents related work. In Section 8, we show the conclusions and future directions of our research.

2 Background

In this section we present some basic concepts and definitions (Kuhn et al. 2013; Petke et al. 2015; Cohen et al. 2003) related to CIT. A CIT algorithm receives as input a number of parameters (also known as factors), p, which refer to the input variables. Each parameter can assume a number of values (also known as levels) v. Moreover, t is the strength of the coverage of interactions. For example, in pairwise testing, the degree of interaction is two, so the value of strength is 2. In t-way testing, a t-tuple is an interaction of parameter values of size equal to the strength. Thus, a t-tuple is a finite ordered list of elements, i.e. it is a set of elements.

A Fixed-value Covering Array (CA) denoted by $\mathrm{CA}(N, p, v, t)$ is an $N \times p$ matrix of entries from the set $\{0, 1, \cdots, (v-1)\}$ such that every set of t-columns contains each possible t-tuple of entries at least a certain number of times (e.g. once). N is the number of rows of the array (matrix). Note that in a CA, entries are from the same set of v values.

A Mixed-value Covering Array $(\mathrm{MCA})^2$ it is an extension of a CA and it is more flexible because it allows parameters to assume values from different sets. Hence, it is represented as $\mathrm{MCA}\left(N, v_1^{p_1} v_2^{p_2} \ldots v_m^{p_m}, t\right)$, where N is the number of rows of the matrix, $\sum_{i=1}^{m} p_i$ is the number of parameters, each v_i is the number of values for each parameter p_i, and t is the strength.

Therefore, in CIT a CA or MCA is a test suite and each row of such matrices is a test case. Suppose that we need to generate a pairwise unconstrained CIT test suite considering the following parameters and their respective values:

$$\begin{aligned} OS &= \{macOS, Linux, Windows\}, \\ Protocol &= \{IPv4, IPv6\}, \\ DBMS &= \{MySQL, PostgreSQL, Oracle\}. \end{aligned}$$

We can formulate this problem as $\mathrm{MCA}(N, 2^1 3^2, 2)$ which is denoted as a model for the CIT problem. In other words, we have one parameter (*Protocol*) which can assume two values, two parameters (*OS*, *DBMS*) which can assume three values, and $t = 2$.

As we have mentioned in Section 1, CIT is an interesting solution for the test suite minimization problem. As a matter of perspective, let us consider that there are 10 parameters $(A, B, \cdots, J)$ and that each parameter has 5 values, i.e. $A = \{a_1, a_2, \cdots, a_5\}$, $B = \{b_1, b_2, \cdots, b_5\}$, ..., $J = \{j_1, j_2, \cdots, j_5\}$. If we performed an exhaustive combination, there would be $5^{10} = 9.765.625$ test cases generated where each test case is: $tc_i = \{a_k, b_k, \cdots j_k\}$. By using version 1.2 of TTR with $t = 2$, even in a unconstrained context, the test suite reduces to 45 test cases. This gives an idea of the strength of CIT for test suite minimization.

Note that the concepts and definitions we provided in this section are related to the context in which our work is inserted: unconstrained CIT. In case of constrained CIT, constraints must be considered and other definitions can be used (see e.g. (Yamada et al. 2016)).

3 TTR: a new algorithm for combinatorial interaction testing

In this section we detail versions 1.1 and 1.2 of our algorithm. The three versions (1.0 (Balera and Santiago Júnior 2015), 1.1, and 1.2) of TTR were implemented in Java.

3.1 TTR: Version 1.1

Version 1.0 of TTR (Balera and Santiago Júnior 2015) can be summarized as follows: (i) it generates all possible t-tuples that have not yet been covered. The *Constructor* procedure constructs the matrix Θ; (ii) it generates an initial solution, the matrix M; and (iii) it reallocates the t-tuples from Θ in order to achieve the best final solution (M) via the *Main* procedure. Then, the final set of test cases is updated in the matrix M. An important point here is that we order the parameters and values that are submitted to the algorithm. In other words, if we submit five parameters A, B, C, D, E with 10, 4, 3, 8, 5 values respectively, TTR orders these five parameters in ascending order: A, D, E, B, C. The goal is trying to be insensitive to the input order of parameters and values.

The same steps described above also exist in TTR 1.1. However, comparing with version 1.0 (Balera and Santiago Júnior 2015), in version 1.1 we do not order the parameters and values submitted to our algorithm. The result is that test suites of different sizes may be derived if we submit a different order of parameters and values. The motivation for such a change is because we realized that, in some cases, less test cases were created due to non-ordering of parameters and values.

Let us consider the running example in Fig. 1 with the strength, t, equals to 2. It is important to note that this is a unit testing level and hence each one of the parameters of *register* is an input parameter sumitted to TTR. Thus, there are 3 parameters: *bank*, *function* and *card*. We assume that there are two banks (*bankA*, *bankB*), two functions (*debit*, *credit*), and three types of cards (*cardA*, *cardB*, *cardC*) to deal with. Therefore, there are 2, 2, and 3 values of *bank*, *function* and *card*, respectively, as shown in Table 1.

A high-level view of version 1.1 of TTR is in Algorithm 1. The main reasoning of TTR 1.1 is to build an MCA M through the reallocation of t-tuples from a matrix Θ to this matrix M, and then each reallocated t-tuple should cover the greatest number of t-tuples

```
public void register(Bank bank, Function function, Card card);
```

Fig. 1 A running example: *register* method

not yet covered, considering a parameter called a *goal* (ζ). Also note that P is the submitted set of parameters, V is the set of values of the parameters, and t is the strength. As we have just pointed out, TTR 1.1 follows the same general 3 steps as we have in TTR 1.0.

Algorithm 1 High-level view: TTR 1.1

input: Set of parameters, P; Set of values, V; Strength, t

output: Matrix, M

1: $\Theta \leftarrow Constructor(P, V, t)$
2: **while** $\Theta \neq \emptyset$ **do**
3: $\quad M \leftarrow AddTestCase(\Theta, t)$
4: $\quad \zeta \leftarrow calculateZeta(M, t)$
5: $\quad [M, \Theta] \leftarrow Main(M, \Theta, \zeta, t)$
6: **end while**

Before going on with the descriptions of the procedures of our algorithm, we need to define the following operators applied to the structures (set, sequence, matrix) we handle. We also present some examples to better illustrate how such operators work.

Definition 1 *Let A be a sequence and B be a set. The addition sequence-set operator, $\odot$, is such that $A \odot B$ is a sequence where the elements of B are added after the last position of A. Thus, if $|A|$ is the length of sequence A and $|B|$ is the cardinality of set B, $|A \odot B| = |A| + |B|$.*

Example: Let us consider sequence $A = \{1, 2, 3\}$ and set $B = \{4, 5\}$. Then, $A \odot B = \{1, 2, 3, 4, 5\}$.

Definition 2 *Let A and B be two sequences with the same length, i.e. $|A| = |B|$. The addition sequence-sequence operator, $\oplus$, is such that $A \oplus B$ is a sequence where the element in position i of $A \oplus B$, ab_i, is a_i, the element of A in position i, or b_i, the element of B in position i. Also note the definition of an "empty" element, λ, within a sequence which is an element with no value. This operator then assumes that if $a_i \neq \lambda$ and $b_i \neq \lambda$ then $ab_i = a_i = b_i$. However, if $a_i = \lambda$ and $b_i \neq \lambda$ then $ab_i = b_i$. On the other hand, if $a_i \neq \lambda$ and $b_i = \lambda$ then $ab_i = a_i$. Note that $|A \oplus B| = |A| = |B|$.*

Example: Let us consider sequences $A = \{1, 2, \lambda\}$ and $B = \{\lambda, 2, 3\}$. Then, $A \oplus B = \{1, 2, 3\}$.

Table 1 Example of parameters and values: Fig. 1

Parameter	Values
Bank	*bankA*, *bankB*
Function	*debit*, *credit*
Card	*cardA*, *cardB*, *cardC*

Definition 3 *Let A and B be two sequences. The removal operator, $\ominus$, is such that $A \ominus B$ is a sequence obtained by "removing" each element of B, b_i, from A. This operator assumes that the original sequences A and B are known so that $A \ominus B = A$.*

Example: Let us consider that originally we have sequences $A = \{1, 2, \lambda\}$, $B = \{\lambda, 2, 3\}$, and $A \oplus B = \{1, 2, 3\}$. Then $A \ominus B = A = \{1, 2, \lambda\}$.

Definition 4 *Let A and B be two sets. The set difference operator, $\setminus$, is as defined in set theory.*

Example: Let us consider we have sets $A = \{1, 2, 3\}$ and $B = \{2, 3\}$. Then $A \setminus B = \{1\}$.

Definition 5 *Let A be a matrix and B be a sequence. The concatenation operator, $\bullet$, is such that $A \bullet B$ is a matrix where a new row (sequence) B is added after the last row of A.*

Example: Let us consider the matrix A below and sequence $B = \{10, 11, 12\}$. The matrix $A \bullet B$ is shown below.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$A \bullet B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Definition 6 *Let A be a matrix and B be a sequence. The removal from matrix operator, $\circ$, is such that $A \circ B$ is a matrix obtained by removing the entire row (sequence) B from the last row of matrix A. This operator assumes that the original matrix A and sequence B are known so that $A \circ B = A$*

Example: Let us consider we have matrix A and sequence B presented in the previous example. Then $A \circ B = A$ as shown below.

$$A \circ B = A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

3.1.1 The constructor procedure

According to the specified input (parameters and values), the *Constructor* procedure aims to generate all t-tuples that needs to be covered. Each t-tuple is in the matrix $\Theta_{|C| \times |P|}$ [3] where $|C|$ represents the number of t-tuples, t is the strength, and $|P|$ is the number of parameters.

Each row, θ_i, of Θ is a t-tuple that has not yet been covered and it has a variable, *flag*, associated with it whose purpose is to aid in the reallocation process of the t-tuple into the final solution. Note that since the order matters, each t-tuple θ_i is indeed a sequence and not a set. Moreover, *flag* does not belong to Θ. Table 2 shows the

Table 2 Matrix Θ for the example in Fig. 1

θ_i	Bank	Function	Card	Flag
1	*bankA*	*debit*	λ	*false*
2	*bankA*	*credit*	λ	*false*
3	*bankB*	*debit*	λ	*false*
4	*bankB*	*credit*	λ	*false*
5	*bankA*	λ	*cardA*	*false*
6	*bankA*	λ	*cardB*	*false*
7	*bankA*	λ	*cardC*	*false*
8	*bankB*	λ	*cardA*	*false*
9	*bankB*	λ	*cardB*	*false*
10	*bankB*	λ	*cardC*	*false*
11	λ	*debit*	*cardA*	*false*
12	λ	*debit*	*cardB*	*false*
13	λ	*debit*	*cardC*	*false*
14	λ	*credit*	*cardA*	*false*
15	λ	*credit*	*cardB*	*false*
16	λ	*credit*	*cardC*	*false*

matrix Θ for the example shown in Fig. 1 and $t = 2$. Note that interactions are made for the values of *bank**function*, *bank**card*, and *function**card*. Then, a t-tuple corresponding to the interaction of factors *bank**function* can be written in the form $\theta_i = \{bankA, debit, \lambda\}$. Initially, all values of *flag* are *false*. Algorithm 2 shows the *Constructor* procedure.

Algorithm 2 The *Constructor* procedure: TTR 1.1

input: $P = \{p_j \mid j = 1...|P|\}, V = \{v_k \mid k = 1...|V|\}, t$

output: $\Theta_{|C|\times|P|} = \{\theta_i \mid i = 1...m\}$

1: $E \leftarrow enumerator(|P|, t)$
2: **let** $I_l \subset E$
3: **let** $p_1 \subset I_l$
4: **let** $\theta_i \subset \Theta$
5: **for all** $\{v_k\} \in p_1$ **do**
6: $\quad \theta_i \leftarrow \theta_i \odot \{v_k\}$
7: $\quad \Theta \leftarrow \Theta \bullet \theta_i$
8: **end for**
9: **while** $E \neq \emptyset$ **do**
10: $\quad$ **let** $I_l \subset E$
11: $\quad$ **let** $p_j \subset I_l$
12: $\quad Aux \leftarrow \Theta$
13: $\quad$ **while** $p_j \subset I_l, j > 1$ **do**
14: $\quad\quad$ **for all** $\{v_k\} \in p_j$ **do**
15: $\quad\quad\quad$ **for all** $\theta_i \subset Aux$ **do**
16: $\quad\quad\quad\quad \Theta \leftarrow \Theta \bullet (\theta_i \odot \{v_k\})$
17: $\quad\quad\quad$ **end for**
18: $\quad\quad$ **end for**
19: $\quad$ **end while**
20: $\quad \Theta \leftarrow \Theta \setminus Aux$
21: $\quad E \leftarrow E \setminus I_l$
22: **end while**

Constructor operates as follows: based on the set of parameters (domain), P, and the strength (t), interactions between the parameters are generated through the enumeration procedure, and stored in a set named E (line 1). For example, we have 3 parameters (*bank*, *function* and *card*) and t = 2 thus we know that the enumerator will generate the interactions 2 per 2 ($t = 2$) between these 3 parameters. Thus $E = \{I_1, I_2, I_3\}$ where we have the sets $I_1 = \{bank, function, \lambda\}$, $I_2 = \{bank, \lambda, card\}$, and $I_3 = \{\lambda, function, card\}$. For better understanding, we denote the elements of I_l in this way: $bank \backslash function$, $bank \backslash card$ and $function \backslash card$. Then, the interactions (I_l) are selected one at a time (line 2), and during this selection, t-tuples are constructed based on each parameter of that interaction: in line 5, the first parameter of the first interaction, p_1, is selected. Note that each parameter, p_j, is indeed another set composed of values, v_k. Thus, $p_1 = bank = \{bankA, bankB\}$, $p_2 = function = \{debit, credit\}$, and $p_3 = card = \{cardA, cardB, cardC\}$. Therefore, each of the values (v_k) is added in t-tuples (θ_i) (line 6) and also in Θ (line 7). Recall that θ_i is indeed a sequence. From now on, subsequent parameters are selected one by one, and a new t-tuple is generated from the combination between each of the values (v_k) with each of the preexisting t-tuples (θ_i) in Θ (line 16). For example, the algorithm selects the first generated interaction, I_1, $bank \backslash function$ and construct all t-tuples between these two parameters. After processing each interaction, I_l, the *Constructor* procedure removes it from the set E (line 21).

Note that the main difference between TTR 1.0 and 1.1 is that TTR 1.0 performs the ordering of the domain, P, that is the parameters are ordered according to the amount of values they have: from the highest to the lowest quantity. For example, considering Fig. 1 and this input order: *bank*, *function*, and *card*. In version 1.0, parameters are stored in an ordered way: the first parameter becomes *card* (3 values), the second parameter is *bank* (2 values) and the last parameter is *function* (2 values). In version 1.1, there is no such ordering and this explains why *bank* and *function* generate the first rows (t-tuples) of Θ (see Table 2).

3.1.2 The initial solution and addition of test cases

The matrix $M_{N \times (|P|+1)}$ is the MCA we need to construct where there are N rows (i.e. test cases) and $|P|$ parameters. The $(|P|+1)$-th column is not used to represent any parameter but rather to mean the value of the *goal* (ζ) associated with that test case. There exists an initial solution for the matrix M that is obtained by selecting the parameters interaction I_l that has the largest amount of uncovered t-tuples (line 3 in Algorithm 1). Considering the input order *bank*, *function*, *card*, $I_2 = bank \backslash card$ that is chosen because it has 6 t-tuples and it appears first than $I_3 = function \backslash card$. All t-tuples derived via I_2 in the initial solution are combined with empty test cases, respecting the order of input of the parameters/values submitted to TTR 1.1 as shown in Table 3 (see t-tuples $\theta_5 = \{bankA, \lambda, cardA\}$, $\theta_6 = \{bankA, \lambda, cardB\}, \cdots$ from Θ (Table 2) in the initial M).

