Evolutionary Algorithm for Prioritized Pairwise Test Data Generation

Javier Ferrer, Peter M. Krüse, Francisco Chicano and Enrique Alba

July 5th, 2012

Abstract

Combinatorial Interaction Testing (CIT) is a technique used to discover faults caused by parameter interactions in highly configurable systems. These systems tend to be large and exhaustive testing is generally impractical. Indeed, when the resources are limited, prioritization of test cases is a must. Important test cases are assigned a high priority and should be executed earlier. On the one hand, the prioritization of test cases may reveal faults in early stages of the testing phase. But, on the other hand the generation of minimal test suites that fulfill the demanded coverage criteria is an NP-hard problem. Therefore, search based approaches are required to find the (near) optimal test suites. In this work we present a novel evolutionary algorithm to deal with this problem. The experimental analysis compares five techniques on a set of benchmarks. It reveals that the evolutionary approach is clearly the best in our comparison. The presented algorithm can be integrated into CTE XL professional tool.

1 Introduction

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [13, 14]. From the very first work [16, 21] to nowadays, many approaches have been proposed for solving the automatic test data generation problem. This great effort in building computer aided software testing tools is motivated by the cost and importance of the testing phase in the software development cycle.

Frequently, software testers are faced with situations in which there is not enough time for testing, since the software under test must be finished on time for the release date not to be delayed. Hence, software testers have to deal with limited resources, unfinished systems, and not much time to test the software. Although a tester aims at executing as many test cases as possible, often a test case selection has to be done. The prioritization of test cases is a re-ordering of tests to find faults in early stages. But, if the time run-out, this technique also allows the tester to specify the desired level of coverage and failure-detection.

The result of the prioritization is then a schedule of test cases so that those with the highest priority, according to some criterion, are executed earlier. Criteria can be error rate, occurrence probabilities or risk values.

Combinatorial Interaction Testing (CIT) is a black box sampling technique to complement traditional testing methods. CIT provides a practical way to detect failures caused by parameter interactions with a good trade-off between cost and efficiency. It samples the large combination space using a smaller test suite to cover certain key parameter value combinations. In recent years, several works [5, 25] have explored the effectiveness of the prioritization with combinatorial interaction coverage. Compared with other criteria, prioritization with interaction coverage was found to provide the fastest rate of fault detection. But, the generation of minimal test suites that fulfill the demanded coverage criteria is an NP-hard problem. Thus, search based approaches are required to find the (near) optimal test suites.

Evolutionary algorithms (EAs) have been the most popular search-based algorithms for generating test cases [20]. In fact, the term *evolutionary testing* is used to refer to this approach. In the paradigm of *white box testing* a lot of research has been performed using EAs covering different aspects of the programs, like the presence of flags in conditions [2], the coverage of loops [8], and the existence of internal states [28]. In black box testing, some relevant topics are the generation of test data from Z specifications [15], and the conformance testing [22]. In this paper, we present a search-based approach for test suite optimization using the classification tree method. To the best of our knowledge it is the very first time that an EA is used to deal with the prioritized combinatorial interaction testing problem.

This study aimed to evaluate the performance of metaheuristic techniques for dealing with this problem. In order to achieve this objective we perform a comparison among five algorithms on a set of benchmarks found in the literature. For most benchmark problems the evolutionary approach will be shown to be the best compared to the other greedy approaches.

The rest of the paper is organized as follows. In the next section we define the combinatorial interaction testing, the classification tree method and the classification tree editor. Then, in Section 3 we introduce the prioritized pairwise test data generation problem. Next, in Section 4 we present five different approaches to solve this problem. Specially, we describe in detail our main contribution, the prioritized genetic solver. After that, Section 5 is devoted to the experimental results. We describe the selected benchmark in Section 5.1 and we analyze the obtained result in Section 5.2. Finally, in Section 6, some conclusions and future work are outlined.

2 Combinatorial Interaction Testing

Combinatorial Interaction Testing [7] is an effective testing approach for detecting failures caused by certain combinations of components or input values. The tester identifies the relevant test aspects and defines the corresponding classes. These classes are also called parameters and their elements are called values. We assume that the parameters are disjoint sets. A test case is a set of n values, one for each parameter.

CIT is used to determine the smallest possible subset of tests that covers all combinations of values specified by a coverage criterion with at least one test case. A coverage criterion is defined by its strength t that determines the degree of parameter interaction and assumes that all parameters are considered.

The most common coverage criterion is 2-wise (or pairwise) testing, that is fulfilled if all possible pairs of values are covered by at least one test case in the result test set. A large number of CIT approaches have been presented in the past. A good overview and classification of approaches can be found in [11, 17], or more recently [23]. A good survey that focuses on CIT with constraints is given in [6]. Nearly all existing works investigate pairwise combination methods, but most of them can be extended to arbitrary t-combinations. The only known approaches supporting prioritized test case generation are the Deterministic Density Algorithm (DDA) [3] and an algorithm based on Binary Decision Diagrams (BDD) [26].

