Preprint: Final version of the article: https://doi.org/10.1007/s10732-020-09462-w

CMSA algorithm for solving the prioritized pairwise test data generation problem in software product lines

Javier Ferrer[1] · **Francisco Chicano**[1] · **José Antonio Ortega-Toro**[2]

Abstract
In Software Product Lines, it may be difficult or even impossible to test all the products of the family because of the large number of valid feature combinations that may exist (Ferrer et al. in: Squillero, Sim (eds) EvoApps 2017, LNCS 10200, Springer, The Netherlands, pp 3–19, 2017). Thus, we want to find a minimal subset of the product family that allows us to test all these possible combinations (pairwise). Furthermore, when testing a single product is a great effort, it is desirable to first test products composed of a set of priority features. This problem is called Prioritized Pairwise Test Data Generation Problem. State-of-the-art algorithms based on Integer Linear Programming for this problem are faster enough for small and medium instances. However, there exists some real instances that are too large to be computed with these algorithms in a reasonable time because of the exponential growth of the number of candidate solutions. Also, these heuristics not always lead us to the best solutions. In this work we propose a new approach based on a hybrid metaheuristic algorithm called *Construct, Merge, Solve & Adapt*. We compare this matheuristic with four algorithms: a Hybrid algorithm based on Integer Linear Programming, a Hybrid algorithm based on Integer Nonlinear Programming, the Parallel Prioritized Genetic Solver, and a greedy algorithm called prioritized-ICPL. The analysis reveals that CMSA is statistically significantly better in terms of quality of solutions in most of the instances and for most levels of weighted coverage, although it requires more execution time.

Keywords Matheuristics · CMSA · Integer programming · Software product lines · Hybrid algorithms · Combinatorial optimization · Feature models

✉ Javier Ferrer
ferrer@lcc.uma.es

Francisco Chicano
chicano@lcc.uma.es

José Antonio Ortega-Toro
josetoro@virustotal.com

1 ITIS Software, Universidad de Málaga, Málaga, Spain

2 VirusTotal, Málaga, Spain

1 Introduction

Software Product Lines (SPLs) are used to achieve a more efficient software development and management of the variability of software products, reducing the costs and time to market, as well as maintenance costs (Pohl et al. 2005). These product lines carry a great variability within products of the same family of products. This variability is due to the mass customization and it implies a great challenge when we face the task of testing because of the combinatorial explosion in the number of products (Engström and Runeson 2011).

Many proposals have arisen having into account these difficulties (Cohen et al. 2008). Some of these are based on *pairwise testing* (Lopez-Herrejon et al. 2013; Oster et al. 2010; Perrouin et al. 2012), where each possible combination of two features must be present in at least one product. Some combinations can be more important than others, introducing a priority among configurations or features. In this case, a weight is assigned to each configuration, which can be derived from weights assigned to products. The optimization problem that we want to solve consists in finding a set of products with the minimum cardinal covering all weighted configurations. Additionally, we want to sort the products in such a way that we first test the products containing higher priority features.

Recent state-of-the-art proposals on pairwise testing include hybrid algorithms, mixing heuristics with exact algorithms (Ferrer et al. 2017). Henard et al. (2014) proposed a similarity measure to build test suites by adding the most dissimilar products. In this way, they designed a very fast search-based algorithm for t-wise test data generation. Also, there are other proposals dealing with multiple objectives (Henard et al. 2015; Xue and Li 2018), however, none of them take into account feature priorities (nor weights) like we do in this work. Other many-objective approaches such as Hierons et al. (2016) define the violation of model constraints as an objective. This approach might lead us to a solution which violates some constraints.

The hypothesis for this work is that a hybrid matheuristic approach can improve the performance on large instances of the problem, particularly, generating in a probabilistic way new sub-instances of the problem that are combined and solved with an exact algorithm in order to find the best solutions to the whole problem.

Our main contribution is the adaptation of a new approach based on a matheuristic named *Construct, Merge, Solve and Adapt* to solve the Prioritized Pairwise Test Data Generation Problem. In order to validate the benefits of our proposal we compare the results with four algorithms: two hybrid algorithms based on integer programming (Ferrer et al. 2017), one with a linear formulation (HILP) and the other one with a nonlinear formulation (HINLP); the Parallel Prioritized Genetic Solver (PPGS) (Lopez-Herrejon et al. 2014); and a greedy algorithm called prioritized-ICPL (Johansen et al. 2012).

The rest of the article is divided into seven sections. In the next section we introduce the background required to understand both the problem and the algorithm that we propose. Section 3 is devoted to the formalization of the Prioritized Pairwise Test Data Generation Problem. In Sect. 4, we explain the design and implementation of the CMSA adaptation to the Prioritized Pairwise Test Data Generation Problem. In

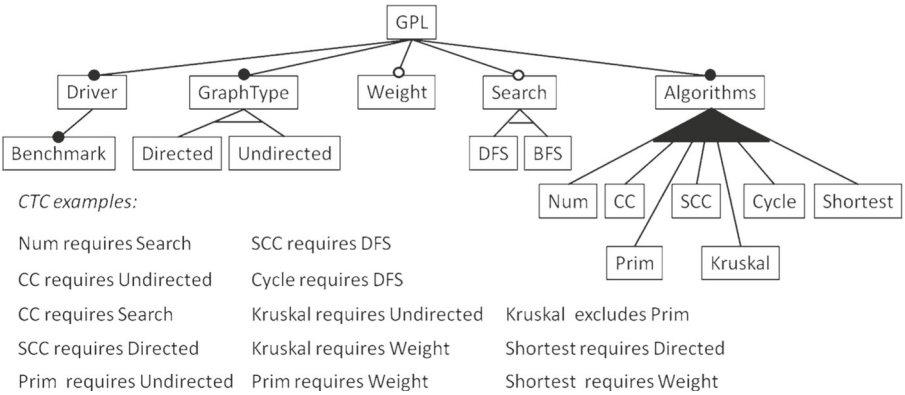

Fig. 1 Graph Product Line feature model

Sect. 5, we briefly explain the algorithms in the comparison and the experimental setup. Results are analyzed in Sect. 6 and we present our conclusions in Sect. 7.

2 Background

2.1 Feature models

Feature models are used in SPLs to define the functionality of a product within a software family as a single combination of features. Models can also represent the constraints that exist between features. A hierarchical tree structure is used to characterize a feature model, where the nodes of the tree are features and the edges represent relationships between these features.

Figure 1 represents the feature model of a classical problem in the evaluation of product-line methodologies, the *Graph Product Line* (GPL) (Lopez-Herrejon and Batory 2001).

There exist four types of relationships between features, differentiated graphically in the feature model:

- *Mandatory* These features are selected when their parent is selected. For example, in the Graph Product Line model, *Driver, Benchmark, GraphType* and *Algorithms* are present in all the products of the family.
- *Optional* They can be or not be selected, like for example, the *Weight* or *Search* features.
- *XOR relations* In these cases only one of the features of the group must be selected when the parent feature is selected. *DFS* and *BFS* features are of this kind in the Graph Product Line model.
- *Inclusive-or* This last kind of relation indicates that at least one of the features of the group must be selected when its parent feature is selected. In the example, when the feature *Algorithms* is selected, at least one of the group {*Num, CC, SCC, Cycle, Shortest, Prim* and *Kruskal*} must be selected.