In the same way, to the extent that existing test cases are no longer sufficient to allocate the remaining t-tuples in the Θ matrix, the same procedure is used to include new test cases in matrix M. In other words, when reallocation of t-tuples becomes inefficient, it is necessary to include new test cases. Thus, as in the construction of the initial solution, the interaction of factors I_l that has the largest amount of uncovered t-tuples is selected, so that these will become new test cases. This strategy is performed on line 3 of Algorithm 1.

Table 3 Initial *M*: example of Fig. 1

Bank	Function	Card	ζ
bankA	λ	*cardA*	2
bankA	λ	*cardB*	2
bankA	λ	*cardC*	2
bankB	λ	*cardA*	2
bankB	λ	*cardB*	2
bankB	λ	*cardC*	2

3.1.3 Goals

In order to modify the current solution to obtain the final solution, the test suite M, we rely on the variable *goal* (ζ). For each row of M, i.e. for each test case, there is an associated *goal*.

As the objective is to address the largest number of uncovered t-tuples, the *goal* is calculated according to the maximum number of uncovered t-tuples which potentially may be covered when a t-tuple θ_i is moved from Θ to M. This results in a temporary test case τ_r. In order to find ζ, it is necessary to take into account: (i) the disjoint parameters, P_d, covered by the union between t-tuple θ_i and a test case from M; (ii) the number of parameter interactions, y, which τ_r has already covered; and (iii) the strength t. Therefore:

$$\zeta = \binom{P_d}{t} - y.$$

Let us consider again Fig. 1 and $t = 2$. According to Θ (see Table 2), the initial solution, M, is composed by the t-tuples due to parameters *bank**card*. This is because the $I_2 =$ *bank**card* has 6 tuples, $I_3 =$ *function**card* has 6 t-tuples, and $I_1 =$ *bank**function* has 4 t-tuples. As *bank**card* appears first than *function**card* and both have 6 tuples, so the algorithm selects it for reallocating into M.

The number of disjoint parameters, P_d, is equal to 3. As the interaction *bank**card* is already contemplated in matrix M, the next parameter interaction providing the largest number of non-addressed t-tuples is *function**card*. Then we have all 3 parameters with *bank**function* and *function**card* which explains $P_d = 3$. As $t = 2$, we have $\binom{3}{2} = 3$. However, one of the 3 parameter interactions has already been covered during the initial solution (*bank**card*), so we need to cover only 2 parameter interactions. Thus, for each t-tuple in the initial solution M, there remains to be covered:

$$\zeta = \binom{3}{2} - 1 = 2.$$

This explains the goal (ζ) in Table 3. It is very important that y is subtracted in order to find ζ. If this is not done, the final goal will never be matched, since there are no uncovered t-tuples that correspond to this interaction.

Even considering y, it is also important to note that not always the expected targets will be reached with the current configurations of the M and Θ matrices. In other words, in certain cases, there will be times when no existing t-tuple will allow the test cases of the M matrix to reach its goals. It is at this point that it becomes necessary to insert new test cases in M. This insertion is done in the same way as the initial solution for M is constructed, as described in the section above.

3.1.4 The Main Procedure

The *Main* procedure is presented in Algorithm 3. After the construction of the matrix Θ, the initial solution, and the calculation of the goals of all t-tuples, *Main* sort Θ so that the elements belonging to the parameter interaction with the greatest amount of t-tuples get ahead (line 1). However, these t-tuples will not be reallocated from Θ to M at once. This is done gradually, one by one, as goals are reached (line 7 to 11). Since the matrix M is being traversed in the loop (line 4), it will be updated every time a t-tuple is combined with some of its test cases (note $\oplus$ in line 5).

Algorithm 3 The *Main* procedure: TTR 1.1

input: $\Theta_{|C|\times|P|} = \{\theta_i \mid i = 1...m\}$, $M_{N\times(|P|+1)} = \{\tau_r \mid r = 1..N\}$
output: $M_{N\times(|P|+1)} = \{\tau_r \mid r = 1..N\}$

1: $\Theta \leftarrow \text{sort}(\Theta)$
2: **for all** $\theta_i \subset \Theta$ **do**
3: **while** $\theta_i \not\subset M$ **do**
4: **for all** $\tau_r \subset M$ **do**
5: $\tau_r \leftarrow \tau_r \oplus \theta_i$ (matrix M update)
6: **if** $goal(\tau_r)$ **then** (test case τ_r matches the goal)
7: $\Theta \leftarrow \Theta \circ \theta_i$
8: **else**
9: $\tau_r \leftarrow \tau_r \ominus \theta_i$ (matrix M update)
10: **end if**
11: **end for**
12: $mark(\theta_i)$ (*flag* is changed to *true*)
13: **end while**
14: **end for**

Let us consider Fig. 2. All matrices in this figure represent snapshots of M. The upper left matrix (a) is the initial solution. As long as there exists t-tuples (θ_i) in Θ, the *Main* procedure works. Thus, *Main* selects from Θ the largest amount of uncovered t-tuples. In Table 2, t-tuples were selected from the parameter interactions $I_3 = function\backslash card$. Every t-tuple of the $function\backslash card$ interaction is combined with each test case in M until the t-tuple matches some goal (line 7).

When an uncovered t-tuple fits into a row of M to complete a test case and this t-tuple is not removed on the line 9 in Algorithm 3, it means that the *goal* for that row of M is reached. Take the first row of the initial M (Table 3) which is a test case (τ_r) originated from $\theta_5 = \{bankA, \lambda, cardA\}$, and the first t-tuple of $function\backslash card$ interaction not yet covered in Θ, $\theta_{11} = \{\lambda, debit, cardA\}$. The addition of $\theta_{11} = \{\lambda, debit, cardA\}$ in M is accepted because $\zeta = 2$ is reached. Note that the initial M, with test cases τ_r, is also an input parameter of this procedure. Hence, in line 5, M is updated due to the *addition sequence-sequence* operator ($\oplus$). In addition, note that τ_r is also a sequence as θ_i. In other words, by inserting $\theta_{11} = \{\lambda, debit, cardA\}$, we have a complete test case $\tau_r = \{bankA, debit, cardA\}$. In this way, the other two interactions $bank\backslash function$ ($\theta_1 = \{bankA, \lambda, debit\}$) and $function\backslash card$ ($\theta_{11} = \{\lambda, debit, cardA\}$) are covered, and the *goal* is achieved. The upper right matrix (b) in Fig. 2 shows the result of this first addition.

a

bank	function	card	ζ
$bankA$		$cardA$	2
$bankA$		$cardB$	2
$bankA$		$cardC$	2
$bankB$		$cardA$	2
$bankB$		$cardB$	2
$bankB$		$cardC$	2

b

bank	function	card	ζ
$bankA$	debit	$cardA$	0
$bankA$	credit	$cardB$	0
$bankA$		$cardC$	2
$bankB$	credit	$cardA$	0
$bankB$	debit	$cardB$	0
$bankB$		$cardC$	2

c

bank	function	card	ζ
$bankA$	debit	$cardA$	0
$bankA$	credit	$cardB$	0
$bankA$		$cardC$	1
$bankB$	credit	$cardA$	0
$bankB$	debit	$cardB$	0
$bankB$		$cardC$	1

d

bank	function	card	ζ
$bankA$	debit	$cardA$	0
$bankA$	credit	$cardB$	0
$bankA$	debit	$cardC$	0
$bankB$	credit	$cardA$	0
$bankB$	debit	$cardB$	0
$bankB$	credit	$cardC$	0

Fig. 2 Snapshots of *M*: **a** initial solution; **b** and **c** intermediate matrices; **d** final test suite

After all combinations between t-tuples and test cases are made, that is, when procedure ends, the new ζ is calculated. The bottom left matrix (c) shows the new values of ζ (see rows 3 and 6). Thus the steps described above are repeated with the insertion/reallocation of t-tuples into the matrix *M*. Once an uncovered t-tuple of Θ is included in *M* and meets the goal, that t-tuple is excluded from Θ (line 7). Note that if t-tuple does not allow the test to which it was combined to reach the goal, it is "unbound" (line 9) from this test case so that it can be combined with the next test case. The final test suite is the matrix *M* shown at the bottom right (d).

It is possible that a certain uncovered t-tuple does not fit into *M*. Consequently, the *flag* variable associated with this t-tuple in Θ is signed as *true* so that the *Main* procedure knows that such a t-tuple can no longer be compared with rows of *M*. *Main* continues as long as there are uncovered t-tuples. Table 4 shows part of Θ after the first iteration. Note that t-tuples $\theta_{13} = \{debit, cardC\}$ and $\theta_{16} = \{credit, cardC\}$ of the *function**card* interaction are not inserted into *M* (see the values *true*).

This exception is ilustred in Table 4, with $\theta_{13} = \{\lambda, debit, cardC\}$ and $\theta_{16} = \{\lambda, credit, cardC\}$ happens because the tests generated by these t-tuples and the available rows of the matrix *M* address t-tuples already covered in Θ. Assuming that the test consists of the combination of a t-tuple and row 3 of *M*, only one t-tuple is covered since there is no more t-tuples to be covered in *bank**card* and *bank**function*, as illustrated in Table 4. However, $\zeta = 2$ is not satisfied and these t-tuples can not be removed from Θ. Then it is necessary to recalculate the *goals* according to the parameter interactions that have been already addressed.

3.2 TTR: version 1.2

The high-level view of the new version of TTR, 1.2, is in Algorithm 4. This new version no longer uses the *Constructor* procedure since t-tuples are generated one at a time as

Table 4 Part of Θ: unfitness

θ_i	Bank	Function	Card	Flag
13	λ	*debit*	*cardC*	*true*
16	λ	*credit*	*cardC*	*true*

```
public void update(Status status, Education education, Regime regime, Working_hours working_hours);
```

Fig. 3 A second running examples: *update* method

they are reallocated. In other words, there is no more Θ, a matrix of t-tuples. What we have now is only φ which is a matrix of parameter interactions. TTR 1.2 works as follow: (i) generates only the parameter interactions (it does not generate the t-tuples yet); (ii) generates an initial solution, the matrix M; and (iii) the t-tuples are generated from φ in order to get the final solution (M) via the *Main* procedure.

Algorithm 4 High-level view: TTR 1.2

input: Set of parameters, P; Set of values, V; Strength, t

output: Matrix, M

1: **while** $\exists\ \tau_r \mid \tau_r \in M \wedge\ \zeta > 0$ **do**
2: $M \leftarrow AddTestCase(P, V, t)$
3: $[M, \varphi] \leftarrow Main(M, \varphi, \zeta, t)$
4: $\zeta \leftarrow calculateZeta(M, t)$
5: **end while**

Let us consider the code in Fig. 3 where parameters and values are given in Table 5 and $t = 3$. It is a method to update information into a database of a company. TTR 1.2 constructs only parameter interactions according to the strength and stores the **number** of corresponding t-tuples (Φ) in a matrix φ. These parameter interactions are $I_1 = \{status, education, regime, \lambda, 8\}$, $I_2 = \{status, education, \lambda, working\_hours, 8\}$, $I_3 = \{status, \lambda, regime, working\_hours, 8\}$, and $I_4 = \{\lambda, education, regime, working\_hours, 8\}$, where the last element of I_l is the number of t-tuples Φ (in all these case $I_l = 8$). Here, each interaction I_l is indeed a sequence because the algorithm needs to know the exact number of t-tuples and hence position matters. Note that λ is the empty element. No t-tuple corresponding to any parameters/values interactions is constructed as shown in Table 6. The calculation of Φ is simply done by multiplying the number of values of each parameter in the corresponding interaction.

3.2.1 Initial solution

In this case, the initial solution is no more than the construction of the t-tuples due to the parameters interactions with greater Φ, and their transformation into test cases. In Table 7, the t-tuples of the parameters interaction $I_1 = \{status, education, regime, 8\}$ were all transformed into test cases and therefore, for this parameters interaction, Φ becomes 0 and it is no longer considered in the *goal* (ζ) calculation (Table 8). In fact, we have 4 parameters and $t = 3$, thus 4 interactions of possible parameters are generated: one

Table 5 Example of parameters and values: Fig. 3

Parameter	Values
status	*active, retired*
education	*undergraduate, graduate*
regime	*partial, full*
working_hours	*afternoon, morning*

Table 6 Matrix φ for the example of Fig. 3

Interation of parameters	Status	Education	Regime	Working_hours	Φ
I_1	x	x	x	λ	8
I_2	x	x	λ	x	8
I_3	x	λ	x	x	8
I_4	λ	x	x	x	8

is already covered remaining 3 parameter interactions (I_2, I_3, I_4) to be addressed. This justifies $\zeta = 3$ (Table 7).

3.2.2 The main procedure

The new *Main* procedure is presented in Algorithm 5. After calculating the parameters interactions, Φ, the initial solution, and the *goals* of all test cases of M, *Main* selects the parameter interaction that has the highest amount of uncovered t-tuples (line 2) and constructs t-tuples so that they can be reallocated. However, they will be reallocated gradually, one by one, as goals are reached (line 4 to 13). The procedure combines the t-tuples with the test cases of M in order to match them.

Algorithm 5 The *Main* procedure: TTR 1.2

input: $\varphi_{|I|\times(|P|+1)} = \{I_l \mid l = 1...m\}, M_{N\times(|P|+1)} = \{\tau_r \mid r = 1..N\}$
output: $M_{|n|\times(|P|+1)} = \{\tau_r \mid r = 1..n\}$

1: **while** $\varphi \neq \emptyset$ **do**
2: **let** $I_l \in \varphi$ (Select the parameter interaction that has the largest amount of uncovered t-tuples)
3: $S \leftarrow buildTuples(I_l)$
4: **let** $\theta_i \subset S$
5: **while** $\Phi_{I_l} \neq \emptyset$ **do**
6: **for all** $\tau_r \subset M$ **do**
7: $\tau_r \leftarrow \tau_r \oplus \theta_i$ (matrix M update)
8: **if** $goal(\tau_r)$ **then** (Verify the test case τ_r which has ζ uncovered t-tuples)
9: $S \leftarrow S \circ \theta_i$
10: **for all** $I_l \subset \varphi \wedge I_l \subset \tau_r$ **do**
11: **let** $\Phi \in I_{l,\Phi}$
12: $\Phi \leftarrow \Phi$ - 1
13: **end for**
14: **else**
15: $\tau_r \leftarrow \tau_r \ominus \theta_i$ (matrix M update)
16: **end if**
17: **end for**
18: $mark(\theta_i)$ (*flag* is changed to *true*)
19: **end while**
20: $\varphi \leftarrow \varphi \circ I_l$
21: **end while**

Let us take the second running example (Fig. 3). The parameters interaction with the highest amount of non-addressed t-tuples is $I_2 = \{status, education, \lambda, working\_hours, 8\}$ (Φ = 8; Table 8 after the initial solution): all t-tuples of this interaction are generated and

Table 7 Initial *M* for the example of Fig. 3

Status	Education	Regime	Working_hours	ζ
active	*undergraduate*	*partial*	λ	3
active	*graduate*	*partial*	λ	3
retired	*undergraduate*	*partial*	λ	3
retired	*graduate*	*partial*	λ	3
active	*undergraduate*	*full*	λ	3
active	*graduate*	*full*	λ	3
retired	*undergraduate*	*full*	λ	3
retired	*graduate*	*full*	λ	3

stored in a sequence S (line 3). The first t-tuple, $\theta_1 = \{active, undergraduate, \lambda, afternoon\}$, is combined with each test case, τ_r in M (line 7). The t-tuple in question fits test case 1, τ_1. At that moment, it is verified whether the t-tuple θ_i makes the τ_r test reach its goal. This control is done through the *goal*() function that receives the τ_r test case and then is broken in t-tuples (line 8) according to the parameters interactions that have Φ other than 0. For example, the test case $\tau_1 = \{active, undergraduate, partial, afternoon\}$ is broken in t-tuples: $\{\{active, undergraduate, partial, \lambda\}, \{active, undergraduate, \lambda, afternoon\}, \{active, \lambda, partial, afternoon\}, \{\lambda, undergraduate, partial, afternoon\}\}$. It is then verified how many of these t-tuples do not exist in M and, if this amount equals the respective ζ, θ_i is permanently stored in M and a unit is taken from the value of Φ of each of the factor interactions that have t-tuples covered by this test case (line 12) because this keeps if the control of the quantity of t-tuples that still have to be covered. Since the matrix M is being traversed in the loop (line 6), it will be updated every time a t-tuple is combined with some of its test cases (line 7).