2.1 Classification Tree Method

The Classification Tree Method [12] aims at systematic and traceable test case identification for functional testing over all test levels (for example, component test or system test). It is based on the category partition method [24], which divides a test domain into disjoint classes representing important aspects of the test object. Applying the classification tree method involves two steps; designing the classification tree and defining test cases.

Design of the classification tree: The classification tree is based on the functional specification of the test object. For each aspect of interest (called classification), the input domain is divided into disjoint subsets (called classes). In the classification tree method, classifications match parameters and classes match parameter values. Figure 1 shows a classification tree for a database management system. Three aspects of interest (*Access Method*, *Operation*, and *Privileges*) have been identified for the system under test. The classifications are partitioned into classes which represent the partitioning of the concrete input values. In our example the refinement aspect *JavaScript* is identified for the class *Browser* and it is divided further into two classes *Yes* and *No*. All classes have been assigned values of importance. As the figure shows, *Edit* is the most probable *Operation*. *Create* and *Delete* are the subsequent values in descending order of importance. The weights of all classes at the same level

in one classification sum 1 in the model. The class *Browser* has an occurrence rate of 0.7 in *Access Methods*. If the *Access Method* is a *Browser*, *JavaScript* enabled (*Yes*) has an occurrence rate of 0.9 and *No* has an occurrence rate of 0.1. Refinements are interpreted as conditional probability in the occurrence model. The resulting occurrence probability for a *Browser* with *JavaScript* enabled (*Yes*) is 0.63 ($= 0.7 * 0.9$), for *No* it is 0.07, accordingly.

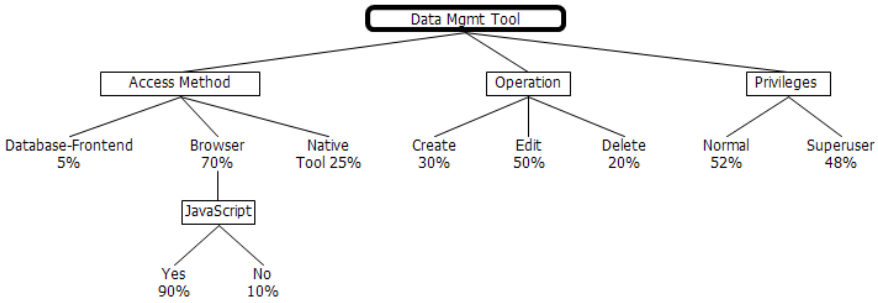

Figure 1: Test Object *Database Management System*

Definition of test cases: Having composed the classification tree, test cases can be defined by combining classes from different classifications. Since classifications only contain disjoint values, test cases cannot contain several values of one classification. The length of the test cases could vary if a class is refined into several classifications.

2.2 Classification Tree Editor

The Classification Tree Editor (CTE XL) is a software tool supporting the classification tree method [19]. It incorporates CIT elements. Current versions of the CTE XL support automated test case generation and user-defined dependency rules. Current test case generation offers four different coverage modes:

- Minimal combination creates a test suite that uses every class from each classification at least once in a test case.
- Pairwise combination creates a test suite that uses every class pair from disjunctive classifications at least once in a test case.
- Threewise combination ("triple-wise") creates a test suite that uses every triple of classes from disjunctive classifications at least once in a test case.

- Complete combination creates a test suite that uses every possible combination of classes from disjunctive classification in a test case.

3 Prioritized Pairwise Test Data Generation

In this section we describe the Prioritized Pairwise Test Data Generation problem. In order to define the problem we first detail how the priorities are assigned to classification tree elements and what is the coverage criterion used.

3.1 Prioritization

Priorities are assigned to the classification tree elements in order to indicate the importance of the element. These priorities are also called *weights*. The higher the weight, the higher importance of the element. These weights can be used to guide the test case generation in order to cover first the most important values.

There exist different prioritization techniques. Elbaum et al. provide good overviews of existing approaches [9, 10]. The following three models were selected to provide a basis for prioritization:

- Prioritization based on a usage model [27] tries to reflect usage distribution of all classes in terms of usage scenarios. Classes with a high occurrence have higher weights than classes with a low occurrence.
- Prioritization based on an error model [9] aims to reflect distribution of error probabilities of all classes. Classes with a high probability of revealing an error have higher weights than classes with a low probability.
- Prioritization based on a risk model [1] is similar to prioritization based on an error model but also takes error costs into account. Classes with a high risk have higher weights than classes with a low risk.

Once we have assigned weights to each value (class), we need to define weights for the pair of classes. This is done by multiplying the weight of each class involved in the pair. Following the previous example the pair weight for (Yes (Javascript), Create) is 0.63 * 0.3 = 0.189

3.2 Coverage Criteria

We need to define a measure of the quality of a test suite in order to decide which test suite is the best one. In white-box testing the use of code coverage measures, like branch coverage or sentence coverage, is common. We also use here a coverage measure which is based on the weights of the class pairs covered. The so called *weight coverage* is defined as:

$$WC = \frac{\text{sum of weights of covered class pairs}}{\text{sum of weights of all coverable class pairs}} \quad (1)$$

The metric is relative, i.e. considers the fact that classes may not be coverable because of dependencies.