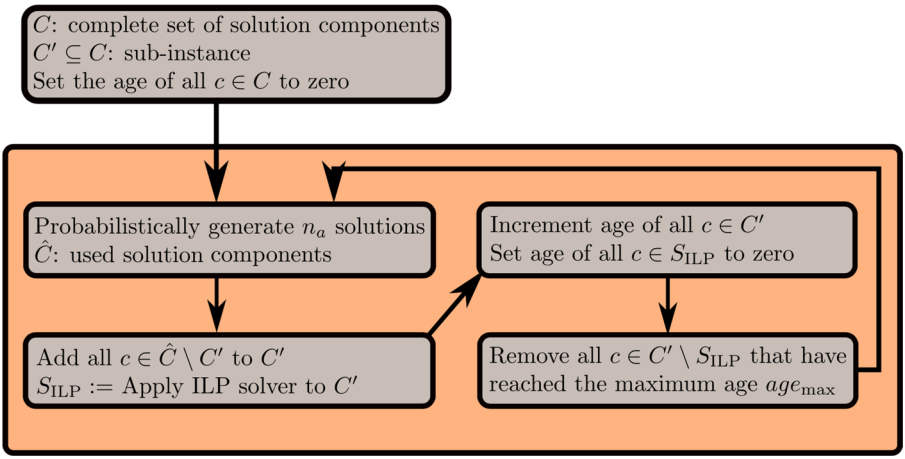

Fig. 2 CMSA flow graph (image by C. Blum)

There are two other types of constraints over the model, called *Cross-Tree Constraints (CTC)*: *requires* and *excludes*. In the Graph Product Line feature model, we can observe these kind of constraints below the tree structure. For example "Num requires Search" implies that when the *Num* feature is selected, the *Search* feature must be selected as well. On the other hand, "Kruskal excludes Prim" implies that when the *Kruskal* feature is selected, the *Prim* feature must not be selected.

It can easily be seen that these constraints can be formalized using propositional logic.

2.2 Construct, merge, solve and adapt

Matheuristics are techniques that combine metaheuristics and mathematical programming techniques. The *Construct, Merge, Solve & Adapt* (CMSA) algorithm is a matheuristic for combinatorial optimization introduced by Blum et al. (2016). Before describing the algorithm we present some concepts.

Given a problem P, let C be the set of all possible components of the solution to the instance of our problem I. C is called the complete set of solution components with respect to I. A valid solution S to I is represented as a subset of the solution components, that is, $S \subseteq C$. Finally, a sub-instance C' of I is a subset of the set of solution components, so $C' \subseteq C$. The idea of the algorithm is the following (flow graph in Fig. 2).

While the time limit established is not reached:

1. First, it generates n_a solutions of the main instance of the problem I.
2. All the components belonging to the generated solutions are merged to form a sub-instance of the problem, C'.
3. An exact algorithm is applied to the sub-instance C', expecting that the solution for C' is (quasi-)optimal for C.

4. The solution is compared with the best solution found at the moment and C' is updated accordingly with a defined aging policy, mostly deleting useless solution components.

The algorithm (see Algorithm 1) has two key components that have to be defined accordingly for the target problem:

– A solution generator: based on some randomized strategy and providing high quality solutions.
– An exact solver for the sub-instances: for example, based on ILP or other exact algorithm.

Algorithm 1: Construct, Merge, Solve & Adapt (CSMA)

```
1  input: set of problem components C, values for parameters n_a and age_max
2  bestSolution := C
3  subInstance := ∅
4  age[c] := 0 for all c ∈ C
5  while CPU time limit not reached do
6      for t = 1, ..., n_a do
7          probSolution := ProbabilisticSolution(C)
8          for all x ∈ probSolution and x ∉ subInstance do
9              age[x] := 0
10             subInstance := subInstance ∪ {x}
11         end
12     end
13     exactSolution := ExactSolver(subInstance)
14     if exactSolution is better than bestSolution then
15         bestSolution := exactSolution
16     end
17     Adapt(subInstance, exactSolution, age_max) //aging and discard mechanisms
18 end
19 output: bestSolution
```

With the aim of clarifying the behaviour of the algorithm, we are going to describe it informally. Once the algorithm has an initial population (several test suites), it merges all the different products which comprises the test suites. Then, an exact solver selects a test suite, from the set of products, with minimum cardinality aiming at total weighted coverage, which is at least as good as the best known. Then, the algorithm execute the Adapt method (Algorithm 1, Line 17). The algorithm increments the age of the products not used in the new solution. When the age of a product is over a threshold, the product is discarded. In Sect. 4, we explain how we adapt the algorithm to prioritized SPL.

3 Problem formalization: prioritized pairwise test data generation

Now we present the terminology related to *Combinatorial Interaction Testing (CIT)*. This approach builds a set of samples that allow to test different system configurations

(Nie and Leung 2011). When we apply this approach to SPL testing, the set of samples is a subset of the products of the family. Next, we introduce the concepts that will lead us to the Prioritized Pairwise Test Data Generation Problem.

Definition 1 (*Feature list*) A *feature list* FL is the list of all the features in a feature model. The feature list of the running example shown in Fig. 1 is the following:

FL = {GPL, Driver, Benchmark, GraphType, Directed, Undirected, Weight, Search, DFS, BFS, Algorithms, Num, CC, Prim, SCC, Kruskal, Cycle, Shortest}.

Definition 2 (*Product*) A *product* is represented by a pair $(S, \bar{S})$, where S is a subset of a feature list FL, $S \subseteq FL$. Thus, $\bar{S} = FL - S$. For example, a product p could be Benchmark selected and the rest of features unselected.

p=({Benchmark},{GPL, Driver, GraphType, Directed, Undirected, Weight, Search, DFS, BFS, Algorithms, Num, CC, Prim, SCC, Kruskal, Cycle, Shortest})

Definition 3 (*Valid product*) We say that a product p is *valid* with respect a feature model fm iff $p.S$ and $p.\bar{S}$ do not violate any constraint described by the feature model. The set of all valid products of a feature model is denoted by P^{fm}. For example, a valid product vp is GPL, Driver, Benchmark, GraphType, Directed, Algorithms, Num, Search, and DFS selected and the rest of features unselected.

vp=({GPL, Driver, Benchmark, GraphType, Directed, Algorithms, Num, Search, DFS},{Undirected, Weight, BFS, CC, Prim, SCC, Kruskal, Cycle, Shortest})

Definition 4 (*Pair*) A *pair* pr is a tuple $(s, \bar{s})$, where s and $\bar{s}$ represent two disjoint feature sets from a feature list FL, which union has two different features. That is, $pr.s \cup pr.\bar{s} \subseteq FL$, $pr.s \cap pr.\bar{s} = \emptyset$ and $|pr.s \cup pr.\bar{s}| = 2$. A pair pr is covered by a product p iff $pr.s \subseteq p.S \ \wedge \ pr.\bar{s} \subseteq p.\bar{S}$. For example, a pair pa is Directed and Undirected both selected.