This step is repeated for all t-tuples. Each time a t-tuple is reallocated from S into M, the *goals* are recalculated. For example, when the matrix M permanently receives the 4th t-tuple, the test cases that become complete (with a value for each parameter) have $\zeta = 0$ while the others still have $\zeta = 3$ (Table 9).

All I_2 t-tuples are reallocated from S in order to achieve the *goal* of all M test cases resulting the final test suite presented in Table 10. In fact, the *Main* procedure does not construct new t-tuples from another parameters interaction if the current one is not zero: if the parameters interaction I_2 (selected due to the greatest Φ) still has t-tuples, *Main* will not select another parameters interaction. To do this, the *goal* of the test cases will be decreased by one, until all t-tuples of the interaction of parameters I_2 make the test cases to match ζ.

4 Controlled experiment 1: TTR 1.1 × TTR 1.2

This section presents a controlled experiment where we compare versions 1.1 and 1.2 of TTR in order to realize whether there is significant difference between both versions of

Table 8 Matrix φ for the example of Fig. 3: after the initial solution

Interation of parameters	Status	Education	Regime	Working_hours	Φ
I_1	x	x	x	λ	0
I_2	x	x	λ	x	8
I_3	x	λ	x	x	8
I_4	λ	x	x	x	8

Table 9 Intermediate matrix *M* for the example of Fig. 3

Status	Education	Regime	Working_hours	ζ
active	*undergraduate*	*partial*	*afternoon*	0
active	*graduate*	*partial*	λ	3
retired	*undergraduate*	*partial*	λ	3
retired	*graduate*	*partial*	*afternoon*	0
active	*undergraduate*	*full*	λ	3
active	*graduate*	*full*	*afternoon*	0
retired	*undergraduate*	*full*	*afternoon*	0
retired	*graduate*	*full*	λ	3

our algorithm. We accomplished such an experiment where we jointly considered cost and efficiency in a multi-objective perspective.

4.1 Definition and context

The primary aim of this study is to evaluate cost and efficiency related to CIT test case generation via versions 1.1 and 1.2 of the TTR algorithm (both implemented in Java). The rationale is to perceive whether we have significant differences between the two versions of our algorithm.

Regarding the metrics, cost refers to the size of the test suites while efficiency refers to the time to generate the test suites. Although the size of the test suite is used as an indicator of cost, it does not necessarily mean that test execution cost is always less for smaller test suites. However, we assume that this relationship (higher size of test suite means higher execution cost) is generally valid. We should also emphasize that the time we addressed is not the time to run the test suites derived from each algorithm but rather the time to generate them. We jointly analyzed cost and efficiency in a multi-objective way.

The set of samples, i.e. the subjects, are formed by instances that were submitted to both versions of TTR to generate the test suites. We randomly chose 80 test instances/samples (composed of parameters and values) with the strength, t, ranging from 2 to 6. Table 11 shows part of the 80 instances/samples used in this study. Full data obtained in this experiment are presented in (Balera and Santiago Júnior 2017).

It is important to mention how each instance/sample can be interpreted. Let us consider instance $i = 1$ in Table 11:

$$2^1 4^1 5^1 3^1 6^1, \quad t = 2.$$

Table 10 Final matrix *M* for the example of Fig. 3

Status	Education	Regime	Working_hours	ζ
active	*undergraduate*	*partial*	*afternoon*	0
active	*graduate*	*partial*	*morning*	0
retired	*undergraduate*	*partial*	*morning*	0
retired	*graduate*	*partial*	*afternoon*	0
active	*undergraduate*	*full*	*morning*	0
active	*graduate*	*full*	*afternoon*	0
retired	*undergraduate*	*full*	*afternoon*	0
retired	*graduate*	*full*	*morning*	0

Table 11 Samples for the controlled experiment: Instances. Caption: *val* = value; *par* = parameter

i	Strength	val^{par}
1	2	$2^1 4^1 5^1 3^1 6^1$
2	2	$3^1 4^1 2^1 6^1 2^1$
3	2	$5^1 4^1 4^1 3^1 2^1$
4	2	$2^1 5^1 5^1 6^1 3^1$
5	2	$6^1 6^1 5^1 4^1 2^1$
6	2	$7^1 8^1 6^1 5^1 2^1$
7	3	$3^1 2^1 2^1 4^1 5^1$
8	3	$4^1 3^1 6^1 2^1 2^1 2^1$
9	3	$7^1 5^1 5^1 6^1 2^1 2^1$
10	3	$8^1 3^1 2^1 5^1 4^1$
...	...	...
21	5	$2^1 2^1 3^1 5^1 4^1 6^1$
22	5	$3^1 2^1 2^1 6^1 5^1 5^1$
23	5	$6^1 3^1 4^1 4^1 2^1 2^1$
24	5	$3^1 3^1 3^1 4^1 5^1 6^1$
25	6	$5^1 5^1 3^1 3^1 2^1 2^1 7^1$
26	6	$3^1 7^1 5^1 2^1 4^1 4^1 2^1$
27	6	$6^1 2^1 2^1 4^1 3^1 4^1 2^1$
28	6	$3^1 4^1 3^1 3^1 2^1 2^1 4^1$
29	6	$6^1 5^1 3^1 2^1 2^1 4^1 3^1$
30	6	$2^1 2^1 3^1 4^1 3^1 3^1 2^1$
...	...	...
41	4	$3^1 4^1 5^1 6^1 6^1 7^1$
42	4	$2^1 2^1 3^1 4^1 5^1 3^1$
43	5	$2^1 4^1 9^1 9^1 3^1 4^1$
44	5	$3^1 3^1 2^1 4^1 5^1 6^1$
45	5	$3^1 4^1 2^1 2^1 3^1 3^1$
46	5	$4^1 3^1 2^1 2^1 4^1 5^1$
47	5	$7^1 5^1 3^1 3^1 2^1 4^1$
48	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$
49	2	$2^1 2^1 2^1 3^1 4^1 2^1 2^1 3^1$
50	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$
...	...	...
61	4	$2^1 2^1 2^1 3^1 4^1 2^1 2^1 3^1$
62	4	$2^1 2^1 3^1 3^1 4^1 4^1$
63	5	$2^1 2^1 2^1 3^1 4^1 2^1 2^1 3^1$
64	5	$6^1 2^1 6^1 4^1 5^1$
65	6	$2^1 2^1 3^1 3^1 4^1 4^1$
66	5	$2^1 3^1 5^1 6^1 7^1 8^1$
67	5	$7^1 5^1 4^1 3^1 3^1 9^1$
68	5	$2^1 2^1 4^1 3^1 3^1 2^1$
69	5	$4^1 5^1 6^1 7^1 8^1 9^1$
70	5	$2^1 5^1 7^1 3^1 3^1 4^1 3^1$
...	...	...
80	3	$7^1 3^1 2^1 2^1$

In the context of unit test case generation for programs developed according to the Object-Oriented Programming (OOP) paradigm, this instance can be used to generate test cases for a class that has one attribute (parameter) which can take 2 values (2^1), 1 attribute that can take 4 values (4^1), another attribute that can take 5 values (5^1), $\cdots$, 1

attribute that can take 6 values (6^1). In the system and acceptance testing context, this same sample can be used to identify test scenarios (test objectives) in a model-based test case generation approach (Santiago Júnior 2011; Santiago Júnior and Vijaykumar 2012). In both cases, the test suites must meet the criteria of pairwise testing ($t = 2$) where each combination of 2 values of all parameters must be covered. Note that these samples were randomly selected and they cover a wide range of combinations of parameters, values, and strengths to be selected for very simple but also more complex case studies with different testing levels (unit, system, acceptance, etc.).

4.2 Hypotheses and variables

We defined two hypotheses as shown below:

- **Null Hypothesis,** $H_{0.1}$ - There is no difference regarding cost-efficiency between TTR 1.1 and TTR 1.2;
- **Alternative Hypothesis,** $H_{1.1}$ - There is difference regarding cost-efficiency between TTR 1.1 and TTR 1.2.

Regarding the variables involved in this experiment, we can highlight the *independent* and *dependent* variables (Wohlin et al. 2012). The first type are those that can be manipulated or controlled during the process of trial and define the causes of the hypotheses. For this experiment, we identified the algorithm/tool for CIT test case generation. The *dependent* variables allow us to observe the result of manipulation of the *independent* ones. For this study, we identified the number of generated test cases and the time to generate each set of test cases and we jointly considered them.

4.3 Description of the experiment

The experiment was conducted by the researchers who defined it. We relied on the experimentation process proposed in (Wohlin et al. 2012), using the R programming language version 3.2.2 (Kohl 2015). Both algorithms/tools (TTR 1.1, TTR 1.2) were subjected to each one of the 80 test instances (see Table 11), one at a time. The output of each algorithm/tool, with the number of test cases and the time to generate them, was recorded.

To measure cost, we simply verified the number of generated test cases, i.e. the number of rows of the final matrix M, for each instance/sample. The efficiency measurement required us to instrument each one of the implemented versions of TTR and measure the computer current time before and after the execution of each algorithm. In all cases, we used a computer with an Intel Core(TM) i7-4790 CPU @ 3.60 GHz processor, 8 GB of RAM, running Ubuntu 14.04 LTS (Trusty Tahr) 64-bit operating system. The goal of this second analysis is to provide an empirical evaluation of the time performance of the algorithms.

To perform the multi-objective cost-efficiency evaluation, we followed two steps. First, we transformed the cost-efficiency (two-dimensional) representation into a one-dimensional one. Thus, in a second step, we used statistical tests, such as the t-test or the nonparametric Wilcoxon test (Signed Rank) (Kohl 2015), to compare the two test suites (TTR 1.1 and TTR 1.2). To address the nondeterminism of the algorithms/tools, related to the the ordering input of parameters and values, we generated test cases with 5 variations in the order of parameters and values, and took into account the average of these 5

assessments for the statistical tests. We then got points (cA_i, tA_i) that represent the average cost (cA_i) and average time (tA_i) of the algorithms A (TTR 1.1, TTR 1.2) for each instance i $(1 \leq i \leq 80)$.

We then determined an optimal point in a two-dimensional space, the point (0,0). This point implies a cost **closer to 0** and requires a time **closer to 0**. The closest condition is because an algorithm is not expected to generate a test suite with, exactly, 0 test case or it does require 0 unit of time to generate the set of test cases. We then used a measure of distance, such as the Euclidean one, to measure the distance from the optimal point (0,0) to (cA_i, tA_i). Thus, each algorithm is then represented by a one-dimensional set, D, where each $d_i \in D$ is the Euclidean distance between (0,0) and (cA_i, tA_i) for every instance i. We selected the Euclidean distance because it is one of the most used similarity distance measure. In software testing, Euclidean distance has been used as a quality indicator in multi-objective test case/data generation (Filho and Vergilio 2015; Santiago Júnior and Silva 2017), to support the automation of test oracles for complex output domains (web applications (Delamaro et al. 2013), text-to-speech systems (Oliveira 2017)), and many others.

Based on this cost-efficiency one-dimensional representation, we relied on appropriate statistical evaluation to check data normality. Verification of normality was done in three steps: (i) by using the Shapiro-Wilk test (Shapiro and Wilk 1965) with a significance level $\alpha = 0.05$; (ii) by checking the *skewness* of the frequency distribution (in this case, $-0.1 \leq skewness \leq 0.1$ so that the data can be considered as normally distributed); and (iii) by using a graphical verification by means of Q-Q plot (Kohl 2015) and histogram. Thus, we believe we have greater confidence in this conclusion on data normality compared to an approach that is based only on the Shapiro-Wilk test considering the effects of polarization due to the length of the samples.

If we concluded that data came from a normally distributed population, then the paired, two-sided t-test was applied with $\alpha = 0.05$. Otherwise, we applied the nonparametric paired, two-sided Wilcoxon test (Signed Rank) (Kohl 2015) with $\alpha = 0.05$, too. However, if the samples presented ties, we applied a variation of the Wilcoxon test, the Asymptotic paired, two-sided Wilcoxon (Signed Rank) (Kohl 2015), suitable to treat ties, with significance level $\alpha = 0.05$.

In order to reject the Null Hypothesis, $H_{0.1}$, we checked whether $p - value < 0.05$ (t-test) or whether both $p - value < 0.05$ and $|z| > 1.96$ (Wilcoxon) where z is the z-score. If $H_{0.1}$ was rejected, we observed the average of all Euclidean distances (80) due to each algorithm. The algorithm that presented the lowest average of Euclidean distances was the one chosen as the most adequate. If $H_{0.1}$ could not be rejected, then the conclusion was that no statistical difference existed between both algorithms.

5 Results and discussion

In this section, we present the results of this first controlled experiment. Based on the cost-efficiency one-dimensional representation (Section 4.3), we considered four evaluation classes as follows:

- All strenghts. In this case, all 80 instances/samples (Table 11) with all strengths (2, 3, 4, 5, and 6) were taken into account. Our idea here is trying to perceive the

cost-efficiency performance of both algorithms in a context where several different strengths are selected to generate a test suite;

- Low strengths. In this case, we selected only the samples with strength equals to 2. Our aim is to note how the algorithms perform for low strengths;
- Medium strengths. By selecting samples with strength equals to 3 or 4, we want to evaluate an intermediate strength context;
- High strengths. We aim to assess the performance for higher strengths, i.e. $t = 5$ or 6.

Table 12 presents the Euclidean distances of part of the 80 samples (all strenghts class only; complete data are in (Balera and Santiago Júnior 2017)) as well as the average values, $\overline{x}$, of such distances. We checked data normality where Table 13 presents the $p - value$, p, due to the Shapiro-Wilk test and the *skewness*. Note that this table shows p and *skewness* of all four classes above (all, low, medium, and high strenghts). Moreover **Sol 1** is TTR 1.1 and **Sol 2** is TTR 1.2. Figures 4 and 5 present the Q-Q plots and histograms for all strengths, Figs. 6 and 7 present the Q-Q plots and histograms for lower strengths, Figs. 8 and 9 present the Q-Q plots and histograms for medium strengths, and Figs. 10 and 11 present the Q-Q plots and histograms for higher strengths, respectively.