3.3 Problem Formulation

Once we defined the concepts of weight and the coverage criteria, we can define the problem. Let T denote the set of all the possible test cases and let $s \in T^*$ denote a finite sequence of test cases from T where s_i is used to refer to the i-th test case. Given m values of interest for the weight coverage WC_1, WC_2, ..., WC_m, we define the functions $f_1(s)$, $f_2(s)$, ..., $f_m(s)$ in the following way: $f_i(s)$ is the length of the minimum prefix of s with weight coverage greater than or equal to WC_i. That is, $f_i(s)$ gives the number of test cases from the beginning of s we have to run in order to get weight coverage WC_i. The problem can be formulated as finding the sequence of test cases s having minimum values for all the $f_i(s)$ functions.

We can think in these problems as a Multi-objective problem. However, it is not the case the all the objectives have the same importance. Finding a minimum number of test cases for covering the lowest values of the weighted coverage is usually more important than minimizing the number of test cases for the highest values of WC. The reason is that the software tester wants to cover fast the more important class pairs.

4 Solution Approaches

In this section we describe five different approaches used to solve the prioritized pairwise test case generation problem. We first introduce in detail our evolutionary approach, a genetic algorithm. To the best of our knowledge this is the first time an evolutionary approach has ever been applied to the target problem. Then, we briefly describe two deterministic algorithms that we have implemented for comparison purposes, the Prioritized Pairwise Combination algorithm (PPC) and the Plain Pairwise Sorting (PPS). We finally present the Deterministic Density Algorithm (DDA) developed in [4] and an algorithm based on Binary Decision Diagrams (BDD) introduced by Lee [18].

4.1 Prioritized Genetic Solver

The Prioritized Genetic Solver (GS) is a novel evolutionary approach that constructs an entire test suite taking into account priorities in the generation. It is a constructive algorithm that adds one new test case to the partial solution in each iteration until all pairwise combinations are covered. In each iteration the algorithm tries to find the test datum that adds more coverage to the partial solution.

We show the main loop of our GA in Algorithm 1. At the beginning the test suite is initialized with an empty list (line 2). In each iteration of the external loop (lines 3-19) the algorithm creates a random population of individuals (line 5). Then, it enters in an inner loop that which applies the traditional steps of a generational evolutionary algorithm (lines 7-17). That is, some individuals (solutions) are selected from the population $P(t)$, they are recombined, mutated and evaluated and they are finally inserted in the offspring population Q. In line

15 the old and the new populations are used to build the population for the next generation $P(t+1)$. The best individuals among $P(t)$ and Q are included in $P(t+1)$. The internal loop is executed until a maximum number of evaluations is reached. Then, the best individual (test datum) found is included in the test suite (line 18) and the external loop starts again.

Algorithm 1 Pseudocode of GA.

```
 1: proc Input:(GA)       //Algorithm parameters in 'GA'
 2: TS ← ∅           // Initialize the test suite
 3: while not Termination_Condition() do
 4:    t=0
 5:    P(t) ← Create_Population() // P = population
 6:    Q ← ∅                      // Q = auxiliar population
 7:    while Evals < TotalEvals do
 8:       for i ← 1 to (GA.popSize) do
 9:          parents←Selection(P(t))
10:          offspring←Recombination(GA.Pc,parents)
11:          offspring←Mutation(GA.Pm,offspring)
12:          Evaluate_Fitness(offspring)
13:          Insert(offspring,Q)
14:       end for
15:       P(t+1) := Replace (Q,P(t))
16:       t= t + 1
17:    end while      //internal loop
18:    Insert(best_solution, TS)
19: end while         //external loop
20: end_proc
```

In this work we have used a one point crossover with probability 1 of recombining the two selected individuals. This operator is able to put together good solution components that are scattered in the small population used of 4 individuals. Regarding the mutation operator, it iterates over all the components in the solution vector changing its value by a random one of the same classification with probability 0.05. The maximum number of evaluations used as stopping criterion in the internal loop is $5,000$ (line 7) while the stopping condition of the external loop is to achieve full pairwise coverage (line 3).

4.1.1 Objective Function

Our algorithm aims at generating an entire test suite (to cover all pairwise combinations). The algorithm constructs the solution by generating the best test datum at a time until all pairwise combinations are covered. The best test datum is the one that most reduce the weighted value of the set of remaining pairs to cover.

The computation of the fitness value for each solution is done through the following process: The algorithm computes the combined class pairs of the solution (test datum). After that, it removes these pairs from the set of remaining pairs. Finally, the fitness value of a solution is computed as the sum of the weights of the remaining pairs. That is, the objective value of a proposed test datum is the sum of the weights of the class pairs that are not covered after adding the test datum to the suite. This objective function must be minimized in order to take first the test datum covering the class pairs with higher weights. As the search

progresses the computational cost of computing the fitness function is reduced, since less class pairs remain uncovered.

4.2 Prioritized Pairwise Combination

In this algorithm, the class pair with the highest weight from the set of uncovered class pairs is chosen for the new test case. We determine all candidate test cases containing this class pair and calculate the index values for these candidates. This index value includes the weights and the number of newly covered class pairs.