$pa = (\{Directed, Undirected\}, \emptyset)$

Definition 5 (*Valid Pair*) A pair pr is *valid* within a feature model fm if there exists a product that covers pr. The set of all valid pairs in a feature model fm is denoted with VPR^{fm}. For example, a valid pair vpa is Directed selected and Undirected unselected.

$vpa = (\{Directed\}, \{Undirected\})$

Based on the previous definitions of feature list, product, valid product, pair and valid pair, we define higher levels concepts related to the problem formulation. In the following we define the concept of test suite, prioritized product, configuration, covering array and coverage.

Definition 6 (*Test suite*) A *test suite* ts for a feature model fm is a set of valid products of fm. A test suite ts is *complete* if it covers all the valid pairs in VPR^{fm}, that is, $\forall pr \in VPR^{fm} \rightarrow \exists p \in ts$ such that p covers pr

Definition 7 (*Prioritized product*) A *prioritized product* pp is a tuple (p, w), where p represents a valid product in a feature model fm and $w \in \mathbb{R}$ represents its weight.

Definition 8 (*Configuration*) A *configuration* c is a tuple (pr, w) where pr is a valid pair and $w \in \mathbb{R}$ represent its weight. w is computed as follows. Let PP be the set of all prioritized products and PP_{pr} a subset of PP, such that PP_{pr} contains all the prioritized products of PP that cover pr, that is, $PP_{pr} = \{\ p \in PP \mid p \text{ covers } pr\ \}$. Then $w = \sum_{p \in PP_{pr}} p.w$

Definition 9 (*Covering Array*) A *covering array* CA for a feature model fm and a set of configurations C is a set of valid products P that covers all configurations in C whose weight is greater that zero: $\forall c \in C\ (c.w > 0 \rightarrow \exists p \in CA$ such that p covers $c.pr)$.

Definition 10 (*Coverage*) Given a covering array CA and a set of configurations C, we define $cov(CA)$ as the sum of all configuration weights in C covered by any configuration in CA divided by the sum of all configuration weights in C, that is:

$$cov(CA) = \frac{\sum_{c \in C,\ \exists p \in CA,\ p \text{ covers } c.pr} c.w}{\sum_{c \in C} c.w}. \tag{1}$$

The optimization problem of our interest consists in finding a covering array CA with the minimum number of products $|CA|$ for a given coverage, $cov(CA)$. This problem is defined as single-objective, but it is possible to formulate the problem as bi-objective, where we want to maximize coverage and minimize the number of products. We can even add more objectives to the formulation, like the cost of building a product for testing (we are considering in this work that all the products have the same construction cost). Adding new objectives to the problem does not prevent CMSA of being used. Although CMSA solves single-objective problems, there are high level algorithms that can be applied to solve a multi-objective problem using single-objective solvers. One interesting example in the case of multi-objective Integer Linear Programming is the one proposed by Dächert and Klamroth in Dächert and Klamroth (2015). Thus, neither the use of ILP solvers inside CMSA nor the use of CMSA itself pose a serious limitation to the number of objectives we can add to our current formulation.

4 Applying CMSA to prioritized SPL

In order to apply the CMSA algorithm to our problem we have to define the three methods described in Algorithm 1: *ProbabilisticSolution*, *ExactSolver* and *Adapt* methods. Before that, we have to guess what is a solution component for the prioritized pairwise test data generation in the CMSA algorithm. In this case, given that we want to minimize the cardinality of the set of products that entirely covers all the possible configurations in the feature model, a solution component is a valid product from the feature model FM, being P the set of all valid products of FM.

Coming up next we introduce the idea of the three components of the CMSA adaptation to our problem. First, we explain the *ProbabilisticSolution* method, then we present the *ExactSolver* method. Finally, we describe the aging policy.

4.1 Solution generation

At the start of the main loop of the algorithm, several solutions for P are generated to be merged in a final sub-instance that is solved later. We consider the whole search space P, generating random products that are valid within the feature model. The generation of random valid products is performed by means of an ILP solver which is very fast, being faster and faster as the set of uncovered pairs becomes smaller, because it considers less weighted pairs in the objective function. We have set a stopping condition of 3 seconds for the ILP solver, just to be sure it returns a solution. This process is also used for the initial population. The pseudo-code of this method is described below in Algorithm 2.

Algorithm 2: ProbabilisticSolution

```
1  input: problem instance I
2  solution := ∅
3  uncovered := validPairs(I)
4  while uncovered ≠ ∅ do
5      x := generateRandomValidProduct
6      if configurations(x) ∩ uncovered ≠ ∅ then
7          solution := solution ∪ {x}
8          uncovered := uncovered − configurations(x)
9      end
10 end
11 output: solution
```

The fact that there is no uncovered configuration in each solution generated ensures that the solution will always cover all the weighted configurations, and this property also holds in the sub-instance generated after merging all the solutions. This strategy guarantees that, eventually, the algorithm will find a solution better than the previous one.

4.2 Exact solver

We use an exact algorithm to compute the best solution for the sub-instance generated. We use an ILP solver to select a subset of products for the sub-instance which, reaching a given weighted coverage, has minimum cardinality. This problem is equivalent to the hitting set problem or the test suite minimization problem, in its mono-objective version (Arito et al. 2012) and, thus, we can use the same integer linear program to solve the problem.

4.3 Adapting the sub-instance

The aging mechanism used here is the same proposed by Blum et al. (2016). In each iteration of the algorithm, the sub-instance is composed by a subset $C' \subseteq C$ of the components of the problem. The *ExactSolver* method returns a solution, which is a

subset $S \subseteq C'$ of components. Then, the "age" of all the components that are part of the solution S are reset to 0, while the rest of the components see their age increased by 1. If any component reaches the maximum age established (age_{max}), then the component is removed from C'. Algorithm 3 shows the pseudo-code for the *Adapt* method.

Algorithm 3: Adapt

```
1  input: sub-instance C', sub-instance solution S
2  for x in S do
3  |  age[x] := 0
4  end
5  for x in C' − S do
6  |  age[x] := age[x] + 1
7  |  if age[x] equals age_max then
8  |  |  C' := C' − {x}
9  |  end
10 end
11 output: C'
```

5 Experimental setup

In this section we describe how the analysis of the approach is performed. First, we introduce the other four algorithms we compare with CMSA. Then, we describe the benchmark used for the evaluation. Finally, we explain the different experiment configurations.

5.1 Hybrid algorithms based on integer programming

Two different hybrid algorithms combining a greedy heuristic and integer programming were introduced by Ferrer et al. (2017). The first one, called HILP, is based on an integer linear formulation, and the second, named HINLP, is based on a quadratic (nonlinear) integer formulation. The two algorithms proposed in this work use the same high level greedy strategy. In each iteration they try to find a product that maximizes the weighted coverage. They select in each iteration the product that contributes with greater coverage to the actual solution. The algorithm applies the heuristic to the whole product set (that can be of billions of possible products) instead of small subsets. For further details on HILP or HINLP, please refer to Ferrer et al. (2017).