We can clearly see that all these data did not come from a normally distribution population because $p < 0.05$ and the *skewness* is far from 0. Moreover, Q-Q plots and histograms reassure this conclusion. Hence, we used the nonparametric paired, two-sided Wilcoxon test (Signed Rank) or its variation (Asymptotic) when ties were detected. Table 14 presents the $p - value$, p, $|z|$, and additional information for classes all and low strengths while Table 15 shows the results for medium and high strengths.

Based on Tables 14 and 15, we could not reject $H_{0.1}$ (no difference) for all strengths, but we could do it for the other evaluation classes and hence accept the Alternative Hypothesis, $H_{1.1}$. As we have previously pointed out, when there is difference regarding cost-efficiency, we examine the average values of the Euclidean distances: the smaller the better. TTR 1.1 is better, in terms of cost-efficiency, than TTR 1.2 for lower strengths ($t = 2$). However, for medium ($t = 3, 4$) and higher strenghts ($t = 5, 6$), TTR 1.2 surpassed TTR 1.1. This makes sense because in TTR 1.2 we do not generate, at the beginning, the matrix of t-tuples and hence we expect that the last version of our algorithm can handle properly higher strengths.

Therefore, even if we did not find statistical difference with all the strengths and TTR 1.1 was the best for lower strenghts, we decided to select TTR 1.2, to compare with the other solutions for unconstrained CIT test case generation, because TTR 1.2 performed better than TTR 1.1 for medium and higher strengths.

5.1 Validity

The conclusion validity has to do with how sure we are that the treatment we used in an experiment is really related to the actual observed outcome (Wohlin et al. 2012). One of the threats to the conclusion validity is the reliability of the measures (Campanha et al. 2010). We automatically obtained the measures via the implementations of the algorithms and hence we believe that replication of this study by other researchers will produce similar results. Even if other researchers may get different absolute results, especially related to the time to generate the test suites simply because such results depend on the computer configuration (processor, memory, operating system), we dot not expect a different

Table 12 Experiment 1 - Results of the analysis of Euclidean Distance (all strengths)

i	Euclidean Distance – TTR 1.1	Euclidean Distance – TTR 1.2
1	3.488323e+01	1.019155e+02
2	2.615645e+01	3.216458e+01
3	2.044798e+01	3.473557e+01
4	3.063201e+01	4.143670e+01
5	3.722633e+01	1.218346e+02
6	5.678345e+01	1.264736e+02
7	6.604362e+01	1.031089e+02
8	1.129060e+07	7.720515e+02
9	1.299138e+03	7.355677e+03
10	1.799441e+02	4.758598e+02
…	…	…
20	1.882607e+05	2.061971e+04
21	6.017033e+03	4.194014e+03
22	1.347988e+04	7.574538e+03
23	2.958807e+03	1.008311e+03
24	1.623848e+04	3.928049e+03
25	9.726865e+05	1.317154e+05
26	6.118455e+05	1.196950e+05
27	4.312599e+04	2.337230e+04
28	9.039640e+03	3.555690e+03
29	2.570393e+05	4.135795e+04
30	1.970342e+03	1.056320e+03
…	…	…
40	2.473112e+03	9.169621e+03
41	1.286935e+05	3.101953e+05
42	1.097300e+03	3.468359e+03
43	7.759235e+05	7.904737e+04
44	2.509135e+04	4.500654e+03
45	2.997279e+02	2.459155e+02
46	1.487948e+03	7.328219e+02
47	1.984648e+04	3.579825e+03
48	8.455744e+01	1.248893e+03
49	1.644384e+01	2.957483e+02
50	1.428690e+02	1.321925e+03
…	…	…
60	1.215468e+05	5.199665e+05
61	2.720418e+04	8.558826e+04
62	7.826624e+02	2.312496e+03
63	9.568609e+04	2.675675e+05
64	1.440006e+03	1.440000e+03
65	5.760009e+02	5.760009e+02
66	1.688547e+06	1.320639e+05
67	1.147371e+06	1.475662e+05
68	1.683444e+02	1.554663e+02
69	2.271243e+02	4.574975e+06
70	1.087280e+04	4.101938e+05
…	…	…
80	4.202333e+01	4.402227e+01
$\bar{x}$	323991	111732.4

Table 13 Experiment 1 - Results of the analysis of data normality

Comparison	Strength	*p* - Sol 1	skewness - Sol 1	*p* - Sol 2	skewness - Sol 2
TTR 1.1 x TTR 1.2	All	2.2e-16	6.523523	2.2e-16	8.186584
TTR 1.1 x TTR 1.2	Low (2)	5.65E-06	3.952605	5.03E-06	4.000011
TTR 1.1 x TTR 1.2	Medium (3, 4)	8.47E-08	4.050349	1.03E-04	2.247599
TTR 1.1 x TTR 1.2	High (5,6)	1.50E-04	1.920776	2.99E-08	5.084482

conclusion validity. Moreover, we relied on adequate statistical methods in order to reason about data normality and whether we did really find statistical difference between TTR 1.1 and TTR 1.2. Hence, our study has a high conclusion validity.

The internal validity aims to analyze whether the treatment actually caused the outcome (result). Hence, we need to be sure whether other parameters have not caused the outcome, parameters that have not been controlled or measured. There are many threats to internal validity such as testing effects (measuring the participants repeatedly), history (experiment external events or between repeated measures of the dependent variable may influence the responses of the subjects, e.g. interruption of the treatment), instrument change, maturation (participants might mature during the study or between measurements), selection bias (differences between groups), etc. Note that the participants of our experiment are randomly samples composed of parameters, values, and strengths. Hence, we neither had any human/nature/social parameter nor unanticipated events to interruption the collection of the measures once started to pose an internal validity. Hence, we claim that our experiment has a high internal validity.

In the construct validity, the goal is to ensure that the treatment reflects the construction of the cause, and the result the construction of the effect. This is also high because we used the implementations of TTR 1.1 and TTR 1.2 to assess the cause, and the results, supported by the decision-making procedure via statistical tests, clearly provided the basis for the decision to be made between both algorithms.

Threats to external validity compromise the confidence in asserting that the results of the study can be generalized to and between individuals, settings, and under the temporal perspective. Basically, we can divide threats to external validity in two categories: threats to population and ecological threats.

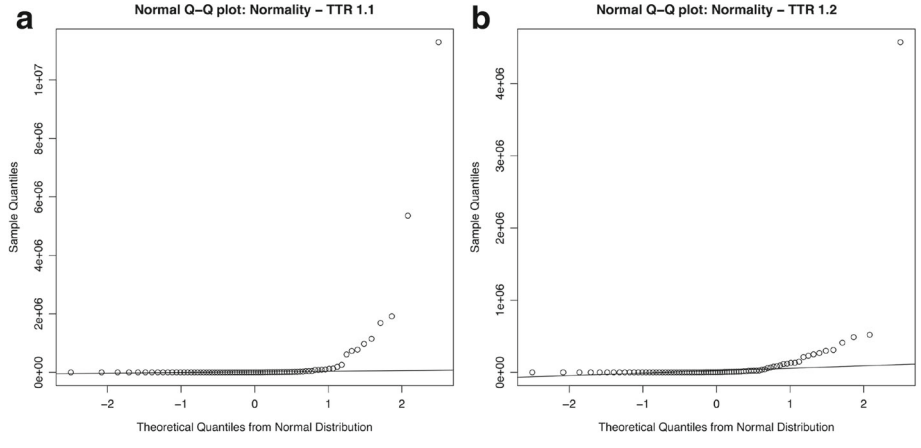

Fig. 4 Experiment 1: Q-Q plots. **a** TTR1.1; **b** TTR 1.2 - All Strengths

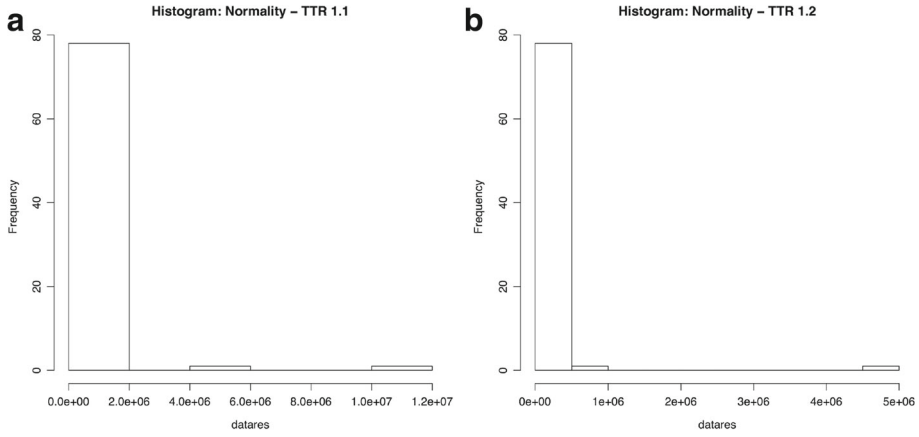

Fig. 5 Experiment 1: Histograms. **a** TTR1.1; **b** TTR 1.2 - All Strengths

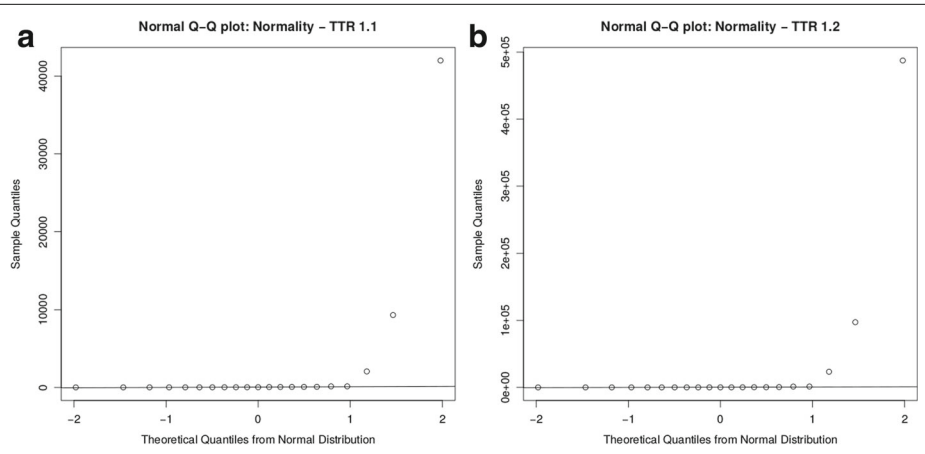

Fig. 6 Experiment 1: Q-Q plots. **a** TTR1.1; **b** TTR 1.2 - 2 Strength

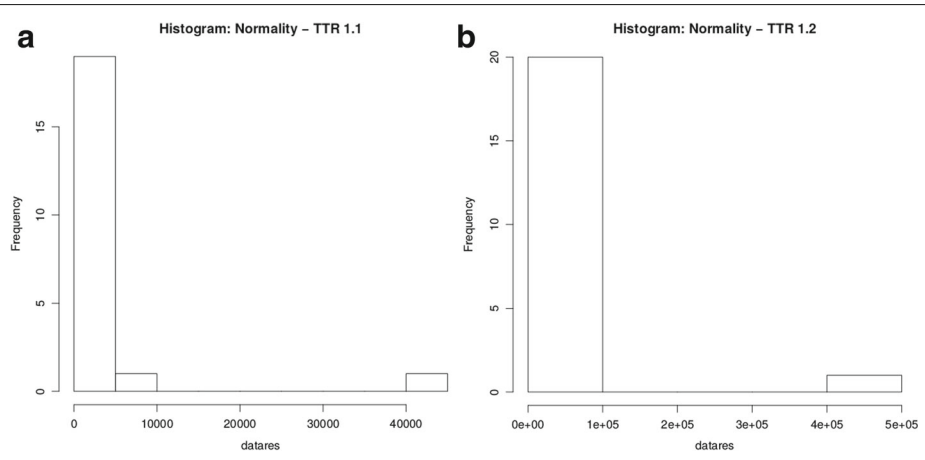

Fig. 7 Experiment 1: Histograms. **a** TTR1.1; **b** TTR 1.2 - 2 Strength

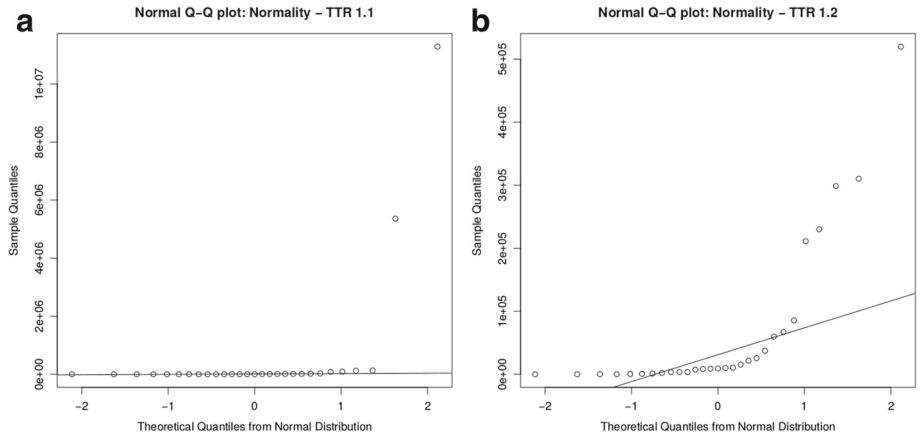

Fig. 8 Experiment 1: Q-Q plots. **a** TTR1.1; **b** TTR 1.2 - 3 and 4 Strengths

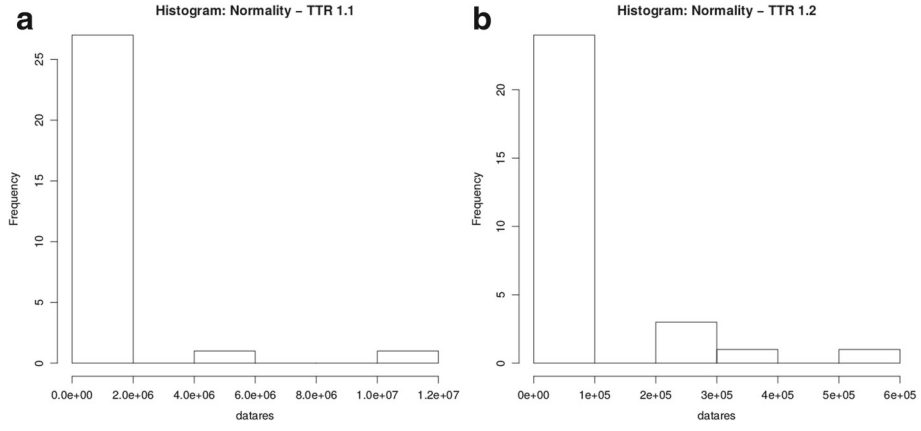

Fig. 9 Experiment 1: Histograms. **a** TTR1.1; **b** TTR 1.2 - 3 and 4 Strengths

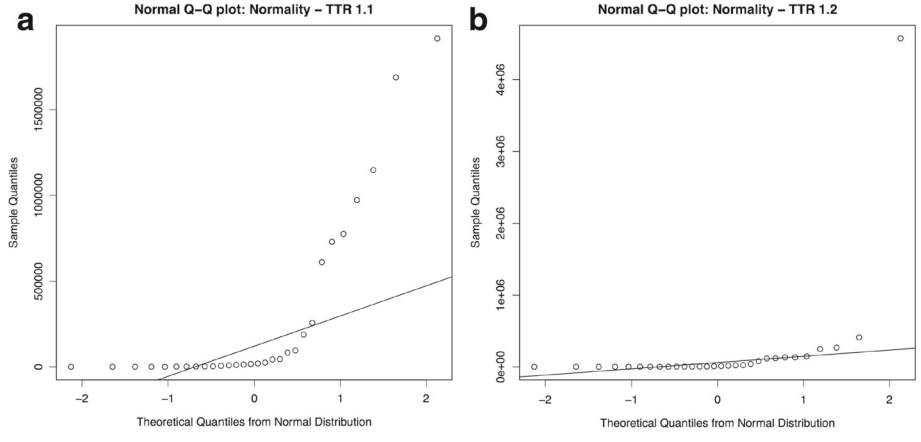

Fig. 10 Experiment 1: Q-Q plots. **a** TTR1.1; **b** TTR 1.2 - 5 and 6 Strengths

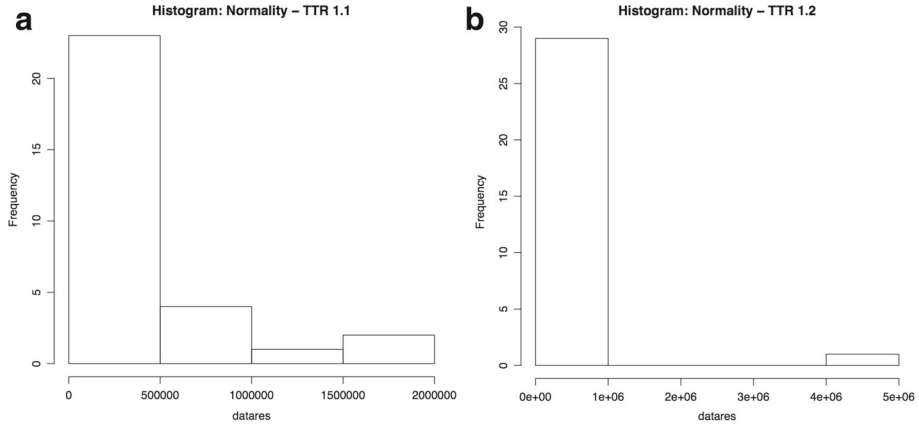

Fig. 11 Experiment 1: Histograms. **a** TTR1.1; **b** TTR 1.2 - 5 and 6 Strengths