PPC then selects the test case with the highest assigned index value. This way, we can guarantee that the n first test cases cover the n more important class pairs. The generated test suite may be slightly larger than the result of the plain pairwise combination since weights are taken into account. The generation process using PPC is deterministic: the same test suite is generated for identical classification trees.

4.3 Plain Pairwise Sorting

This algorithm first applies a plain pairwise algorithm (the one integrated in CTE XL professional tool), that computes a sequence of test cases covering all the class pairs. Then it sorts the test cases taking into account their absolute weight at first. Then, it applies as many discriminatory reorderings as test cases.

Note that this approach does not guarantee coverage of any n most important class pairs by the n first test cases. However, the generated test suite will have exactly the same size as the plain pairwise combination, as the suite does not grow by sorting. The generation process using sorting is deterministic, however its results differ from the PPC results.

4.4 Deterministic Density Algorithm

In the Deterministic Density Algorithm (DDA) one test datum is constructed at a time and new test data are generated until all t-tuples are covered. Each classification is assigned a class value one-at-a-time. A classification that has been assigned a class value is referred to as fixed; one that has not, as free. For each classification, the class value that covers the largest density is selected. Then, a density formula calculates the likelihood of covering future tuples. To modify DDA to account for prioritization, the density formula is modified. Instead of computing the ratio of uncovered pairs to be covered, the amount of weight to be covered is computed.

4.5 Binary Decision Diagrams Algorithm

Binary Decision Diagrams (BDD) are acyclic directed graphs used to represent propositional logical formulas. In [26] the authors introduced an approach based on the modeling of the combinatorial interaction test problem as a single

propositional logic formula. They constructed the formula such that the set of satisfying interpretations of the formula corresponds to the set of valid test cases and such that a one-to-one relation between a satisfying interpretation of the formula and a valid test case of the CIT problem exists. The formula is the conjunction of a subformula representing the set of all test cases and a subformula representing the set of constraints. They have used this formula in a greedy algorithm, in the following BDD, to select test cases until the desired coverage criterion is fulfilled.

5 Experiments

This section is aimed at describing the experiments performed on a benchmark of programs. First, we describe the experimental benchmark and then we analyze the results of the comparison among the algorithm presented in Section 4.

5.1 Experimental Benchmark

For a more detailed and systematic evaluation, we use the set of benchmarks proposed in [4]. The scenarios $S1 - S8$ are given in Table 1. The number of classes of each scenario are given in a shorthand notation, where for example S5 with $8^2 7^2 6^2 2^4$ consists of 2 classifications with 8 classes, 2 classifications with 7 classes, 2 classifications with 6 classes, and 4 classifications with 2 classes.

Table 1: Scenarios and number of factors.

Scenarios	#Classes
S1	3^4
S2	10^{20}
S3	3^{100}
S4	$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$
S5	$8^2 7^2 6^2 2^4$
S6	$15^1 10^5 5^1 4^1$
S7	$3^{50} 2^{50}$
S8	$20^2 10^2 3^{100}$

The given benchmark uses four different weight distributions applied to the eight scenarios. The distributions are:

- d1 (equal weights): All classes have the same weight
- d2 (50/50 split): Half of the weights for each classification are set to 0.9 the other half to 0.1
- d3 ($(1/vmax)^2$ split): All weights of classes for a classification are equal to $(1/vmax)^2$, where $vmax$ is the number of classes associated with the classification
- d4 (random): Weights are randomly distributed.

Table 2: Number of test cases needed for the GA, PPC, and PPS algorithms in eight scenarios and for four distributions. When significant differences exist between the GS and other algorithm we add an asterisk.