5.2 Prioritized pairwise genetic solver

Prioritized Pairwise Genetic Solver (PPGS) is a constructive genetic algorithm that follows a master-slave model to parallelize the individuals' evaluation. In each iteration, the algorithm adds the best product to the test suite until all weighted pairs are

covered. The best product to be added is the product that adds more weighted coverage (only pairs not covered yet) to the set of products.

The parameter setting used by PPGS is the same of the reference paper for the algorithm (Lopez-Herrejon et al. 2014). It uses binary tournament selection and a one-point crossover with a probability 0.8. The population size of 10 individuals favours the exploitation rather than the exploration during search. The termination condition is to reach 1000 fitness evaluations. The mutation operator iterates over all selected features of an individual and randomly replaces a feature by another one with a probability 0.1. The algorithm stops when all the weighted pairs have been covered. For further details on PPGS see Lopez-Herrejon et al. (2014).

5.3 Prioritized-ICPL algorithm

Prioritized-ICPL (pICPL) is a greedy algorithm to generate t-wise covering arrays proposed by Johansen et al. (2012). pICPL does not compute covering arrays with full coverage but rather covers only those t-wise combinations among features that are present in at least one of the prioritized products, as was described in the formalization of the problem in Sect. 3. We must highlight here that the pICPL algorithm uses *data parallel execution*, supporting any number of processors. Their parallelism comes from simultaneous operations across large sets of data. For further details on prioritized-ICPL please refer to Johansen et al. (2012).

5.4 Benchmark

The feature models that we use for the comparison of the algorithms are generated from 16 real SPL systems. We considered a method called *measured values* to assign weight values to prioritized products. This method consists in assigning the weights derived from non-functional property values obtained from 16 real SPL systems, that were measured with the SPL Conqueror approach introduced by Siegmund et al. (2013). This approach aims at providing reliable estimates of measurable non-functional properties such as performance, main memory consumption, and footprint. These estimations are then used to emulate more realistic scenarios where software testers need to schedule their testing effort giving priority, for instance, to products or feature combinations that exhibit higher footprint or performance. In this work, we use the actual values taken on the measured products considering pairwise feature interactions. Table 1 summarizes the SPL systems evaluated, their feature number (`FN`), products number (`PN`), configurations number measured (`CN`), and the percentage of prioritized products (`PP%`) used in our comparison.

In the case of the feature models where the percentage of the prioritized products is equal to 100%, applying the heuristic to the whole solution components set P (without generating sub-instances) should give similar results than applying HINLP. A greedy implementation of CMSA without generating sub-instances have been implemented in order to test the rest of the functionalities of the algorithm and validate the consistency of the results.

Table 1 Benchmark of feature models

Model name	FN	PN	CN	PP%
Apache	10	256	192	75.0
BerkeleyDBFootprint	9	256	256	100.0
BerkeleyDBMemory	19	3840	1280	33.3
BerkeleyDBPerformance	27	1440	180	12.50
Curl	14	1024	68	6.6
LinkedList	26	1440	204	14.1
Linux	25	$\approx 3E7$	100	≈ 0.0
LLVM	12	1024	53	5.1
PKJab	12	72	72	100.0
Prevayler	6	32	24	75.0
SensorNetwork	27	16704	3240	19.4
SQLiteMemory	40	$\approx 5E7$	418	≈ 0.0
Violet	101	$\approx 1E20$	101	≈ 0.0
Wget	17	8192	94	1.15
x264	17	2048	77	3.7
ZipMe	8	64	64	100.0

5.5 Experiments configuration

The experiments were run on a cluster of 16 machines with Intel Core2 Quad processors Q9400 (4 cores per processor) at 2.66 GHz and 4 GB memory and 2 nodes (96 cores) equipped with two Intel Xeon CPU (E5-2670 v3) at 2.30 GHz and 64 GB memory. The cluster was managed by HTCondor 8.2.7, which allowed us to perform parallel independent executions to reduce the overall experimentation time.

For the CMSA algorithm, different parameter configurations of time limit (max_{time}), solutions generated per iteration (n_a) and maximum age for the aging policy (age_{max}) were used. As a result, we chose the best configuration with n_a=5 and $age_{max} = 4$. For each configuration, a total of 30 independent runs were executed.

We have to keep in mind that the max_{time} =3 seconds indicates the limit of time for the last iteration, in cases of some feature models (i.e. Violet) where the number of constraints is high, the algorithm can take more time. We applied the Kolmogorov–Smirnov normality test to confirm that the distribution of the results is not normal. Therefore, we applied the non-parametric Kruskal–Wallis test with a confidence level of 95% (p-value under 0.05) with Bonferroni's p value correction to check if the observed differences are statistically significant. In the cases where Kruskal-Wallis test rejects the null hypothesis, we run a single factor ANOVA post hoc test for pairwise comparisons.

In order to properly interpret the results of statistical tests, it is always advisable to report effect size measures. For that purpose, we have also used the non-parametric effect size measure $\widehat{A}_{12}$ statistic proposed by Vargha and Delaney (2000). It tells us how often, on average, one technique outperforms the other. It could be used to determine the probability of yielding higher performance by different algorithms.

Table 2 Mean and standard deviation of number of products and time in 30 independent runs of the whole benchmark of feature models

Coverage	CMSA	HILP	HINLP	PPGS	pICPL
50%	$\mathbf{1.56_{0.50}}$	$\mathbf{1.56_{0.50}}$	$\mathbf{1.56_{0.50}}$	$1.58_{0.49}$	$\mathbf{1.56_{0.50}}$
75%	$\mathbf{2.53_{0.71}}$	$2.63_{0.78}$	$2.63_{0.78}$	$2.66_{0.77}$	$2.75_{0.75}$
80%	$\mathbf{2.75_{0.75}}$	$2.81_{0.81}$	$2.81_{0.81}$	$2.81_{0.73}$	$3.25_{0.97}$
85%	$\mathbf{3.31_{0.87}}$	$3.44_{0.86}$	$3.44_{0.86}$	$3.46_{0.87}$	$3.81_{0.95}$
90%	$\mathbf{3.79_{0.77}}$	$4.06_{1.03}$	$4.00_{0.94}$	$4.12_{1.04}$	$4.56_{1.27}$
95%	$\mathbf{4.94_{0.90}}$	$5.37_{1.05}$	$5.38_{1.05}$	$5.45_{1.14}$	$6.06_{1.44}$
96%	$\mathbf{5.17_{1.05}}$	$5.69_{1.16}$	$5.69_{1.16}$	$5.86_{1.18}$	$6.38_{1.58}$
97%	$\mathbf{5.67_{1.09}}$	$6.13_{1.22}$	$6.13_{1.22}$	$6.24_{1.38}$	$6.75_{1.39}$
98%	$\mathbf{5.96_{1.26}}$	$6.81_{1.42}$	$6.75_{1.39}$	$6.98_{1.55}$	$7.44_{1.66}$
99%	$\mathbf{6.79_{1.52}}$	$7.75_{1.64}$	$7.75_{1.64}$	$7.92_{1.87}$	$8.75_{2.08}$
100%	$\mathbf{10.06_{4.99}}$	$11.69_{5.51}$	$11.63_{5.33}$	$12.08_{6.50}$	$12.19_{5.68}$
Time (s)	164_{346}	18_{10}	$\mathbf{1_{2}}$	2049_{7684}	24_{59}

The best result for each percentage of prioritized coverage and total time are highlighted in bold

Given a performance measure M, $\hat{A}_{12}$ measures the probability that running algorithm A yields higher M values than running another algorithm B. If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. If $\hat{A}_{12} = 0.3$ then one would obtain higher values for M with algorithm A, 30% of the time.