Threats to population refer to how significant is the selected samples of the population. For our study, the ranges of strengths, parameters, and values are the determining points for this threat. Note that for such a study, the possibility of combination of strengths and parameters/values is literally infinite. However, we believe that our choice of the set of samples is significant (80) with strengths spanning from 2 to 6. Also, recall that the samples were determined completely randomly (by combining parameters, values, and strengths), as well as the input order of parameters and values was also random (for the 5 executions addressing nondeterminism). With this, we guarantee one of the basic principles of the sampling process which is the randomness to avoid selection bias.

Ecological threats refer to the degree to which the results may be generalized between different configurations. Pre-test effects, Post-test effects, and the Hawthorne effects (due to the participants simply feel stimulated by knowing that they are participating in an innovative experiment) are some of these threats. The participants in our experiment are the instances/samples composed of parameters, values and strengths and, therefore, this type of threat does not apply to our case.

6 Controlled experiment 2: TTR 1.2 × other solutions

In this section, we present a second controlled experiment where we compare TTR 1.2 with five other significant greedy approaches for unconstrained CIT test case generation. Many characteristics of this second controlled experiment ressemble the first one (Section 4). We emphasize here the main differences and point to this previous section whenever necessary.

6.1 Definition and context

The aim of this experiment is to compare TTR 1.2 with five other greedy algorithms/tools for unconstrained CIT: IPOG-F (Forbes et al. 2008), jenny (Jenkins 2016), IPO-TConfig (Williams 2000), PICT (Czerwonka 2006), and ACTS (Yu et al. 2013). These algorithms/tools have been selected due to their relevance for unconstrained CIT via greedy strategies.

The IPO algorithm (Lei and Tai 1998) is the basis for several other solutions such as IPOG, IPOG-D (Lei et al. 2007), IPOG-F, IPOG-F2 (Forbes et al. 2008), IPOG-C

Table 14 Experiment 1 - Results of the Wilcoxon test

Comparison	Strength							
	All				Low			
	p	|*z*|	Reject H0.x?	Winner	*p*	|*z*|	Reject H0.x?	Winner
TTR 1.1 x TTR 1.2	0.3152	1.0067	No	-	9.54E-04	4.0145	Yes	TTR 1.1

(Yu et al. 2013), IPO-TConfig (Williams 2000), ACTS (where several versions of IPO are implemented) (Yu et al. 2013), and CitLab (Cavalgna et al. 2013). Thus, we considered three of its variations: own our implementation of IPOG-F (in Java), IPO-TConfig (in Java), and IPOG-F2 implemented within ACTS (in Java). Note that ACTS is probably one of the most popular CIT tools where not only academia but industry professionals have been using it for various purposes (NIST National Institute of Standards and Technology 2015). A tool implemented in C, jenny (Jenkins 2016), has been used in informal (Pairwise 2017) and more formal (Segall et al. 2011) CIT comparisons. PICT (in C++) can be regarded as one baseline greedy tool where other tools have been created based on it (PictMaster 2017).

Like in Section 4, the metrics are cost, measured as the size of the test suites, and efficiency which again refers to the time to generate them. However, to proper measure the time to generate the test suites, we should have access to the source code of the tools in order to instrument them and get more precise and accurate measures. We had only the code of the implementation of TTR 1.2, our own implementation of IPOG-F, and jenny. Thus, we could not measure the time to generate the test cases due to IPO-TConfig, PICT, and ACTS (IPOG-F2). Moreover, note that the time measurements may be influenced by different programming languages within the cost-efficiency evaluation (TTR 1.2, IPOG-F, and jenny). In this case, we implemented TTR 1.2 not only in Java but also in C too in order to address a possible evaluation bias in the time measures when comparing TTR 1.2 against the other solutions. To sum up, we decided to perform two evaluations:

- Cost-Efficiency (multi-objective). Here, we focused on TTR 1.2, IPOG-F, and jenny since these were the solutions we had the source code and could properly measure the time to generate the test suites. Hence, we compared TTR 1.2 (in Java) with IPOG-F (in Java), and TTR 1.2 (in C) with jenny (in C);
- Cost (single objective). In this case, we compared TTR 1.2 (only in Java since efficiency is not considered here and thus time does not matter) with PICT, IPO-TConfig, and ACTS.

With respect to the subjects, the same 80 participants of Section 4 were used (Table 11 and full data are in (Balera and Santiago Júnior 2017)).

6.2 Hypotheses and variables

Hypotheses of this second experiment are:

Table 15 Experiment 1 - Results of the Wilcoxon test

Comparison	Strength							
	Medium				High			
	p	|*z*|	Reject H0.x?	Winner	*p*	|*z*|	Reject H0.x?	Winner
TTR 1.1 x TTR 1.2	0.000242	3.4705	Yes	TTR 1.2	0.000809	3.2110	Yes	TTR 1.2

- **Null Hypothesis,** $H_{0.2}$ - There is no difference regarding cost-efficiency between TTR 1.2 (in Java) and IPOG-F (in Java);
- **Alternative Hypothesis,** $H_{1.2}$ - There is difference regarding cost-efficiency between TTR 1.2 (in Java) and IPOG-F (in Java);
- **Null Hypothesis,** $H_{0.3}$ - There is no difference regarding cost-efficiency between TTR 1.2 (in C) and jenny (in C);
- **Alternative Hypothesis,** $H_{1.3}$ - There is difference regarding cost-efficiency between TTR 1.2 (in C) and jenny (in C);
- **Null Hypothesis,** $H_{0.4}$ - There is no difference regarding cost between TTR 1.2 (in Java) and PICT;
- **Alternative Hypothesis,** $H_{1.4}$ - There is difference regarding cost between TTR 1.2 (in Java) and PICT;
- **Null Hypothesis,** $H_{0.5}$ - There is no difference regarding cost between TTR 1.2 (in Java) and IPO-TConfig;
- **Alternative Hypothesis,** $H_{1.5}$ - There is difference regarding cost between TTR 1.2 (in Java) and IPO-TConfig;
- **Null Hypothesis,** $H_{0.6}$ - There is no difference regarding cost between TTR 1.2 (in Java) and ACTS;
- **Alternative Hypothesis,** $H_{1.6}$ - There is difference regarding cost between TTR 1.2 (in Java) and ACTS.

The *independent* variable is the algorithm/tool for CIT test case generation for both assessments (cost-efficiency, cost). The *dependent* variables are the number of generated test cases (cost evaluation), and this number of test cases in addition to the time to generate each set of test cases in a multi-objective perspective as in the previous section (cost-efficiency evaluation).

6.3 Description of the experiment

The general description of both evaluations (cost-efficiency, cost) of this second study is basically the same as shown in Section 4. Algorithms/tools were subjected to each one of the 80 test instances, one at a time, and the outcome was recorded. Cost is the number of generated test cases, and efficiency was obtained via instrumentation of the source code with the same computer previously mentioned.

For the multi-objective cost-efficiency evaluation (IPOG-F, jenny), we followed the same two steps previously mentioned: transformation of the cost-efficiency (two-dimensional) representation into a one-dimensional one and usage of statistical tests, such as the t-test or the nonparametric Wilcoxon test (Signed Rank) (Kohl 2015), to compare each pair of test suites (TTR 1.2 and other). To address the nondeterminism of the algorithms/tools, we again generated test cases with 5 variations in the order of parameters and values, and took into account the average of these 5 assessments for the statistical tests. Hence, we obtained the points (cA_i, tA_i) and calculated the Euclidean distances from the optimal point (0,0) to (cA_i, tA_i). Then, we checked data normality and, based on the result of normality, we used the the paired, two-sided t-test with $\alpha = 0.05$ (normal data) or the nonparametric paired, two-sided Wilcoxon test (Signed Rank) or its Asymptotic version with $\alpha = 0.05$ (non-normal data).

For the evaluation of cost (PICT, IPO-TConfig, ACTS), we did not need to transform from two into one dimension because it is a single dimension problem. The optimal point here is the value 0 and the Euclidean distance from 0 to cA_i (average cost of the algorithms A for each instance i, $1 \leq i \leq 80$) is $|0 - cA_i| = |cA_i|$. We then performed the statistical evaluation just as in the multi-objective case.

6.4 Results, discussion and validity

In this section, we present the outcomes of both assessments of our second controlled experiment. Like in the first controlled experiment, to compare TTR 1.2 with IPOG-F, jenny, PICT, IPO-TConfig, and ACTS, we considered four evaluation classes: all, low, medium, and high strengths. Table 16 presents the Euclidean distances of part of the 80 samples (all strenghts class only; complete data are in (Balera and Santiago Júnior 2017)) and the average values, $\bar{x}$. Table 17 presents results of the analysis of data normality ($p - value$ (p) and *skewness*) where we can see all evaluation classes. In this table, **Sol 1** is the other solution and **Sol 2** is TTR 1.2. Figures 12 and 13 present the Q-Q plots and histograms for all strengths, Figs. 14 and 15 present the Q-Q plots and histograms for lower strengths, Figs. 16 and 17 present the Q-Q plots and histograms for medium strengths, and Figs. 18 and 19 present the Q-Q plots and histograms for higher strengths, respectively.

Again we note that all these data did not come from a normally distribution population. The nonparametric paired, two-sided Wilcoxon test (Signed Rank) or its variation (Asymptotic) where then applied. Table 18 presents the $p - value$, p, $|z|$, and additional information for classes all and low strengths while Table 19 shows the results for medium and high strengths. We should mention that in 23 instances (3 with $strength = 4$, 12 with $strength = 5$, and 8 with $strength = 6$) jenny was not able to generate test cases, in some input order of the parameters, due to out of memory issue. Specifically, jenny failed to finish when the test suite size was more than 1,000 test cases. Similar outcomes happened in IPO-TConfig: even if we waited for about 6 hours, it did not generate anything out and hence the tool did not create test cases in 20 instances (3 with $strength = 4$, 9 with $strength = 5$, and 8 with $strength = 6$). In these cases, we adopted a policy penalty: in order to consider these unsuccessful participants, we doubled the respective measure we obtained (average value of the Euclidean distance) due to TTR 1.2 to be the one of jenny and IPO-TConfig. We believe that this is a fair decision because TTR 1.2 could finish generating test cases for all 80 instances.

As shown in Table 18, for class all strengths, two Null Hypotheses were rejected: $H_{0.2}$ (TTR 1.2 $\times$ IPOG-F) and $H_{0.5}$ (TTR 1.2 $\times$ IPO-TConfig). TTR 1.2 was better (lowest average value of Euclidean distances) than IPO-TConfig but it was worse than IPOG-F. There is no difference between TTR 1.2 and jenny, PICT, and ACTS.

As in controlled experiment 1, TTR 1.2 did not demonstrate good performance for low strengths. There is no difference between TTR 1.2 and IPO-TConfig. In all the other comparisons, the Null Hypothesis was rejected and TTR 1.2 was worse than the other solutions. This can be attributed to the fact that the algorithm focuses on test cases that have parameter interactions that generate a large amount of t-tuples, which is usually seen in test cases with larger strenghts. This can be verified by the fact that the algorithm gives priority to just covering the interaction of parameters with the greatest amount of t-tuples.