Scenario	Coverage	d1			d2			d3			d4		
		GS	PPC	PPS	GS	PPC	PPS	GS	PPC	PPS	GS	PPC	PPS
S1	25%	3	3	3	1	1	1	3	3	3	2	2	2
	50%	5	5	5	1	1	2	5	5	5	3	3	3
	66%	6.29	7*	6*	1	1	3*	6.29	7*	6*	5	5	5
	75%	7.48	8*	7*	3	3	4*	7.35	8*	7*	6	6	5*
	90%	9.3	9*	9*	6.29	7*	5*	9.18	9*	9*	8	8	7*
	95%	9.93	10	9*	8	8	7*	9.88	10	9*	9	10*	8*
	99%	10.42	10	9*	10.28	11*	8*	10.19	10	9*	11	11	9*
S2	25%	26	27*	27*	8.23	9*	12*	26	27*	27*	12	12	19*
	50%	56.1	56	60*	19	18*	36*	56.11	56	60*	31	31	47*
	66%	80.05	79*	89*	29	27*	60*	80.04	79*	89*	49.83	50	74*
	75%	96,75	95*	110*	38.13	36*	79*	96.78	95*	110*	64.02	65*	95*
	90%	134.48	132*	162*	89.69	87*	131*	134.48	132*	162*	102.33	104*	150*
	95%	154.84	152*	190*	122.77	121*	163*	154.76	152*	190*	125.59	129*	181*
	99%	182.94	180*	228*	169.9	169*	212*	182.93	180*	228*	163.14	169*	223*
S3	25%	3	3	3	1	1	2*	3	3	3	1	1	3*
	50%	6	6	6	1	1	4*	6	6	6	3	3	5*
	66%	8	8	8	1.76	1*	7*	8	8	8	5	5	8*
	75%	9.02	9	10*	3.64	4*	8*	9.03	9	10*	7	7	10*
	90%	14	15*	16*	8	9*	13*	14	15*	16*	12	12	15*
	95%	17.9	20*	20*	11.84	12	16*	17.91	20*	20*	15	15	19*
	99%	24.13	37*	26*	19	19	23*	24.07	37*	26*	21	21	26*
S4	25%	9.41	11*	10*	3	3	4*	5	4*	4*	7	6*	7
	50%	21.02	24*	22*	7	8*	11*	9	8*	8*	15.09	15	18*
	66%	31.95	36*	34*	11.62	12*	17*	13.03	12*	14*	23.12	23	28*
	75%	39.67	45*	42*	15.46	16*	23*	17	16*	19*	29.09	29	36*
	90%	59.12	64*	63*	32.7	36*	38*	30.84	30*	35*	45.02	46*	56*
	95%	70.57	74*	74*	49.4	53*	51*	43.28	42*	46*	55.37	59*	67*
	99%	88.24	86*	88	77.41	76*	77*	72.17	72	70*	75.76	82*	82*
S5	25%	8	8	7*	2.46	2*	3*	2	2	2	3	3	4*
	50%	16.97	18*	17	5.46	6*	8*	4	4	4	8	8	10*
	66%	27.09	28*	27	9.46	10*	15*	7	7	7	13.12	14*	17*
	75%	34.65	35*	34*	12.52	13*	21*	9	9	10*	18.05	19*	24*
	90%	50.04	50*	51*	26.89	28*	37*	14.01	15*	19*	32.13	33*	41*
	95%	58	57*	59*	42.06	43*	46*	25.22	23*	25*	41.32	44*	50*
	99%	69.5	66*	68*	60.86	61	62*	52.54	52*	54*	56.61	60*	65*
S6	25%	22	23*	22	7	7	9*	12	12	12	10.04	11*	14*
	50%	45,98	52*	49*	16	17*	24*	25	26*	26*	25.98	27*	36*
	66%	67.17	74*	74*	24.53	25*	39*	36	38*	41*	40.51	43*	57*
	75%	82.48	89*	92*	31.33	31	53*	44.1	46*	53*	52.1	56*	73*
	90%	117.65	123*	131*	71.71	73*	92*	81.15	83*	92*	84.32	90*	111*
	95%	136.98	139*	149*	105.23	106*	114*	105.4	107*	120*	104.83	112*	131*
	99%	158.19	159*	169*	146.44	148*	153*	146.43	149*	158*	139.87	148*	160*
S7	25%	2	2	2	1	1	2*	2	2	2	1	1	2*
	50%	4	4	4	1	1	3*	3	3	4*	2	2	4*
	66%	6	6	6	1	1	4*	5	5	5	4	4	6*
	75%	7	7	8*	2	2	5*	6	6	7*	5	5	7*
	90%	11	11	12*	6	6	9*	9	10*	10*	8.81	9	11*
	95%	14	14	15*	8.06	9*	12*	12	13*	13*	11	11	14*
	99%	20.99	20*	21	14.65	14*	18*	19	21*	19	16.99	17	21*
S8	25%	3	3	-	1	1	-	2.82	3	-	2	3*	-
	50%	8	8	-	3	3	-	5.64	6	-	5	6*	-
	66%	12	13*	-	6	7*	-	7.52	8	-	9	8*	-
	75%	16.38	18*	-	9	12*	-	9.39	9*	-	14	9*	-
	90%	37.86	64*	-	20.37	30*	-	14.1	15	-	31.66	15*	-
	95%	52.97	92*	-	35.06	56*	-	17.71	19*	-	46.62	19*	-
	99%	130.87	145*	-	97.32	114*	-	25.94	28*	-	118.22	28*	-
	Times	22	12	7	26	11	3	19	14	5	24	8	4

5.2 Experimental Analysis

In this section we analyze the behaviour of the algorithms with the aim of highlighting the technique that works better. In a first subsection, we study our three algorithms, the GS, the PPC and the PPS algorithm. In this subsection we present an exhaustive comparison among our proposed algorithms. In a second subsection, we compare our best algorithm, the GS, with the other two algorithms: DDA and BDD.

5.2.1 Comparison between our algorithms

In this comparison we are evaluating our three algorithms (GS, PPC and PPS) in order to analyze their behaviour for the computation of minimal test suites when we use weight coverage as adequacy criterion. In general the GS performs better than the PPC and the PPS algorithm in most cases, nevertheless we want to highlight the weaknesses and strengths of each algorithm. The detailed results for all the scenarios, distributions, and weight coverage are given in Table 2. We should take into account that the observations of the number of test cases needed to achieve the different values of coverage are taken in the same execution of the algorithm. We execute the GS 100 times for a particular scenario-distribution combination, then we have done $3,200$ executions of the GS algorithm. In order to validate the experimental results we compared the 100 samples of the GS with the values of the deterministic algorithms using the Wilcoxon rank-sum test with 95% of statistical confidence. In Table 2 we marked with an asterisk ($*$) the values that are statistically different with respect to the GS's value.