6 Analysis of the results

In this section we analyze the results of the execution of CMSA in comparison with the results of state-of-the-art algorithms (HILP, HINLP, PPGS and pICPL). Table 2 summarizes the results of the execution of the algorithms for different values of weighted coverage. Each column corresponds to one algorithm and in the rows we show the number of products required to reach 50% up to 100% of weighted coverage. The data shown in each cell is the mean and the standard deviation of the number of products required to reach the coverage of the SPL for all the independent runs of the whole benchmark of feature models. We also show the average and standard deviation of the required runtime in the last line. We highlight the best value for each percentage of weighted coverage and the shortest execution time.

We observe that CMSA is the best in solution quality (less products required to obtain a particular level of coverage) for all percentages of weighted coverage. After CMSA, the algorithms based on integer programming obtain the best solutions. The difference in quality between HILP and HINLP are almost insignificant, except for 100% coverage, so it is difficult to claim that one algorithm is better than the other. Then, PPGS is the fourth algorithm in our comparison, and finally pICPL is the worst.

Regarding the execution time, we can appreciate that HINLP is clearly the fastest algorithm, actually thanks to the nonlinear formulation it produces a boost in computation time due to the reduction of clauses in comparison with the linear variant of the

Table 3 Times an algorithm has the best average number of products per percentage of coverage

Coverage	CMSA	HILP	HINLP	PPGS	pICPL
50%	16	16	16	14	16
75%	16	13	13	10	11
80%	16	15	15	14	9
85%	16	14	14	11	10
90%	16	12	12	8	7
95%	15	8	8	4	3
96%	16	6	6	3	3
97%	16	8	8	4	3
98%	15	2	2	0	1
99%	16	2	2	1	2
100%	15	2	2	1	3
Total	173	98	98	70	68

algorithm. It is closely followed by HILP and pICPL (also based on a greedy strategy). They are followed by CMSA in the speed ranking and, finally, quite far from the rest, PPGS, a genetic algorithm.

In order to check whether the differences between the algorithms are statistically significant or just a matter of chance, we have applied the statistical tests explained in the previous section. For 50% coverage there is no significant differences between CMSA and the others. Next, for 75% up to 85% of weighted coverage, there are significant differences between CMSA and pICPL. Finally, for 90% up to 100% CMSA is statistically significantly better than the other algorithms. Regarding the execution time, HINLP is statistically better than the other algorithms.

In Table 3 we show the number of times an algorithm obtains the minimum mean value for each percentage of coverage and for the 16 realistic feature models. Note that Table 3 summarizes the results showed in Table 6 in the "Appendix". On the one hand, we observe that in only 3 out of 176 comparisons (16 feature models and 11 percentages of weighted coverage) other algorithm different than CMSA has obtained a better value for solution quality. On the other hand, in 173 out of 176 comparisons CMSA obtains the best results of the comparison, moreover in 70 out of 176 CMSA is the only obtaining the best value, i.e., no other algorithm obtains such a low value. In general, it can also be seen in this table that, the larger the value of weighted coverage, the better the results of CMSA. The reason behind this behavior is that in early stages of the search, for small and medium values of coverage, it is easier to find the products which add not-covered-yet pairs. When the search progresses, it is harder to find the products which are able to add not-covered-yet pairs.

Let us now focus on how the algorithms obtain total weighted coverage in each feature model. In Table 4 we show the mean value for 100% weighted coverage. We also show the standard deviation, which is 0 in many cases. In most feature models (15 out of 16) CMSA obtains the best value. The exception is in the Linux feature model, where pICPL is the best. In three models (Curl, Prevayler and Violet) CMSA obtains the best value, although at least other algorithm is able to reach the same value. In the

Table 4 Mean values over 30 independent runs for CMSA, HILP, HINLP, PPGS and pICPL

Feature model	CMSA	HILP	HINLP	PPGS	pICPL
Apache	**$6.00_{0.00}$**	7.00	7.00	7.00	8.00
BerkeleyDBFootprint	**$6.07_{0.25}$**	8.00	8.00	$8.17_{0.38}$	9.00
BerkeleyDBMemory	**$20.03_{0.18}$**	21.00	21.00	$23.33_{1.06}$	21.00
BerkeleyDBPerformance	**$9.00_{0.00}$**	10.00	10.00	$10.60_{0.50}$	12.00
Curl	**$8.00_{0.00}$**	9.00	9.00	$9.63_{0.67}$	**8.00**
LinkedList	**$11.10_{0.30}$**	13.00	13.00	$13.37_{0.49}$	14.00
Linux	$11.00_{0.00}$	11.00	11.00	$11.10_{0.66}$	**10.00**
LLVM	**$6.90_{0.30}$**	10.00	10.00	$8.17_{0.46}$	8.00
PKJab	**$6.00_{0.00}$**	7.00	7.00	7.00	8.00
Prevayler	**$6.00_{0.00}$**	**6.00**	**6.00**	**6.00**	**6.00**
SensorNetwork	**$10.00_{0.00}$**	14.00	14.00	$13.97_{1.16}$	17.00
SQLiteMemory	**$23.87_{1.41}$**	28.00	27.00	$31.53_{1.99}$	28.00
Violet	**$12.00_{0.00}$**	**12.00**	**12.00**	$12.83_{0.59}$	15.00
Wget	**$9.87_{0.35}$**	12.00	12.00	$11.37_{1.00}$	11.00
x264	**$9.07_{0.25}$**	12.00	12.00	$12.10_{1.03}$	13.00
ZipMe	**$6.00_{0.00}$**	7.00	7.00	$7.03_{0.18}$	7.00
Total	**$10.06_{4.99}$**	$11.69_{5.51}$	$11.63_{5.33}$	$12.08_{6.50}$	$12.19_{5.68}$

The best value for each FM is highlighted in bold

rest of the feature models analyzed here (12 out of 16) no other algorithm reach the same solution quality than CMSA. As we expected, there are significant differences between CMSA and the other proposals in the pairwise comparisons. Specifically, in 52 out of 64 comparisons (16 feature models and 4 algorithms) CMSA is significantly better than the other algorithms, in 11 out of 64 there is no difference between CMSA and other algorithm, and only once it is significantly worse, particularly in the Linux model with pICPL.