Table 16 Experiment 2 - Results of the analysis of Euclidean Distance (all strengths)

i	Euclidean Distance – ACTS	Euclidean Distance – IPOGF	Euclidean Distance – jenny	Euclidean Distance – PICT	Euclidean Distance - IPO-TConfig
1	30.2	34.25376	3.126339e+01	30.0	29.4
2	24.0	24.45240	2.403032e+01	24.0	24.0
3	20.0	23.64149	2.302671e+01	22.0	21.2
4	30.0	36.82662	3.263703e+01	33.0	32.0
5	36.0	41.68117	3.824911e+01	36.0	37.2
6	56.0	64.99261	5.747287e+01	59.0	58.4
7	60.4	63.10277	6.262871e+01	60.0	65.0
8	73.8	88.55123	7.726934e+01	73.0	75.4
9	225.2	281.81128	2.491693e+02	244.0	238.0
10	160.2	151.87482	1.640817e+02	162.0	164.0
…	…	…	…	…	…
20	2523.0	117983.01608	4.123941e+04	2530.0	2590.8
21	768.8	1070.65296	8.145790e+02	758.0	764.8
22	961.0	1163.62637	1.009413e+03	934.0	957.0
23	614.2	799.74498	6.468449e+02	603.0	640.8
24	1274.8	3309.79266	7.856099e+03	1271.0	2268.0
25	3414.4	140138.76666	2.634307e+05	3374.0	6308.0
26	3645.4	133219.16188	2.393899e+05	3583.0	4237.6
27	1311.0	5700.77912	4.674461e+04	1318.0	2324.0
28	943.8	1249.43131	7.111379e+03	940.0	1620.0
29	2348.8	13992.60905	8.271590e+04	2303.0	4324.0
30	487.2	556.19741	5.392609e+02	504.0	451.2
…	…	…	…	…	…
40	308.2	344.40546	3.160750e+02	303.0	317.0
41	1310.6	5352.50891	6.203907e+05	1324.0	2698.8
42	188.4	213.49005	1.940438e+02	187.0	210.4
43	3893.6	301379.27998	1.580947e+05	3900.0	7780.0
44	1082.4	4493.69065	9.001308e+03	1094.0	2488.8
45	228.2	230.91280	2.512118e+02	230.0	227.0
46	513.6	554.51922	5.447095e+02	506.0	536.6
47	1262.2	1672.73134	7.159649e+03	1275.0	2524.0
48	64.4	86.16728	7.211915e+01	68.0	71.6
49	12.8	17.02939	1.540589e+01	13.0	15.0
50	72.2	97.43100	7.694289e+01	74.0	76.6
…	…	…	…	…	…
60	235.0	1038.16209	2.537152e+02	236.0	255.6
61	112.4	298.63081	1.081020e+02	106.0	108.2
62	151.0	169.40106	1.593580e+02	149.0	168.0
63	239.0	1093.02578	2.375884e+02	234.0	234.0
64	1440.0	1440.79978	2.880001e+03	1440.0	1440.0
65	576.0	576.07347	5.765385e+02	523.4	576.0
66	5045.8	398648.10830	2.641278e+05	5056.0	10084.0
67	4235.0	22245.98615	2.951324e+05	4166.0	7696.0
68	160.6	146.47266	1.741181e+02	162.0	144.0
69	15632.6	177.68331	9.149951e+06	15743.0	322.0
70	1566.6	771.97049	8.203877e+05	1565.0	3049.6
…	…	…	…	…	…
80	42.4	44.41126	4.600265e+01	44.0	43.0
$\bar{x}$	962.485	21781.67	172885.3	958.88	1290.382

Table 17 Experiment 2 - Results of the analysis of data normality

Comparison	Strength	*p* - Sol 1	skewness - Sol 1	*p* - Sol 2	skewness - Sol 2
IPOG-F x TTR 1.2	All	2.2E-16	3.821529	2.2E-16	8.186584
jenny x TTR 1.2	All	< 2.2E-16	6.215192	< 2.2E-16	5.392207
PICT x TTR 1.2	All	1.28E-15	5.195747	5.87E-9	2.116826
IPO-TConfig x TTR 1.2	All	6.95E-13	2.238054	5.87E-9	2.115547
ACTS x TTR 1.2	All	1.46E-15	5.134732	5.87E-9	2.116826
IPOG-F x TTR 1.2	Low (2)	1.53E-02	2.528968	5.03E-06	4.000011
jenny x TTR 1.2	Low (2)	0.03384	0.9001327	9.09E-06	3.742379
PICT x TTR 1.2	Low (2)	0.03585	0.9637761	0.009991	0.9031568
IPO-TConfig x TTR 1.2	Low (2)	0.03183	0.8693103	0.005334	1.138218
ACTS x TTR 1.2	Low (2)	0.02786	1.002096	0.009991	0.9031568
IPOG-F x TTR 1.2	Medium (3 and 4)	3.17E-07	4.418432	1.03E-04	2.247599
jenny x TTR 1.2	Medium (3 and 4)	4.30E-07	2.746003	9.73E-05	2.818052
PICT x TTR 1.2	Medium (3 and 4)	1.06E-02	1.729118	1.3E-05	1.696949
IPO-TConfig x TTR 1.2	Medium (3 and 4)	6.35E-05	2.006703	1.3E-05	1.696949
ACTS x TTR 1.2	Medium (3 and 4)	1.33E-02	1.688993	1.3E-05	1.696949
IPOG-F x TTR 1.2	High (5 and 6)	5.29E-8	1.965489	2.99E-008	5.084482
jenny x TTR 1.2	High (5 and 6)	1.26E-06	3.877021	2.30E-11	3.821616
PICT x TTR 1.2	High (5 and 6)	6.66E-8	3.48041	0.001581	0.9643054
IPO-TConfig x TTR 1.2	High (5 and 6)	0.0003957	1.049076	0.001581	0.9643054
ACTS x TTR 1.2	High (5 and 6)	7.75E-005	3.439773	0.001581	0.9643054

For medium strengths, TTR 1.2 had alternate results. While the Null Hypothesis $H_{0.6}$ (TTR 1.2 × ACTS) could not be rejected and our algorithm was better than IPO-TConfig, IPOG-F, jenny, and PICT surpassed TTR 1.2.

The greatest advantage of TTR 1.2 turned out to be again for higher strengths. Recall that TTR 1.2 does not create the matrix of t-tuples at the beginning, and this can potentially benefit our solution compared with the other five for higher strengths. Note that

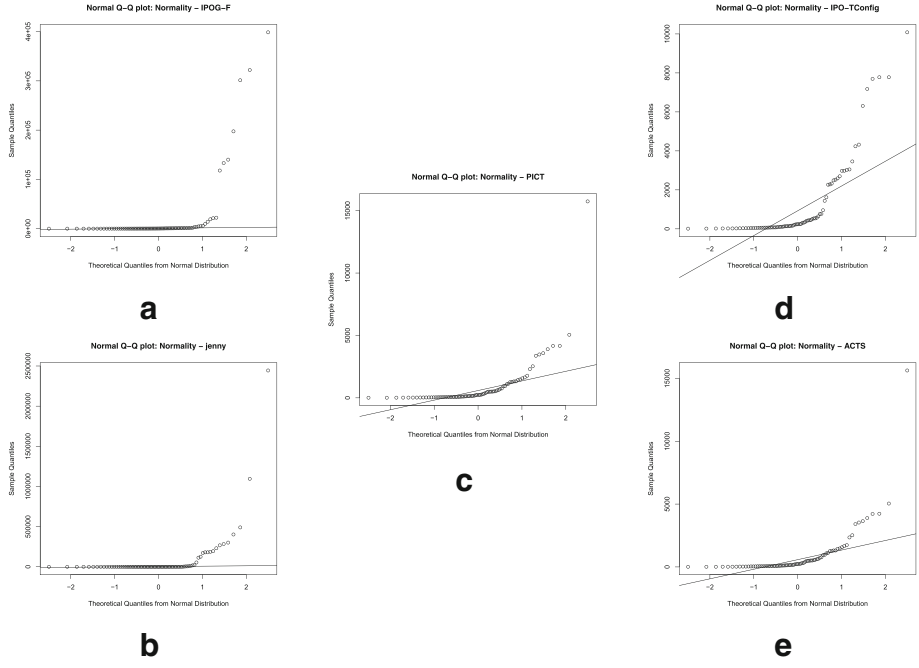

Fig. 12 Experiment 2: Q-Q plots. **a** IPOG-F; **b** jenny; **c** PICT; **d** IPO-TConfig; **e** ACTS - All Strengths

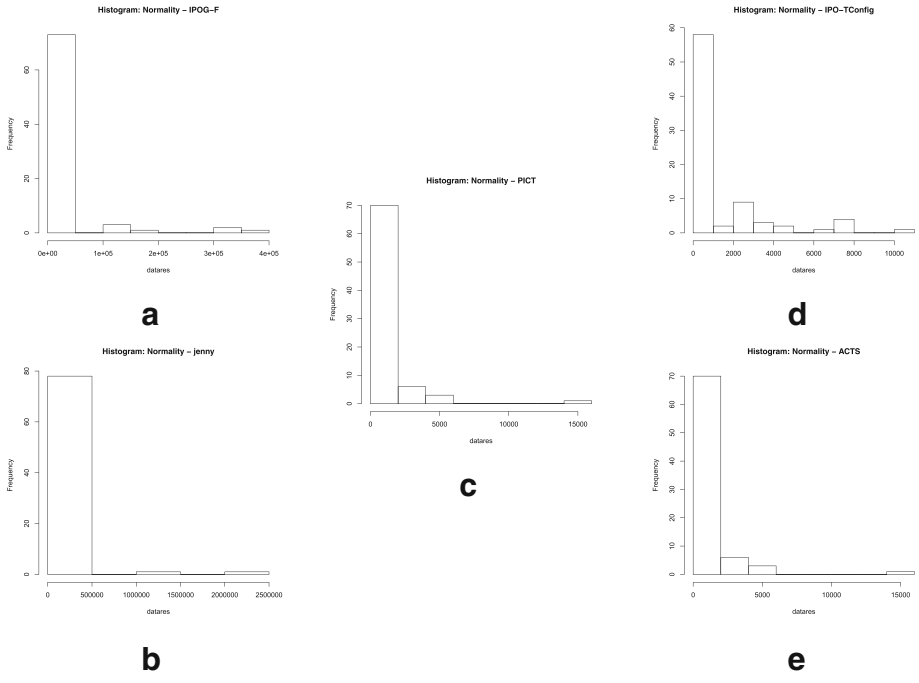

Fig. 13 Experiment 2: Histograms. **a** IPOG-F; **b** jenny; **c** PICT; **d** IPO-TConfig; **e** ACTS - All Strengths

TTR 1.2 was better than jenny, PICT, IPO-TConfig, and ACTS. The only exception is the comparison against IPOG-F where the Null Hypothesis, $H_{0.2}$, could not be rejected and thus there is no statistical difference between both approaches.

In general, we can say that IPOG-F presented the best performance compared with TTR 1.2, because IPOG-F was better for all strengths, as well as lower and medium strengths. For higher strengths, there was a statistical draw between both approaches. An

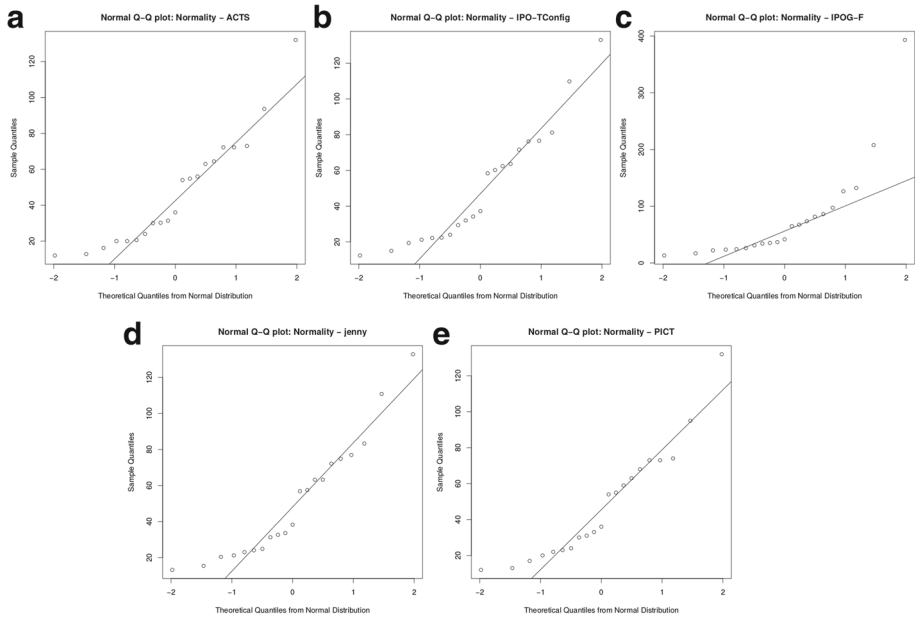

Fig. 14 Experiment 2: Q-Q plots. **a** ACTS; **b** IPO-TConfig; **c** IPOG-F; **d** jenny; **e** PICT - Lower Strengths

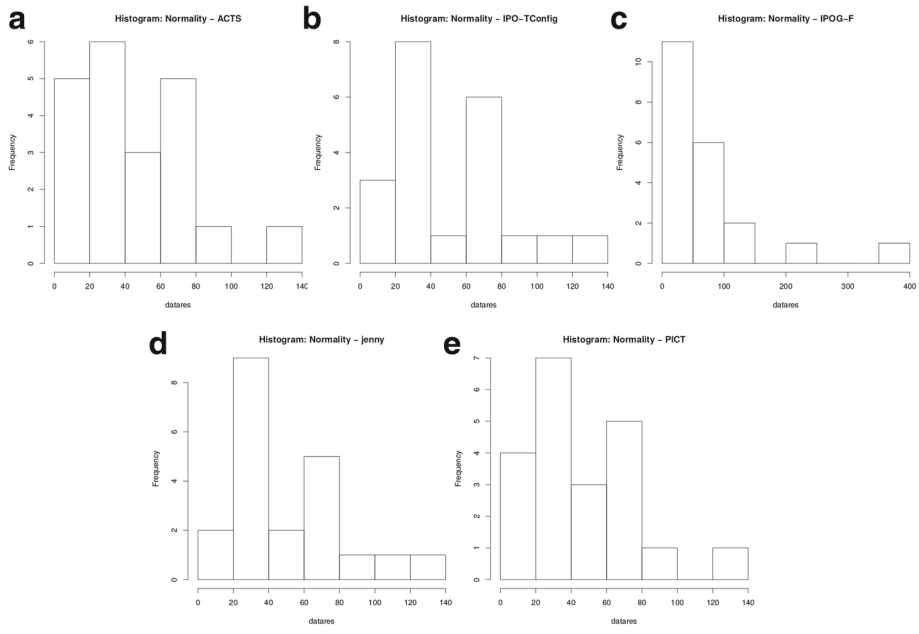

Fig. 15 Experiment 2: Histograms. **a** ACTS; **b** IPO-TConfig; **c** IPOG-F; **d** jenny; **e** PICT - Lower Strengths

explanation for the fact that IPOG-F is better than TTR 1.2 is that TTR 1.2 ends up making more interactions than IPOG-F. In general, we might say that efficiency of IPOG-F is better than TTR 1.2 which influenced the cost-efficiency result. However, if we look at cost in isolation for all strengths, the average value of the test suite size generated via TTR 1.2 (734.50) is better than IPOG-F (770.88).