Let us first analyze the results obtained by distribution. If all the weights of the interactions are the same (distribution d1), the GS performs better than the other algorithms. In particular, the GS is the best 22 times, while the PPC algorithm is the best 12 times, and the PPS algorithm only 7 times. Although the GS is the best algorithm, it is worse than the PPC algorithm when 99% of coverage is required. Besides, the PPS algorithm obtains its better results with this distribution. This behaviour was expected because solving the problem with d1 is the same as the pairwise combination problem without priorities as we commented in section 5.1. Particularly, the sorting in PPS is carried out after executing of a plain pairwise algorithm, which do not use the weight, since it only tries to cover as many class pairs as possible in each test datum. This is the reason why the PPS algorithm works well with the distribution d1.

In d2, when extreme values are used, GS obtains the best results. It is the best algorithm in 26 scenarios, while the PPC is the best in 11 and the PPS in 3. The GS performs quite well for all target weights and it is specially good with 95% of weight coverage, where it is the best in 6 out of 8 scenarios (with statistically significant differences in most of the cases).

In d3, the GS is slightly better than the PPC algorithm. The GS obtains the minimum number of test cases in 19 scenarios while the PPC algorithm is the best in 14. Thus, the GS and the PPC algorithm obtain similar results.

When the target weight is 75%, the PPC algorithm is better in 5 out of the 8 scenarios, but when high coverage is required (90%) the GS is better in the same proportion (5 out of 8).

When random weights are used (distribution d4), the PPC and PPS algorithms obtain the worse results of all the distributions, whilst the GS behaves very well. The GS is the best in 24 scenarios, the PPC algorithm in 8 and the PPS only in 4. At the beginning of the search, when the target weights are less than 75%, the differences among the algorithms are not very large, but for weights larger than 75% the GS is much better than the others.

The GS is the best algorithm for all the studied distributions as we have commented in the previous paragraphs. However, if we analyze the results by considering each scenario independently, some weaknesses of the GS appear. Let us analyze the influence of the benchmark scenarios in the obtained results. In Table 3 we summarize the number of scenarios in which one algorithm is better than the others.

Table 3: Number of times that one algorithm is better than the other two for each instance.

Scenario	GS	PPC	PPS
S1	0	0	12
S2	8	18	0
S3	9	3	0
S4	14	9	1
S5	13	6	3
S6	24	1	0
S7	5	2	0
S8	19	6	-

In S1, the GS is not the best in any combination distribution/scenario. In this small scenario, the PPS algorithm is the best (12 times), while the other algorithms cannot outperform PPS. Since it is a small scenario, there is no reason to use a prioritized test case generation, but at least these bad results should be taken into account as a disadvantage of the prioritized generation.

In S2, the PPC algorithm is the best. It is the best in 18 observations, the GS is the best in 8 and the PPS algorithm never outperforms the previous algorithms in this scenario. In d1, d2 and d3 distributions the GS is the best for 25% target weight, which means that the algorithm is able to combine high weight pairs in early test cases. For the rest of target weights, the PPC algorithm is the best.

In the other six scenarios we can observe that GS is usually the algorithm computing the best results. In particular, in s6 it is the best in 24 observations out of 28, while the PPC algorithm is the best only in 1 observation, and the PPS algorithm is never the best.

We also show in Table 4 the number of observations where there exists significant difference among the GS, the PPC and PPS algorithms. The number in front of a triangle up (▲) is the number of times that the results of GS are better than the ones of the other algorithms with statistically significant difference. The number in front of a triangle down (▽) is the number of times GS is worse with statistically significant difference. According to the results of

Table 4, the GS is better than the other two algorithms in all the distributions. In the comparison between GS and PPC, the best distributions for the GS are d1 and d2 where the GS outperforms the PPC algorithm in 28 and 26 times, respectively. If we compare the GS and the PPS algorithm, the differences between the algorithms are even larger. In d2 and d4 the GS outperforms in 42 and 41 times, respectively. Thus, based on the statistical tests, we can state that the GS is clearly the best algorithm.

Table 4: Number of observations where there exists significant difference among the GS, the PPC and PPS algorithms.

Algorithm-Distribution	PPC	PPS
GS-d1	28▲10▽	29▲8▽
GS-d2	26▲9▽	42▲3▽
GS-d3	19▲10▽	29▲8▽
GS-d4	22▲6▽	41▲4▽

In conclusion, the GS obtains better test suite size for all the distributions. There is not much difference among the distributions, thus the good performance of the GS does not depend on the distribution itself. According to the target weights, the GS is always the best except when the target weight is 75%. Under that circumstances the PPC algorithm and the GS obtain similar results. PPC is better in 12 observations while the GS is better in 11. Thus, we consider that GS is better in most target weights as well. Despite the optimization solver is the best in most cases, there are two scenarios, S1 and S2, where the GS is not the best. In the first one, S1 (the smallest), sorting is clearly the best option. In the second one, S2, GS is not the best but for the firsts weight coverage it obtains the best results. This behaviour is desirable when we are computing prioritized test data.