In the comparison between CMSA and HINLP (the second best algorithm), we can appreciate that HINLP is doubtless faster. However, in solution quality it is only able to find the best known solution in two feature models. In addition, there are significant differences between CMSA and HINLP in 14 out of 16 models, so we can claim that CMSA is definitely the best algorithm in the comparison in regards to solution quality.

In order to illustrate the comparison regarding the solution quality for total weighted coverage, in Fig. 3 we show a boxplot for each algorithm considering all the feature models. Note that several outliers for all algorithms are outside the worst range shown in the plot. The boxplot confirms again that CMSA is the best for 100% weighted coverage with a median value of 9 products, followed by HILP and HINLP with a median of 10.5 products, and PPGS with a median of 11 products. In addition, we can also appreciate that CMSA has a lower interquartile range, and all the quartile marks are lower in comparison to the other algorithms.

In light of the obtained results and with the intention of determining whether the results are of practical significance or not, we analyze the $\hat{A}_{12}$ statistic. In Table 5

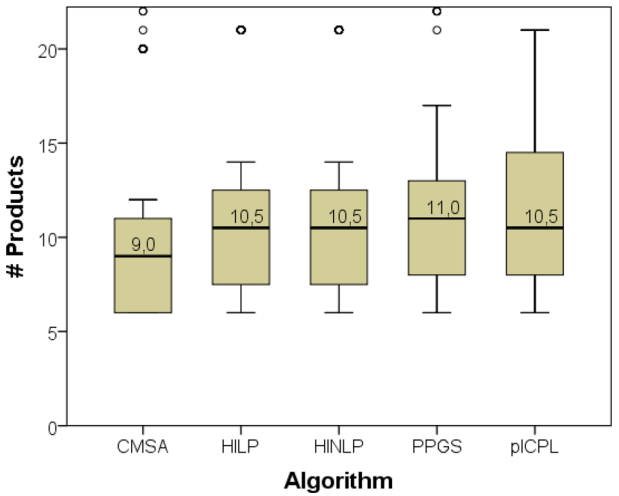

Fig. 3 Number of products needed to achieve total coverage. For each feature model (16), there are 30 solutions obtained by independent replications

Table 5 Vargha and Delaney's statistical test results ($\widehat{A}_{12}$) for total coverage

	CMSA	PPGS	HINLP	HILP	pICPL
CMSA	0.5000	0.1732	0.2010	0.1803	0.0969
PPGS	0.8268	0.5000	0.5021	0.5198	0.3333
HINLP	0.7990	0.4979	0.5000	0.4719	0.3135
HILP	0.8197	0.4802	0.5281	0.5000	0.3729
pICPL	0.9031	0.6667	0.6865	0.6271	0.5000

Each cell is the average value from the $\widehat{A}_{12}$ statistic for the 16 models analyzed. A represents algorithms in rows and B represents algorithms in columns

we summarize the $\hat{A}_{12}$ statistic values for all models and algorithms. The differences between CMSA and the rest of algorithms are quite large. Specifically, CMSA beats PPGS in 82.68%, HINLP in 79.90%, HILP in 81.97% and pICPL in 90.31% of the runs. Therefore, CMSA is able of generating test suites with maximum levels of coverage, and obtain better results than the other algorithms with a high probability.

7 Conclusions

In this work we have applied a novel matheuristic approach (CMSA) to the Prioritized Pairwise Test Data Generation Problem, aiming to ease the task of testing on large SPLs. Our main contribution is the adaptation of the CMSA algorithm to this problem for SPL, relating the CMSA algorithm to the specific nuances of the problem.

We present the empirical results derived from the evaluation of our CMSA approach on the introduced benchmark of feature models. We compare CMSA with four different approaches to tackle the problem of prioritized pairwise test data generation for SPL.

Regarding the solution quality, our analysis showed an improvement in terms of the quality metric, which is better (lower in terms of the number of products) in almost all instances of the benchmark and for all percentages of weighted coverage. In addition, in most comparisons the test suites computed by CMSA are statistically significantly better than those computed by the other algorithms.

Testing on a SPL means a high cost in resources and time due to the effort devoted to the testing phase of even one single product, which can require several hours. Therefore, it is straightforward to think that the best approach is the one that reduces the size of the test suite, in this case our proposal: CMSA. In addition, the execution of the algorithm only requires a few minutes, much less than testing one single product in most of the scenarios. A general conclusion is that CMSA is clearly the best approach for computing prioritized pairwise test data. On the other hand, the approach based on nonlinear integer programming (HINLP) is able to obtain good quality solutions in only a few seconds, then it is the best option when a good solution is immediately needed. This could be the case when testing a single product of the SPL only requires a few seconds.

There remains one Achilles' heel clearly identified in our proposal, the execution time is higher than several approaches studied here. Future work will require a deeper analysis of the performance and quality of the generation of random products, that is the baseline of the construct phase of CMSA. We plan to assess whether CPLEX, the optimizer used to solve the integer programming problems found in this work, is able to provide enough different products to obtain the diversity that every search algorithm requires.

Acknowledgements This research has been partially funded by the Spanish Ministry of Economy and Competitiveness (MINECO) and the European Regional Development Fund (FEDER) under contract TIN2017-88213-R (6city project), the University of Málaga, Consejeria de Economia y Conocimiento de la Junta de Andalucia and FEDER under contract UMA18-FEDERJA-003 (PRECOG project), the Ministry of Science, Innovation and Universities and FEDER under contract RTC-2017-6714-5 (ECOIoT project), the H2020 European Project Tailor (H2020-ICT-2019-3), the Spanish SBSE Research Network (RED2018-102472-T), and the University of Málaga under contract PPIT-UMA-B1-2017/07 (EXHAURO Project). J. Ferrer thanks University of Mállaga for his postdoc fellowship.

Appendix

In this appendix we present the detailed results in Table 6. This table shows the mean number of products required to reach all percentages considered of weighted coverage for all the feature models and all the algorithms.

Table 6 Mean number of products needed to achieve all percentages of weighted coverage for all feature models