As we have just stated, for higher strengths, TTR 1.2 is better than two IPO-based approaches (IPO-TConfig and ACTS/IPOG-F2) but there is no difference if we consider

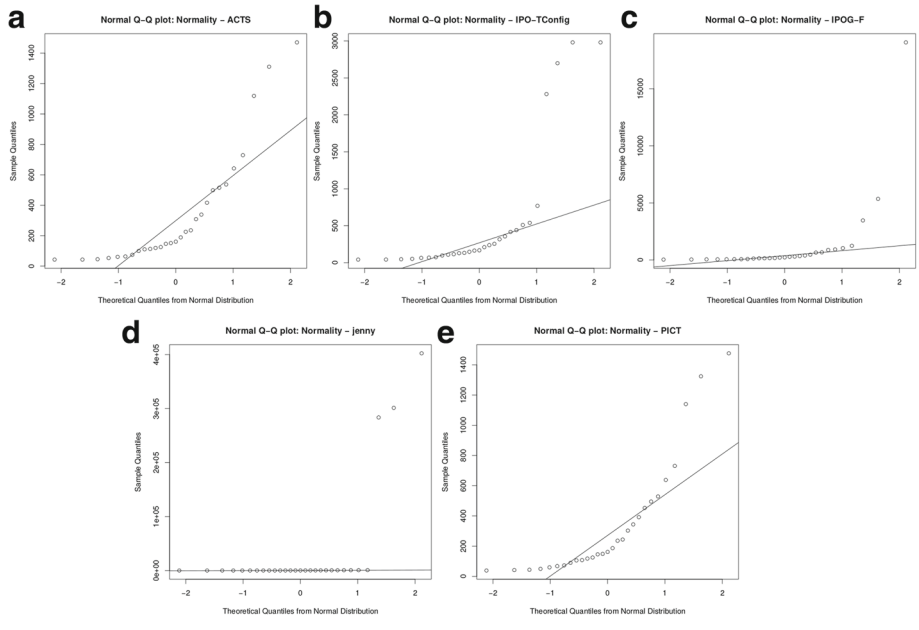

Fig. 16 Experiment 2: Q-Q plots. **a** ACTS; **b** IPO-TConfig; **c** IPOG-F; **d** jenny; **e** PICT - Medium Strengths

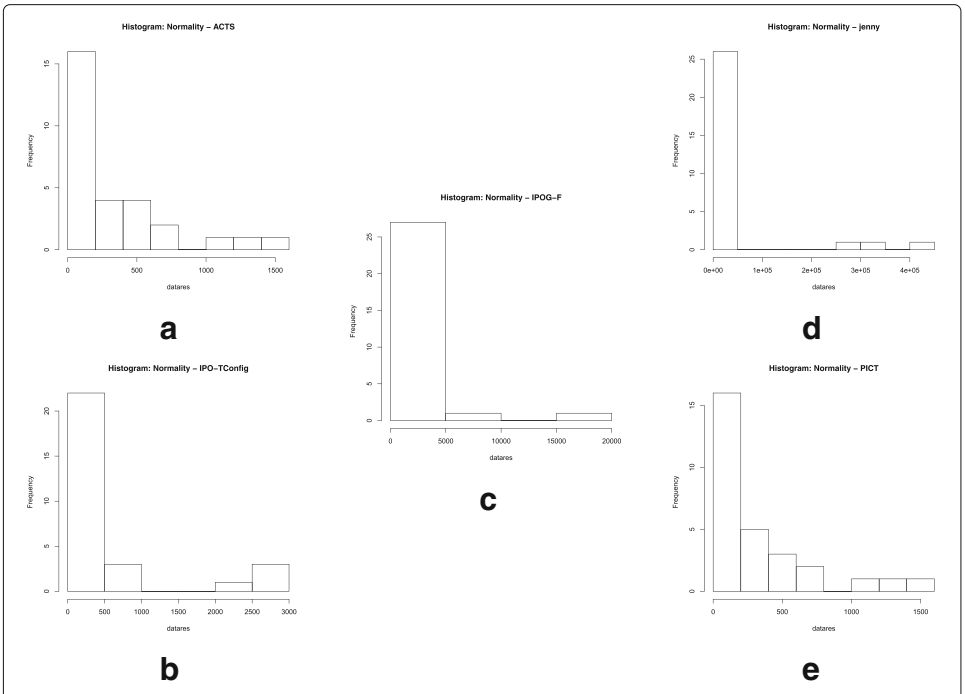

Fig. 17 Experiment 2: Histograms. **a** ACTS; **b** IPO-TConfig; **c** IPOG-F; **d** jenny; **e** PICT - Medium Strengths

our own implementation of IPOG-F and TTR 1.2. This can be explained as follows. The way the array that stores all t-tuples is constructed influences the order in which the t-tuples are evaluated by the algorithm. However, it is not described how this should be done in IPOG-F, leaving it to the developer to define the best way. As the order in which the parameters are presented to the algorithms alters the number of test cases generated,

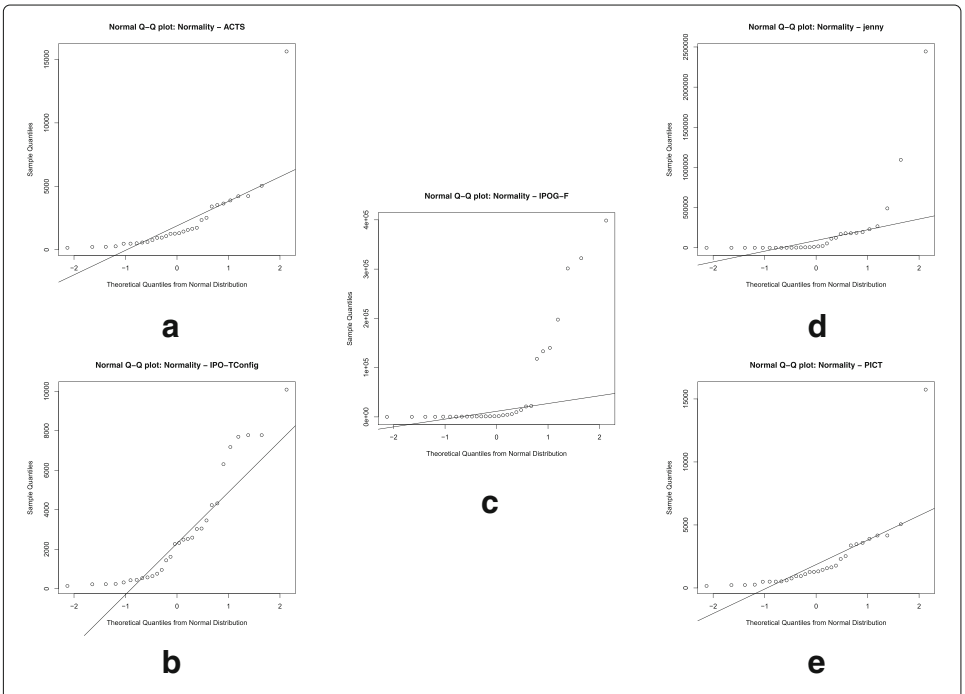

Fig. 18 Experiment 2: Q-Q plots. **a** ACTS; **b** IPO-TConfig; **c** IPOG-F; **d** jenny; **e** PICT - Higher Strengths

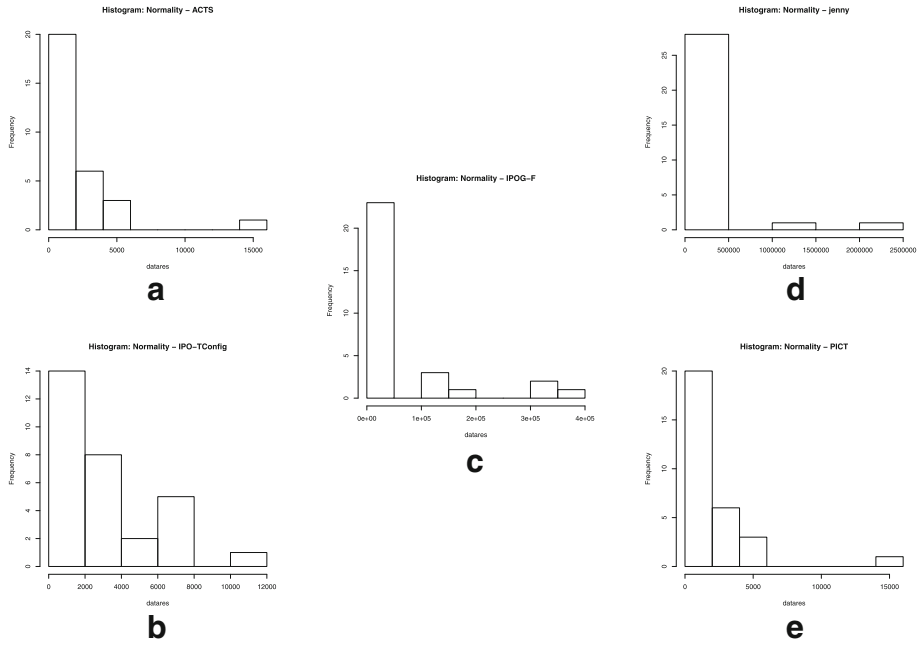

Fig. 19 Experiment 2: Histograms. **a** ACTS; **b** IPO-TConfig; **c** IPOG-F; **d** jenny; **e** PICT - Higher Strengths

as previously stated, the order in which the t-tuples are evaluated can also generate a certain difference in the final result.

The conclusion of the two evaluations of this second experiment is that our solution is better and quite attractive for the generation of test cases considering higher strengths (5 and 6), where it was superior to basically all other algorithms/tools. Certainly, the main fact that contributes to this result is the non-creation of the matrix of t-tuples at the beginning which allows our solution to be more scalable (higher strengths) in terms of cost-efficiency or cost compared with the other strategies. However, for low strengths, other greedy approaches, like IPOG-F, may be better alternatives.

As before and by making a comparison between pairs of solutions (TTR 1.2 $\times$ other), in both assessments (cost-efficiency and cost), we can say that we have a high conclusion, internal, and construct validity. Regarding the external validity, we believe that we selected a significant population for our study. Detailed explanations have been given in Section 5.1 and are valid here.

Table 18 Results of the Wilcoxon test

Comparison	Strength							
	All				Low			
	p	|*z*|	Reject H0.x?	Winner	*p*	|*z*|	Reject H0.x?	Winner
IPOG-F x TTR 1.2	9.26E-05	5.3431	Yes	IPOG-F	1.34E-02	3.8060	Yes	IPOG-F
jenny x TTR 1.2	0.4913	0.6907	No	-	2.86E-03	3945	Yes	jenny
PICT x TTR 1.2	0.3769	0.8862	No	-	0.0002899	3.2881	Yes	PICT
IPO-TConfig x TTR 1.2	3.13E-02	4.1664	Yes	TTR 1.2	0.05584	1.9117	No	-
ACTS x TTR 1.2	0.3392	0.9584	No	-	0.0001068	3.4623	Yes	ACTS

Table 19 Results of the Wilcoxon test (medium and high strengths)

Comparison	Strength							
	Medium				High			
	p	|*z*|	Reject H0.x?	Winner	*p*	|*z*|	Reject H0.x?	Winner
IPOG-F x TTR 1.2	7.45E-06	4.6814	Yes	IPOG-F	0.2801	1.1004	No	-
jenny x TTR 1.2	0.003511	2.8435	Yes	jenny	0.0002316	3.4863	Yes	TTR 1.2
PICT x TTR 1.2	0.03676	2.0782	Yes	PICT	0.002109	2.9732	Yes	TTR 1.2
IPO-TConfig x TTR 1.2	0.008742	2.5732	Yes	TTR 1.2	2.76E-03	4.1672	Yes	TTR 1.2
ACTS x TTR 1.2	0.2276	1.2217	No	-	0.0007162	3.2335	Yes	TTR 1.2

7 Related work

In this section we present some relevant studies related to greedy algorithms for CIT. The IPO algorithm (Lei and Tai 1998) is one very traditional solution designed for pairwise testing. Several approaches are based on IPO such as IPOG, IPOG-D (Lei et al. 2007), IPOG-F, IPOG-F2 (Forbes et al. 2008), IPOG-C (Yu et al. 2013), IPO-TConfig (Williams 2000), ACTS (where IPOG, IPOG-D, IPOG-F, IPOG-F2 are implemented)(Yu et al. 2013), and CitLab (Cavalgna et al. 2013). All IPO-based proposals have in common the fact that they perform horizontal and vertical growths to construct the final test suite. Moreover, some need two auxiliary matrices which may decrease its performance by demanding more computer memory. Such algorithms accomplish exhaustive comparisons within each horizontal extension which may penalize efficiency.

IPOG-F (Forbes et al. 2008) is an adaptation of the IPOG algorithm (Lei et al. 2007). Through two main steps, horizontal and vertical growths, an MCA is built. Both growths work based on an initial solution. The algorithm is supported by two auxiliary matrices which may decrease its performance by demanding more computer memory to use. Moreover, the algorithm performs exhaustive comparisons within each horizontal extension which may cause longer execution. On the other hand, TTR 1.2 only needs one auxiliary matrix to work and it does not generate, at the beginning, the matrix of t-tuples. These features make our solution better for higher strengths (5, 6) even though we did not find statistical difference when we compared TTR 1.2 with our own implementation of IPOG-F (Section 6.4).

IPO-TConfig is an implementation of IPO in the TConfig tool (Williams 2000). The TConfig tool can generate test cases based on strengths varying from 2 to 6. However, it is not entirely clear whether the IPOG algorithm (Lei et al. 2007) was implemented in the tool or if another approach was chosen for t-way testing. In our empirical evaluation, TTR 1.2 was superior to IPO-TConfig not only for higher strengths (5, 6) but also for all strengths (from 2 to 6). Moreover, IPO-TConfig was unable to generate test cases in 25% of the instances (strengths 4, 5, 6) we selected.

The ACTS tool (Yu et al. 2013) is one of the most used CIT tools to date. Several variations of IPO are implemented in ACTS: IPOG, IPOG-D (Lei et al. 2007), IPOG-F, and IPOG-F2 (Forbes et al. 2008). The implementation of our algorithm performed better in terms of cost, compared with IPOG-F2/ACTS, for higher strengths. However, both solutions performed similarly when we considered all strengths.

IPOG-C (Yu et al. 2013) generates MCAs considering constraints. It is an adaptation of IPOG where constraint handling is provided via a SAT solver. The greatest contribution are three optimizations that seek to reduce the number of calls of the SAT solver. As

IPOG-C is based on IPOG, it accomplishes exhaustive comparisons in the horizontal growth which may lead to a longer execution. Besides, each t-tuple is evaluated to see if it is valid or not.

The algorithm implemented in the PICT tool (Czerwonka 2006) has two main phases: preparation and generation. In the first phase, the algorithm generates all t-tuples to be covered. In the second phase, it generates the MCA. The generation of all t-tuples which can often be a bad thing, since many tuples require large disk space for storage. With respect to the application of the tool, this tool is best applied in strenghts of low value as an example, there is no study (Yamada et al. 2016). Other tools have been created based on PICT (PictMaster 2017).

The jenny tool is implemented in C (Jenkins 2016). It is a light greedy tool but one of its limitation is the number of parameters it handles: from 2 to 52. In the controlled experiment we performed, TTR 1.2 was superior to jenny for higher strengths (5, 6) but they presented similar performances for all strengths (from 2 to 6). In 27.5% of the samples (strengths 4, 5, 6), jenny could not create test cases as we have mentioned before.

Automatic Efficient Test Generator (AETG) (Cohen et al. 1997) is based on algorithms that use ideas of statistical experimental design theory to minimize the number of tests needed for a specific level of test coverage of the input test space. AETG generates test cases by means of Experimental Designs (ED) (Cochran and Cox 1950) which are statistical techniques used for planning experiments so that one can extract the maximum possible information based on as few experiments as possible. It makes use of its greedy algorithms and the test cases are constructed one at a time, i.e. it does not use an initial solution.

In (Cavalgna et al. 2013), a new tool is presented for generating MCAs with constraint handling support: CitLab. Like ACTS, CitLab has several algorithms for test suite generation: AETG, IPO, and others. The bottom of line is that test case generation is only one of the characteristics of the tool. Like ACTS, CitLab does not present a new algorithm as it just implements algorithms proposed in the literature. Hence, the same limitations of the existing proposals are also here.

The Feedback Driven Adptative Combinatorial Testing Process (FDA-CIT) algorithm is shown in (Yilmaz et al. 2014). At each iteration of the algorithm, verification of the masking of potential defects is accomplished, isolating their probable causes and then generating a new configuration which omits such causes. The idea is that masked deffects

Table 20 Greedy algorithms/tools for CIT

Algorithm/Tool	1	2	3	4	5	6	7	8	9
IPOG-F ((Forbes et al. 2008))	*	-	-	-		*		*	
AETG (Cohen et al. 1997)	-		-	-				*	*
PICT ((Czerwonka 2006))	*		-	-	*			*	
ACTS ((Yu et al. 2013))	-	-	-	-		-		*	*
CitLab ((Cavalgna et al. 2013))	-		-	-				*	
IPOG-C ((Yu et al. 2013))	*	-		-		*		*	*
FDA-CIT ((Yilmaz et al. 2014))	*		-	-				*	
jenny ((Jenkins 2016))	*		-			-	*	*	
TTR - 1.2	*	*	*	*	-	*	*	*	-

Caption: 1 = new algorithm, 2 = no more than 1 auxiliary matrix, 3 = evaluated via controlled experiments/quasiexperiments, 4 = no generation of full matrix of t-tuples, 5 = time optimization, 6 = works based on an initial solution, 7 = do not impose parameter/value input restriction, 8 = support for t-way testing, 9 = support for constraints

exist and that the proposed algorithm provides an efficient way of dealing with this situation before test execution. However, there is no assessment about the cost of the algorithm to generate MCAs.