5.2.2 Comparison between Genetic Solver and other existing algorithms

In this section we compare the results of our evolutionary algorithm with the ones of other approaches found in the literature. We show in Table 5 the results of three approaches (GS, DDA and BDD), with eight scenarios, four distributions and three values of weight coverage 50%, 75% and 100%. We have chosen these values according to the information available on these approaches extracted from the literature.

Let us analyze the obtained results by weight coverage. If we focus on 50% of weight coverage, we can see that the GS is the unique algorithm that is able to outperform the others. The GS is the best in 11 observations. The GS is even better in the first measures of coverage, when it is more difficult to generate differences. For 50% of weight coverage the DDA algorithm needs 5.24% more test cases and the BBD algorithm needs 23.34% more test cases than the GS. Thus, we can state that our algorithm is clearly the best for 50% of weight coverage.

For 75% of weight coverage, the GS is the best again. Our algorithm is better than the other algorithms in 18 observations, while the DDA algorithm

Table 5: Number of test cases needed for the GA, DDA, and BDD algorithms in eight scenarios and for four distributions. When significant differences exist between the GS and other algorithm we add an asterisk. Algorithm legend: $G \equiv GS$, $D \equiv DDA$ and $B \equiv BDD$

	50%			75%			100%		
	G	D	B	G	D	B	G	D	B
S1-d1	5	5	5	7,48	7*	7*	10,42	9*	10
S2-d1	56,1	57*	62*	96,76	97	112*	203,29	220*	251*
S3-d1	6	6	6	9,02	9	11*	31,91	32	33*
S4-d1	21,02	22*	26*	39,67	40*	47*	97,53	95*	94*
S5-d1	15,97	16	19*	33,65	33*	37*	75,16	72*	74*
S6-d1	45,98	47*	62*	82,48	84*	105*	171,38	175*	184*
S7-d1	4	4	4	7	7	8*	29,47	29*	28*
S8-d1	8	8	13*	16,38	16*	35*	437,81	400*	400*
S1-d2	1	1	1	3	3	3	11,98	14*	14*
S2-d2	19	19	21*	38,13	39*	49*	221,53	256*	278*
S3-d2	1	1	1	3,64	4*	4*	31,45	35*	35*
S4-d2	7	7	10*	15,46	16*	22*	103,39	203*	100*
S5-d2	5	5	7*	12,06	12	15*	79,21	83*	83*
S6-d2	16	16	28*	31,33	32*	55*	185,94	192*	207*
S7-d2	1	1	1	2	2	2	28,81	31*	29
S8-d2	3	3	3	9	9	16*	438,4	400*	400*
S1-d3	5	5	5	7,35	8*	7*	10,19	13*	10
S2-d3	56,09	60*	62*	96,78	112*	112*	203,2	325*	249*
S3-d3	6	6	6	9,03	10*	11*	31,9	37*	33*
S4-d3	8	8	8	16	19*	18*	113,24	115*	137*
S5-d3	4	4	5*	9	10*	10*	84,66	94*	101*
S6-d3	25	26*	27*	44,1	51*	53*	179,63	208*	222*
S7-d3	3	3	4*	6	6	7*	30,81	33*	33*
S8-d3	5,76	6	6	9,59	10	11*	407,47	417*	463*
S1-d4	3	3	4*	6	5*	6	12	13*	13*
S2-d4	31	32*	44*	64,02	67*	94*	220,59	265*	286*
S3-d4	3	3	5*	7	7	11*	31,28	34*	47*
S4-d4	14,09	25*	17*	28,09	29*	33*	99,95	100	108*
S5-d4	8	9*	10*	18,05	19*	24*	76,96	82*	88*
S6-d4	25,98	27*	35*	52,1	56*	70*	178,44	192*	208*
S7-d4	2	2	3*	5	5	7*	28,05	32*	42*
S8-d4	5	5	8*	14	15*	26*	418,77	406*	406*
	11	0	0	18	5	1	23	2	4

is the best in 5 observations and the BDD algorithm is the best in only one. For this value of weight coverage, the DDA and BDD algorithms need 4.98% and 28.63% more test cases, respectively. Once again the GS obtains better results than the DDA and BDD algorithms.

For total coverage, the GS is better than the other algorithms in 23 observations, the DDA in 2 and the BDD in 4. The difference between the GS and the others here is the largest one. This is a very interesting property of our GS algorithm, since it is not usual that an algorithm which is good for low/medium values of weight coverage is also good for total coverage. It is noteworthy that we do not configure the algorithm to obtain good test suite size for a particular value of coverage, but we just try to achieve all values of coverage with the minimum number of test data.

Regarding the different distributions, the GS maintains a good behaviour in all the distributions. The GS algorithm is the best in 10 observations with distributions d1 and d2. In addition, the GS is even better in the distributions d3 and d4, since it is the best in 16 observations. Besides, if we focus on the other algorithms, the differences appear. The d1 is clearly the best distribution for them. We should highlight that d1 (equal weights) is the distribution where the priority is not used, the same weight is used for all the classes.