Feature models	Algor.	50%	75%	80%	85%	90%	95%	96%	97%	98%	99%	100%	Time (ms)
Apache	CMSA	2.00	3.00	3.00	4.00	4.00	**5.00**	**5.00**	6.00	**6.00**	**6.00**	**6.00**	9522.80
	HILP	2.00	3.00	3.00	4.00	4.00	6.00	6.00	6.00	7.00	7.00	7.00	9695.00
	HINLP	2.00	3.00	3.00	4.00	4.00	6.00	6.00	6.00	7.00	7.00	7.00	449.00
	pICPL	2.00	3.00	3.00	4.00	5.00	6.00	7.00	7.00	7.00	8.00	8.00	741.60
	PPGS	2.00	3.00	3.00	4.00	4.00	6.00	6.00	6.00	7.00	7.00	7.00	10394.03
BDBFootprint	CMSA	2.00	3.00	4.00	**4.00**	**4.00**	**6.00**	**6.00**	**6.00**	**6.00**	**6.07**	**6.07**	9537.20
	HILP	2.00	4.00	4.00	5.00	6.00	7.00	7.00	7.00	8.00	8.00	8.00	19015.00
	HINLP	2.00	4.00	4.00	5.00	5.00	7.00	7.00	7.00	7.00	8.00	8.00	440.00
	pICPL	2.00	4.00	5.00	6.00	7.00	8.00	8.00	8.00	8.00	9.00	9.00	687.07
	PPGS	2.00	4.00	4.00	5.00	5.97	6.97	6.97	6.97	7.97	8.00	8.17	11213.53
BDBMemory	CMSA	2.00	3.00	3.00	4.00	4.00	**6.00**	**6.73**	**7.27**	**8.20**	**10.00**	**20.03**	148818.63
	HILP	2.00	3.00	3.00	4.00	4.00	6.00	7.00	8.00	9.00	11.00	21.00	21404.00
	HINLP	2.00	3.00	3.00	4.00	4.00	6.00	7.00	8.00	9.00	11.00	21.00	1010.00
	pICPL	2.00	3.00	3.00	4.00	6.00	7.00	8.00	8.00	10.00	11.00	21.00	9911.80
	PPGS	2.00	3.00	3.00	4.00	4.73	6.87	7.80	8.77	9.97	11.90	23.33	117607.53
BDBPerformance	CMSA	1.00	2.00	2.00	3.00	3.00	4.00	4.00	5.00	**5.00**	**6.00**	**9.00**	56589.83
	HILP	1.00	2.00	2.00	3.00	3.00	4.00	4.00	5.00	6.00	7.00	10.00	14993.00
	HINLP	1.00	2.00	2.00	3.00	3.00	4.00	4.00	5.00	6.00	7.00	10.00	577.00
	pICPL	1.00	2.00	3.00	3.00	4.00	6.00	6.00	6.00	6.00	7.00	12.00	6008.37
	PPGS	1.00	2.00	2.00	3.00	3.00	4.00	4.83	5.00	5.93	7.00	10.60	47361.73
Curl	CMSA	2.00	2.53	3.00	**3.00**	4.00	**5.00**	**5.00**	6.00	**6.00**	**6.90**	8.00	22736.93
	HILP	2.00	3.00	3.00	4.00	4.00	6.00	6.00	6.00	7.00	8.00	9.00	13639.00
	HINLP	2.00	3.00	3.00	4.00	4.00	6.00	6.00	6.00	7.00	8.00	9.00	451.00
	pICPL	2.00	3.00	3.00	4.00	4.00	6.00	6.00	6.00	7.00	7.00	8.00	648.27

Table 6 continued

Feature models	Algor.	50%	75%	80%	85%	90%	95%	96%	97%	98%	99%	100%	Time (ms)
	PPGS	2.00	3.00	3.00	3.97	4.03	5.83	6.00	6.50	7.37	8.07	9.63	17454.57
LinkedList	CMSA	1.00	2.00	2.00	2.00	3.00	4.00	4.00	5.00	**5.00**	**7.00**	**11.10**	69734.97
	HILP	1.00	2.00	2.00	2.00	3.00	4.00	5.00	5.00	6.00	8.00	13.00	12894.00
	HINLP	1.00	2.00	2.00	2.00	3.00	4.00	5.00	5.00	6.00	8.00	13.00	625.00
	pICPL	1.00	2.00	2.00	3.00	3.00	4.00	4.00	5.00	7.00	11.00	14.00	6865.53
	PPGS	1.00	2.00	2.00	2.00	3.00	4.23	5.00	5.00	6.13	7.73	13.37	60684.57
Linux	CMSA	2.00	4.00	4.00	5.00	**5.70**	7.00	**7.00**	**7.90**	8.03	9.00	11.00	137948.13
	HILP	2.00	4.00	4.00	5.00	6.00	7.00	8.00	8.00	9.00	10.00	11.00	29396.00
	HINLP	2.00	4.00	4.00	5.00	6.00	7.00	8.00	8.00	9.00	10.00	11.00	6813.00
	pICPL	2.00	4.00	5.00	5.00	6.00	8.00	8.00	8.00	**8.00**	9.00	**10.00**	3539.37
	PPGS	2.00	4.00	4.00	5.00	6.00	7.00	7.67	8.00	8.37	9.40	11.10	49385.43
LLVM	CMSA	2.00	3.00	**3.00**	4.00	**4.00**	**5.00**	6.00	**6.00**	**6.00**	**6.00**	**6.90**	15747.90
	HILP	2.00	3.00	4.00	4.00	5.00	6.00	6.00	7.00	7.00	8.00	10.00	19164.00
	HINLP	2.00	3.00	4.00	4.00	5.00	6.00	6.00	7.00	7.00	8.00	10.00	472.00
	pICPL	2.00	3.00	4.00	4.00	5.00	6.00	7.00	7.00	7.00	8.00	8.00	526.50
	PPGS	2.00	3.00	3.03	4.00	5.00	6.00	6.00	6.07	7.00	8.00	8.17	12805.90
PKJab	CMSA	1.00	2.00	2.00	3.00	3.00	**4.00**	**4.00**	5.00	5.00	**5.00**	**6.00**	8524.03
	HILP	1.00	2.00	2.00	3.00	3.00	5.00	5.00	5.00	5.00	6.00	7.00	6137.00
	HINLP	1.00	2.00	2.00	3.00	3.00	5.00	5.00	5.00	5.00	6.00	7.00	368.00
	pICPL	1.00	2.00	3.00	3.00	3.00	5.00	5.00	6.00	7.00	8.00	8.00	501.13
	PPGS	1.00	2.00	2.00	3.00	3.07	4.53	5.00	5.00	5.17	6.00	7.00	11439.47
Prevayler	CMSA	2.00	3.00	3.00	3.00	4.00	5.00	5.00	5.00	**5.00**	6.00	6.00	4755.83
	HILP	2.00	3.00	3.00	3.00	4.00	5.00	5.00	6.00	6.00	6.00	6.00	8086.00
	HINLP	2.00	3.00	3.00	3.00	4.00	5.00	5.00	6.00	6.00	6.00	6.00	325.00