In order to better compare the previous studies with our algorithm, TTR 1.2, in Table 20 we show some main characteristics of all the algorithms/tools. In this table, * means that the characteristic is present, - means that it is not present, and empty (blank space) means that either it is not totally evident that the algorithm/tool has such a feature or it is not applicable.

8 Conclusions

This paper presented a novel CIT algorithm, called TTR, to generate test cases specifically via the MCA technique. TTR produces an MCA M, i.e. a test suite, by creating and reallocating t-tuples into this matrix M, considering a variable called *goal* (ζ). TTR is a greedy algorithm for unconstrained CIT.

TTR was implemented in Java and C (TTR 1.2) and we developed three versions of our algorithm. In this paper, we focused on the description of versions 1.1 and 1.2 since version 1.0 was detailed elsewhere (Balera and Santiago Júnior 2015).

We carried out two rigorous evaluations to assess the performance of our proposal. In total, we performed 3,200 executions related to 8 solutions (80 instances $\times$ 5 variations $\times$ 8). In the first controlled experiment, we compared versions 1.1 and 1.2 of TTR in order to know whether there is significant difference between both versions of our algorithm. In such experiment, we jointly considered cost (size of test suites) and efficiency (time to generate the test suites) in a multi-objective perspective. We conclude that TTR 1.2 is more adequate than TTR 1.1 especially for higher strengths (5, 6). This is explained by the fact that, in TTR 1.2, we no longer generate the matrix of t-tuples (Θ) but rather the algorithm works on a t-tuple by t-tuple creation and reallocation into M. This benefits version 1.2 so that it can properly handle higher strengths.

Having chosen version 1.2, we conducted another controlled experiment where we confronted TTR 1.2 with five other greedy algorithms/tools for unconstrained CIT: IPOG-F (Forbes et al. 2008), jenny (Jenkins 2016), IPO-TConfig (Williams 2000), PICT (Czerwonka 2006), and ACTS (Yu et al. 2013). In this case, we carried out two evaluations where in the first one we compared TTR 1.2 with IPOG-F and jenny since these were the solutions we had the source code (to precisely measure the time). Moreover, to address a possible evaluation bias in the time measures when comparing TTR 1.2 against jenny (developed in C), we also implemented it in C in addition to the standard implementation in Java. Hence, a cost-efficiency (multi-objective) evaluation was performed. In the second assessment, we did a cost (single objective) evaluation where TTR 1.2 was compared with PICT, IPO-TConfig, and ACTS. The conclusion is as previously stated: TTR 1.2 is better for higher strengths (5, 6) where only in one case our solution is not superior (in the comparison with IPOG-F where we have a draw). The fact of not creating the matrix of t-tuples at the beginning explains this result.

Therefore, considering the metrics we defined in this work and based on both controlled experiments, TTR 1.2 is a better option if we need to consider higher strengths (5, 6). For lower strengths, other solutions, like IPOG-F, may be better alternatives.

Thinking about the testing process as a whole, one important metric is the time to execute the test suite which eventually may be even more relevant than other metrics. Hence,

we need to run multi-objective controlled experiments where we execute all the test suites (TTR 1.1 × TTR 1.2; TTR 1.2 × other solutions) probably assigning different weights to the metrics. We also need to investigate the parallelization of our algorithm so that it can perform even better when subjected to a more complex set of parameters, values, strengths. One possibility is to use the Compute Unified Device Architecture/Graphics Processing Unit (CUDA/GPU) platform (Ploskas and Samaras 2016). We must develop other multi-objective controlled experiment addressing effectiveness (ability to detect defects) of our solution compared with the other five greedy approaches.

Endnotes

[1] Despite this classification, some algorithms/tools are both SAT and greedy-based.

[2] Some authors (Kuhn et al. 2013; Cohen et al. 2003) abbreviate a Mixed-Level Covering Array as CA too. However, as we have made a explicit distinction between Fixed-value and Mixed-Level arrays, we prefer abbreviate it as MCA. Note that an MCA is naturally a Covering Array. We have just used this abbreviation to stress that our work relates to mixed and not fixed arrays.

[3] Θ is a matrix whose order varies. In other words, TTR knows the number of columns beforehand ($|f|$), but the number of rows ($|C|$) depends on the interaction of t-way parameter's values. During the reallocation process, TTR removes the rows until Θ is empty.

Abbreviations

ACTS: Advanced combinatorial test system; AETG: Automatic efficient test generator; CA: Coverage array; CIT: Combinatorial interaction test; CUDA: Compute unified device architecture; ED: Experimental designs; GA: Genetic algorithm; GPU: Graphics processing unit; IPOG: In parameter order general; IPO-TConfig: In parameter order TConfig; MCA: Mixed-level covering array; MOA: Mixed-level orthogonal array; OA: Orthogonal array; OOP: Object-oriented programming; PICT: Pairwise indepedent combinatorial testing; SA: Simulated annealing; SWPDC: Software for the payload data handling computer; TSA: Tabu search approach; TTR: T-tuple reallocation

Acknowledgements

The authors would like to thank the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)* for supporting this research and Leoni Augusto Romain da Silva for his support in running part of the second controlled experiment.

Funding

This work was partially funded by *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) through a scholarship granted to the first author (JMB).

Availability of data and materials

Full data obtained during the experiments are in (Balera and Santiago Júnior 2017).

Authors' contributions

JMB worked in the definitions and implementations of all three versions of the TTR algorithm, and carried out the two controlled experiments. VASJ worked in the definitions of the TTR algorithm, and in the planning, definitions, and executions of the two controlled experiments. All authors contributed to all sections of the manuscript. All authors read and approved the submitted manuscript.

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Received: 15 April 2017 Accepted: 5 November 2017
Published online: 28 December 2017

References

Ahmed BS (2016) "Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing". Eng Sci Technol, Int J 19(2):737-753. http://www.sciencedirect.com/science/article/pii/S2215098615001706

Balera JM, Santiago Júnior VA (2015) T-tuple Reallocation: An algorithm to create mixed-level covering arrays to support software test case generation. In: 15th International Conference on Computational Science and Its Applications (ICCSA). Springer International, Publishing, Berlin, Heidlberg. pp 503-517

Balera JM, Santiago Júnior VA (2016) "A controlled experiment for combinatorial testing". In: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing. ACM, New York, NY, USA, SAST. pp 2:1-2:10. http://doi.acm.org/10.1145/2X00000.993288.2993289

Balera JM, Santiago Júnior VA (2017) Data set. https://www.dropbox.com/sh/to3a47ncqpliq5l/AACj34JQ9S1I4fzQJf0xPZfva?dl=0. Accessed 17 Oct 2016

Bryce RC, Colbourn CJ (2006) "Prioritized interaction testing for pair-wise coverage with seeding and constraints". Inf Softw Technol 48(10):960–970

Cochran WG, Cox GM (1950) "Experimental designs". John, Wiley & Sons, New York; Chichester

Cohen MB, Dalal SR, Fredman ML, Patton GC (1997) "The AETG system: an approach to testing based on combinatorial design". IEEE Trans Softw Eng 23(7):437–444

Cohen MB, Dwyer MB, Shi J (2008) "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach". IEEE Trans Softw Eng 34(5):633–650

Cohen MB, Gibbons PB, Mugridge WB, Colbourn CJ, Collofello JS (2003) "A variable strength interaction testing of components". In: Proceedings of 27th Annual Int. Comp. Software and Applic. Conf. (COMPSAC). IEEE, USA. pp 413–418

Campanha DN, Souza SRS, Maldonado JC (2010) "Mutation testing in procedural and object-oriented paradigms: An evaluation of data structure programs". In: Brazilian Symposium on Software Engineering. IEEE, USA. pp 90-99

Cavalgna A, Gargantini A, Vavassori P (2013) "Combinatorial interaction testing with citlab". In: Proceedings on 2013 IEEE Sixth International, Conference on Software Testing, Verification and Validation. IEEE, Nova York. pp 376–382

Czerwonka J (2006) "Pairwise testing in the real world: Practical extensions to test-case generators". In: Proceedings 24th Pacific Northwest Software Quality Conference. Academic Press, Portland. pp 285–294

Dalal SR, A Jain NK, Leaton JM, Lott CM, Patton GC, Horowitz B (1999) "Model-based testing in pratice". In: Proceedings 21st International Conference on Software Engineering (ICSE'99). AMC, Nova York. pp 285–294

Delamaro ME, de Lourdes dos Santos Nunes F, de Oliveira RAP (2013) "Using concepts of content-based image retrieval to implement graphical testing oracles". Softw Test Verif Reliab 23:171–198. doi:10.1002/stvr.463

Filho RAM, Vergilio SR (2015) "A Mutation and Multi- objective Test Data Generation Approach for Feature Testing of Software Prod- uct Lines". 29th Brazilian, Symposium on Software Engineering, Belo Hori-zonte

Forbes M, Lawrence J, Lei Y, Kacker RN, Kuhn DR (2008) "Refining the in-parameter-order strategy for constructing covering arrays". J Res Natl Inst Stand Technol 113(5):287–297

Garvin BJ, Cohen MB, Dwyer MB (2011) "Evaluating improvements to a meta-heuristic search for constrained interaction testing". Empirical Soft Eng 16(1):61–102

Hernandez LG, Valdez NR, Jimenez JT (2010) "Construction of mixed covering arrays of variable strength using a tabu search approach". Springer, International Publisher, Berlin, Heidelberg

Huang CY, Chen CS, Lai CE (2016) "Evaluation and analysis of incorporating fuzzy expert system approach into test suite reduction". Inf Softw Technol 79:79–105. http://www.sciencedirect.com/science/article/pii/S0950584916301197

Jenkins B (2016) "Jenny: A pairwise tool". http://burtleburtle.net/bob/math/jenny.html. Accessed 6 June 2016

Khan SUR, Lee SP, Ahmad RW, Akhunzada A, Chang V (2016) "A survey on test suite reduction frameworks and tool"s. Int J Inf Manag 36(6, Part A):963–975. http://www.sciencedirect.com/science/article/pii/S0268401216303437

Kohl M (2015) "Introduction to statistical data analysis with R". bookboon.com, London

Kuhn DR, Wallace DR, Gallo AM (2004) "Software fault interactions and implications for software testing". IEEE Trans Software Eng 30(6):418–421. http://doi.ieeecomputersociety.org/10.1109/TSE.2004.24

Kuhn RD, Kacker RN, Lei Y (2013) "Introduction to Combinatorial Testing". Chapman and Hall/CRC, USA

Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J (2007) "IPOG: A general strategy for t-way software testing"

Lei Y, Tai K-C (1998) "In-Parameter-Order: A test generation strategy for pairwise testing". In: Proceedings of the IEEE Int. Symp. on High-Assurance Syst. Eng. (HASE). IEEE Computer Society Press, USA. pp 254–261

Mathur AP (2008) "Foundations of software testing". Dorling, Kindersley (India), Pearson Education in South Asia, Delhi, India

NIST National Institute of Standards and Technology (2015) "Automated combinatorial testing for software (ACTS)". http://csrc.nist.gov/groups/SNS/acts/. Accessed 29 July 2017

Oliveira RAP (2017) "Test oracles for systems with complex outputs: the case of TTS systems". PhD Thesis, Univesi-dade de São Paulo, Brazil

Pairwise (2017) "Pairwise Testing: Combinatorial Test Case Generation". http://www.pairwise.org/tools.asp. Accessed 29 July 2017

Petke J, Cohen MB, Harman M, Yoo S (2015) "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection". IEEE Trans Softw Eng 41(9):901–924

PictMaster (2017) "Combinatorial testing tool PictMaster". https://osdn.net/projects/pictmaster/. Accessed 29 July 2017

Ploskas N, Samaras N (2016) "GPU Programming in MATLAB". Morgan Kaufmann, Boston. http://www.sciencedirect.com/science/article/pii/B9780128051320099951

Qu X, Cohen MB, Woolf KM (2007) "Combinatorial interaction regression testing: A study of test case generation and prioritization". In: Proc. IEEE Int. Conf. Softw. Maintenance. IEEE Computer Society Press, USA. pp 255–264

Santiago Júnior VA (2011) "Solimva: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications". PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE)

Santiago Júnior VA, Silva FEC (2017) "From Stat- echarts into Model Checking: A Hierarchy-based Translation and Specification Patterns Properties to Generate Test Cases". In: the 2nd Brazilian Symposium, 2017, Fortaleza.

Proceedings of the 2nd, Brazilian Symposium on Systematic and Automated Software Testing - SAST. ACM Press, New York. pp 10–20
Santiago Júnior VA, Vijaykumar NL (2012) "Generating model-based test cases from natural language requirements for space application software". Softw Qual J 20(1):77–143. doi:10.1007/s11219-011-9155-6
Schroeder PJ, Korel B (2000) Black-box test reduction using input-output analysis. In: Harold M (ed). Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00). ACM, New York. pp 173–177
Segall I, Tzoref-Brill R, Farchi E (2011) Using binary decision diagrams for combinatorial test design. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). ACM, New York. pp 254–264
Shapiro SS, Wilk MB (1965) "An analysis of variance test for normality (complete samples)". Biometrika 52(3-4):591
Shiba T, Tsuchiya T, Kikuno T (2004) "Using artificial life techniques to generate test cases for combinatorial testing". In: Proceedings 28th Int. Comput. Softw. Appl. Conf., Des. Assessment Trustworthy Softw.-Based Syst. IEEE Computer Society Press, USA. pp 72–77
Stinson DR (2004) "Combinatorial Designs: Constructions and Analysis". Springer, New York
Tai KC, Lei Y (2002) "A test generation strategy for pairwise testing". IEEE Trans Softw Eng 28(1):109–111
Tzoref-Brill R, Wojciak P, Maoz S (2016) "Visualization of combinatorial models and test plans". In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, USA. pp 144–154
Williams AW (2000) "Determination of test configurations for pairwise interaction coverage". In: Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), August 29 - September 1, 2000, Ottawa, Canada. pp 59–74
Wohlin C, Runeson P, Host M, Ohlsson MC, Regnell B, Wesslén A (2012) "Experimentation in Software Engineering. Springer-Verlag Berlin Heidelberg, Germany
Yamada A, Kitamura T, Artho C, Choi E, Oiwa Y, Biere A (2015) "Optimization of combinatorial testing by incremental SAT solving". IEEE, USA
Yamada A, Biere A, Artho C, Kitamura T, Choi EH (2016) "Greedy combinatorial test case generation using unsatisfiable cores". In: Proceedings of 2016 31st IEEE/ACM International, Conference on Automated Software Engineering (ASE). IEEE, USA. pp 614–624
Yilmaz C, Cohen MB, Porter A (2014) "Reducing masking effects in combinatorial interaction testing: A feedback driven adaptative approach". IEEE Trans Softw Eng:43–66
Yoo S, Harman M (2012) "Regression testing minimization, selection and prioritization: A survey". Softw Test Verif Reliab 22(2):67–120. https://dl.acm.org/citation.cfm?id=2284813
Yu L, Lei Y, Nourozborazjany M, Kacker RN, Kuhn DR (2013) "An efficient algorithm for constraint handling in combinatorial test generation". In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, Nova York. pp 242–251
Yu L, Lei Y, Kacker RN, Kuhn DR (2013) "ACTS: A combinatorial test generation tool". In: Proceedings on 2013 IEEE Sixth International, Conference on Software Testing, Verification and Validation. IEEE, Nova York. pp 370–375