In order to provide a high level of confidence to these results, we have performed statistical tests. The results are shown in Table 6. There are some

differences among the algorithms; we again take as a reference the GS values. Despite the GS and the DDA are both statistically better in 7 times for the d1 distribution, in the rest of values of the table we can see that the GS is clearly the best algorithm. In the comparison between GS and DDA, the GS is significantly better in 49 observations while the DDA is only significantly better in 10 observations. In the comparison between GS and BDD, the GS is significantly better in 71 observations while the BDD is only significantly better in 9 observations. Thus, we can state again that the GS is the best overall algorithm for the prioritized pairwise combination problem.

Table 6: Number of observations where there exists significant difference among the GS, the DDA, and BDD algorithms.

Algorithm-Distribution	DDA	BDD
GS-d1	7▲7▽	15▲5▽
GS-d2	10▲1▽	16▲2▽
GS-d3	16▲0▽	18▲1▽
GS-d4	16▲2▽	22▲1▽

6 Conclusions

In this paper we have studied the prioritized pairwise test data generation problem with the aim of analyzing the performance of several approaches. We have compared five different approaches, three of them proposed by the authors. Our three approaches have been successfully implemented in the classification tree editor, thus they could be integrated in a professional tool. The other two approaches have been extracted from the literature. A benchmark of eight scenarios and four weight distributions were used to execute all the algorithms. We have performed some experiments on these 32 different scenario/distribution combinations and for different values of weight coverage, which makes our study meaningful.

One of our proposals is a genetic algorithm. To the best of our knowledge, it is the first time that an evolutionary algorithm is used to solve this problem. The genetic algorithm outperforms the other algorithms in most scenarios and distributions, it is the best choice when one has some time restrictions or the execution of a test case is quite costly.

After analyzing the results obtained by all the algorithms we can draw some advices about which technique should be used. If the results of a particular technique like PPS are good for equal weight distribution (d1) but are not good enough for the other distributions, then the technique is designed to be used without priorities. If one really needs some values of weight coverage for different scale scenarios and non-uniform distributions, the genetic solver is the best choice. But, if the GS does not achieve a satisfactory result for a particular configuration, one should use the PPC algorithm. Finally, if the test suites have already been computed, the PPS algorithm should be used in order to give a better ordering of the test cases.

Future work will verify these findings with larger scenarios and more distributions. We would like to deal with prioritized t-wise coverage that is an

open and interesting field for research, besides, it could pose a real challenge for the community. In this way, we also want to advance in designing better evolutionary algorithms, who seems to be very effective for solving this kind of problems.

References

[1] S. Amland. Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287–295, 2000.

[2] A. Baresel, D. W. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *ISSTA 2004*, pages 108-118, 2004.

[3] Bryce and Memon. Test suite prioritization by interaction coverage. In *DOSTA'07*, pages 1-7, New York, 2007.

[4] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960-970, 2006.

[5] R. C. Bryce and C. J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *GECCO '07*, pages 1082-1089, New York, 2007.

[6] D. Cohen and Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07*, New York,2007. ACM. 594071.

[7] M. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. *SIGSOFT Softw. Eng. Notes*, 31:1–9, November 2006.

[8] E. Díaz, R. Blanco, and J. Tuya. Tabu search for automated loop coverage in software testing. In *(ICKEDS06)*, pages 229-234, Porto, 2006.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159 –182, feb 2002.

[10] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12:185–210, September 2004.

[11] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[12] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[13] M. Harman. The current state and future of search based software engineering. In *(ICSE/FOSE '07)*, pages 342-357, Minneapolis, May 2007.

[14] M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, December 2001.

[15] Y. X. Jones B, Sthamer H and E. D. The automatic generation of software test data sets using adaptive search techniques. In *3rd International Conference on Software Quality Management*, pages 435–444, 1995.

[16] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[17] R. Kuhn, Y. Lei, and R. Kacker. Practical combinatorial testing: Beyond pairwise. *IT Professional*, 10:19–23, May 2008.

[18] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985:999, July 1959.

[19] E. Lehmann and J. Wegener. Test case design by means of the cte xl. In *EuroSTAR 2000*, Kopenhagen, Denmark, December 2000.

[20] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[21] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.

[22] T. N. A search-based automated test-data generation framework for safety critical software. Master's thesis, PhD Thesis, University of York, 2000.

[23] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.

[24] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31:676–686, June 1988.

[25] X. Qu, M. Cohen, and K. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *ICSM 2007*, pages 255 –264, oct. 2007.

[26] E. Salecker, R. Reicherdt, and S. Glesner. Calculating prioritized interaction test sets with constraints using binary decision diagrams. ICSTW '11, pages 278–285, Washington, 2011.

[27] G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical testing of software based on a usage model. *Softw. Pract. Exper.*, 25:97–108, January 1995.

[28] Y. Zhan and J. A. Clark. The state problem for test generation in simulink. In *GECCO'06:*, pages 1941–1948. ACM Press, 2006.