Table 6 continued

Feature models	Algor.	50%	75%	80%	85%	90%	95%	96%	97%	98%	99%	100%	Time (ms)
	pICPL	2.00	3.00	3.00	3.00	4.00	5.00	5.00	5.00	6.00	6.00	6.00	238.37
	PPGS	2.00	3.00	3.00	3.00	4.00	5.00	5.00	5.60	6.00	6.00	6.00	8091.17
SensorNetwork	CMSA	1.00	**2.97**	3.00	3.00	4.00	5.00	5.00	**5.20**	**6.00**	**6.93**	**10.00**	155619.87
	HILP	1.00	3.00	3.00	3.00	4.00	5.00	5.00	6.00	7.00	8.00	14.00	21154.00
	HINLP	1.00	3.00	3.00	3.00	4.00	5.00	5.00	6.00	7.00	8.00	14.00	951.00
	pICPL	1.00	3.00	4.00	5.00	6.00	8.00	9.00	9.00	10.00	11.00	17.00	74181.93
	PPGS	1.00	3.00	3.00	3.00	4.00	5.03	5.47	6.00	6.97	7.87	13.97	71971.50
SQLiteMemory	CMSA	1.00	2.00	2.00	3.00	4.00	**5.00**	6.00	**6.37**	**7.57**	**9.30**	**23.87**	1416002.00
	HILP	1.00	2.00	2.00	3.00	4.00	6.00	6.00	7.00	8.00	10.00	28.00	47816.00
	HINLP	1.00	2.00	2.00	3.00	4.00	6.00	6.00	7.00	8.00	10.00	27.00	6450.00
	pICPL	1.00	3.00	4.00	4.00	5.00	8.00	8.00	9.00	11.00	14.00	28.00	41312.30
	PPGS	1.03	2.17	2.90	3.23	4.07	6.03	6.97	7.93	9.23	11.70	31.53	903118.97
Violet	CMSA	1.00	1.00	1.00	2.00	2.00	3.00	3.00	3.00	3.00	4.00	12.00	483297.43
	HILP	1.00	1.00	1.00	2.00	2.00	3.00	3.00	3.00	3.00	4.00	12.00	31558.00
	HINLP	1.00	1.00	1.00	2.00	2.00	3.00	3.00	3.00	3.00	4.00	12.00	2878.00
	pICPL	1.00	1.00	1.00	2.00	2.00	3.00	3.00	4.00	4.00	6.00	15.00	241170.97
	PPGS	1.00	1.00	1.00	2.00	2.00	**2.93**	3.00	3.07	3.30	4.53	12.83	31376054.20
Wget	CMSA	2.00	2.00	3.00	3.00	4.00	5.00	**5.97**	6.00	**6.63**	**7.40**	**9.87**	46015.90
	HILP	2.00	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	8.00	12.00	16141.00
	HINLP	2.00	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	8.00	12.00	478.00
	pICPL	2.00	3.00	3.00	4.00	4.00	6.00	6.00	7.00	7.00	9.00	11.00	1337.43
	PPGS	2.00	2.13	3.00	3.07	4.00	5.43	6.00	6.40	7.00	8.03	11.37	31525.37
x264	CMSA	1.00	2.00	3.00	3.00	4.00	5.00	**5.00**	6.00	**6.00**	**7.00**	**9.07**	36472.90

Table 6 continued

Feature models	Algor.	50%	75%	80%	85%	90%	95%	96%	97%	98%	99%	100%	Time (ms)
	HILP	1.00	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	8.00	12.00	8547.00
	HINLP	1.00	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	8.00	12.00	479.00
	pICPL	1.00	2.00	3.00	3.00	4.00	5.00	6.00	7.00	7.00	9.00	13.00	1224.70
	PPGS	1.23	2.23	3.00	3.07	4.00	5.30	6.00	6.50	7.23	8.47	12.10	37368.53
ZipMe	CMSA	2.00	3.00	3.00	4.00	**4.00**	**5.00**	**5.00**	**5.00**	**6.00**	**6.00**	**6.00**	7097.73
	HILP	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	7.00	7.00	7.00	12562.00
	HINLP	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	7.00	7.00	7.00	355.00
	pICPL	2.00	3.00	3.00	4.00	5.00	6.00	6.00	6.00	7.00	7.00	7.00	384.50
	PPGS	2.00	3.00	3.00	4.00	5.00	6.00	6.00	7.00	7.00	7.00	7.03	13035.17

A value is highlighted in bold when it is strictly smaller than the others for that particular level of coverage and feature model

References

Arito, F., Chicano, F., Alba, E.: On the Application of Sat Solvers to the Test Suite Minimization Problem. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7515 LNCS, pp. 45–59 (2012)

Blum, C., Pinacho, P., López-Ibáñez, M., Lozano, J.A.: Construct, merge, solve & adapt a new general algorithm for combinatorial optimization. Comput. Oper. Res. **68**(C), 75–88 (2016)

Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: a Greedy approach. IEEE Trans. Softw. Eng. **34**(5), 633–650 (2008)

Dächert, K., Klamroth, K.: A linear bound on the number of scalarizations needed to solve discrete tricriteria optimization problems. J. Glob. Optim. **61**(4), 643–676 (2015)

Engström, E., Runeson, P.: Software product line testing a systematic mapping study. Inf. Softw. Technol. **53**(1), 2–13 (2011)

Ferrer, J., Chicano, F., Alba, E.: Hybrid algorithms based on integer programming for the search of prioritized test data in software product lines. In: Squillero, G., Sim, K. (eds.) EvoApps 2017, LNCS 10200, pp. 3–19. Springer, The Netherlands (2017)

Henard, C., Papadakis, M., Harman, M., Le Traon, Y.: Combining multi-objective search and constraint solving for configuring large software product lines. In: Proceedings of the 37th International Conference on Software Engineering, vol. 1, pp. 517–528. ICSE '15, IEEE Press, Piscataway (2015)

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the combinatorial explosion: using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans. Softw. Eng. **40**(7), 650–670 (2014)

Hierons, R.M., Li, M., Liu, X., Segura, S., Zheng, W.: SIP: optimal product selection from feature models using many-objective evolutionary optimization. ACM Trans. Softw. Eng. Methodol. **25**(2), 17:1–17:39 (2016)

Johansen, M.F., Haugen, Ø., Fleurey, F., Eldegard, A.G., Syversen, T.: Generating better partial covering arrays by modeling weights on sub-product lines. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MoDELS, Volume 7590 of Lecture Notes in Computer Science, vol. 7590, pp. 269–284. Springer, New York (2012)

Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: International Symposium on Generative and Component-Based Software Engineering, pp. 10–24 . Springer (2001)

Lopez-Herrejon, R.E., Chicano, F., Ferrer, J., Egyed, A., Alba, E.: Multi-objective optimal test suite computation for software product line pairwise testing. In: 2013 29th IEEE International Conference on Software Maintenance (ICSM), IEEE, pp. 404–407 (2013)

Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Haslinger, E.N., Egyed, A., Alba, E.: A parallel evolutionary algorithm for prioritized pairwise testing of software product lines, pp. 1255–1262 . GECCO'14, ACM, New York, NY, USA (2014)

Nie, C., Leung, H.: A survey of combinatorial testing. ACM Comput. Surv. **43**(2), 11:1–11:29 (2011)

Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines, pp. 196–210. Springer, Berlin (2010)

Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Pairwise testing for software product lines: comparison of two approaches. Softw. Qual. J. **20**(3–4), 605–643 (2012)

Pohl, K., Böckle, G., van Der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)

Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines: footprint and memory consumption. Inf. Softw. Technol. **55**(3), 491–507 (2013). Special Issue on Software Reuse and Product Lines

Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**(2), 101–132 (2000)

Xue, Y., Li, Y.F.: Multi-objective integer programming approaches for solving optimal feature selection problem: a new perspective on multi-objective optimization problems in SBSE. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, New York, NY, USA, ACM, pp. 1231–1242 (2018)