Generating Cost-Aware Covering Arrays For Free

Mustafa Kemal Taş, Hanefi Mercan, Gülşen Demiröz, Kamer Kaya and Cemal Yilmaz
{mkemaltas,hanefimercan,gulsend,kaya,cyilmaz}@sabanciuniv.edu

Faculty of Engineering and Natural Sciences, Sabancı University, Istanbul

Abstract. Software systems generally have a large number of configurable options interacting with each other. Such systems are more likely to be prone to errors, crashes, and faulty executions that are usually caused by option interactions. To avoid such errors, testing all possible configurations during the development phase is usually not feasible, since the number of all possible configurations is exponential in the order of number of options. A t-way covering array (CA) is a 2-dimensional combinatorial object that helps to efficiently cover all t-length option interactions of the system under test. Generating a CA with a small number of configurations is important to shorten the testing phase. However, the testing cost (e.g. the testing time) may differ from one configuration to another. Currently, most sequential tools can generate optimum CAs in terms of number of configurations, but they are not cost-aware, i.e., they cannot handle the varying costs of configurations. In this work, we implement a parallel, cost-aware CA-generation tool based on a sequential tool, `Jenny`, to generate lower-cost CAs faster. Experimental results show that our cost-aware CA construction approach can generate 32% and 21% lower cost CAs on average for t=2 and t=3, respectively, compared to state-of-the-art CA-generation tools. Moreover, the cost-awareness comes for free, i.e., we speed up our algorithm by leveraging parallel computation. The cost models and cost reduction techniques we propose could also be adapted for other existing CA generation tools.

Keywords: Software testing; Testing cost; Combinatorial interaction testing; Covering arrays; Cost-aware testing; Parallel covering array generation

1 Introduction

In software testing, testing all possible configurations of the System Under Test (SUT) is not always feasible. For example, a recent version of the Apache v2.2 web server has 172 configurable options and 1.8×10^{55} unique configurations. Hence, a full coverage of all configurations becomes quite expensive and is not feasible. On the other hand, considering that most of the faults occur due to a small number of option interactions [1], testing all such combinations is usually sufficient to detect the faults. Combinatorial Interaction Testing (CIT) [2, 3] is

commonly used to test such software systems with a large configuration space. Instead of an exhaustive test, CIT samples the configuration space based on coverage criteria and tests each of these samples individually. Typically, such samples contain some specific combination of option-value pairs. For example, pairwise testing requires each possible option-value pair combinations to be tested at least once. For that purpose, in CIT, *covering arrays* are commonly used as test suites.

A *t-way covering array* (CA) is a matrix with N rows (configurations) and k columns (options) in which every possible t length option-value pair exists in (is covered by) some row at least once where t is called the strength of the CA. Many empirical results state that most of the faults in practice are caused by the interactions of only a small number options, i.e., mostly $2 \leq t \leq 6$ [1], therefore, CIT becomes an efficient approach in detecting these kind of faults.

Generating a CA with minimum size is known to be an NP-complete problem [4, 5]. Thus, developing an approach to generate minimal CAs is not practical. Instead, heuristic approaches such as simulated annealing [6] or greedy algorithms [7, 8] have been commonly used for CA generation. Simulated annealing based heuristics are shown to produce smaller CAs whereas greedy algorithms are faster [9]. We refer the user to a recent survey of the available algorithms [10].

To reduce the actual cost of testing, most of the tools aim to generate smaller CAs since they assume that cost of testing each configuration is the same. However, empirical results have shown that cost may vary from one configuration to another [11, 12], thus minimizing the size does not necessarily mean minimizing the cost of testing. Avoiding costly combinations may increase the CA size but may lead to a shorter testing time. For example, in a study we conducted on MySQL – a highly-configurable database management system, we observed that the cost of configuring MySQL Community Server (a core component of the system) with its default configuration takes about 6 minutes on average. On the other hand, configuring the server with NDB cluster storage support – a feature that enables clustering of in-memory databases, takes about 9 minutes (50% more) [11]. Consequently, reducing the number of configurations that include the NDB feature in a CA without adversely affecting the coverage properties of the array, can greatly reduce the amount of time required for testing. For testing such systems (with varying testing costs) a novel object for testing, a *cost-aware covering array*, is introduced by Demiroz and Yilmaz [11, 13]. One of the closest works to our cost-aware CA is test prioritization in CAs, where only the cost of switching the order of configurations are considered [14].

We propose a novel approach to enable the cost-awareness within a greedy CA generation tool. `Jenny` is an open-source tool with good experimental results [15], hence been selected for this study. We add the cost-awareness at multiple steps of `Jenny` and significantly reduce the CA costs. None of the available tools [16] are cost-aware and almost all of them are sequential except Pairwiser [17] and `Jenny` is not an exception. We implement a parallel version of `Jenny` to generate cost-aware CAs more efficiently via utilizing multicore CPUs. Empirical results show that generating cost-aware CAs is not necessarily more costly than generating standard CAs. With the additional parallelism, on average, 32% and 21% cheaper

CAs can be generated up to 2.5 and 3.5 times faster than the original tool with 8 threads, for t=2 and t=3, respectively. Thus, cost-awareness comes for free.

The rest of the paper is organized as follows: Section 2 introduces the notations and explains CIT in detail as well as describing some of the existing covering array generation tools. The methods for cost-awareness and parallelism are explained in detail in Section 3 and experimental results are presented in Section 4. Section 5 concludes the paper and presents future work.

2 Background and Notations

Combinatorial Interaction Testing [2] takes a configuration model of the SUT as input and produces a test suite, such as a t-way Covering Array (CA). A configuration model includes the set of k many configurable options $O=\{o_1, \ldots, o_k\}$ where each option o_i $(1 \leq i \leq k)$ takes a value from a finite set of values $V=\{V_1, \ldots, V_k\}$. The model may also contain the coverage criteria, the costs of option combinations and the constraints.

2.1 Standard covering arrays

Given a configuration space model $M=<O, V>$ where $k=|O|$:

Definition 1. *A t-tuple $\phi_t=\{<o_{i_1}, v_{j_1}>, <o_{i_2}, v_{j_2}>, \ldots, <o_{i_t}, v_{j_t}>\}$ is a tuple comprised of option-value pairs for a combination of t distinct configuration options, such that $1 \leq t \leq k$, $1 \leq i_1 < i_2 < \cdots < i_t \leq k$, and $v_{j_p} \in V_{i_p}$ for $p=1, 2, \ldots, t$. Let Φ_t be the set of all t-tuples.*

Definition 2. *A configuration c is a k-tuple, i.e., $c \in \Phi_k$. The configuration space $C=\{c : c \in \Phi_k\}$ is the set of all configurations.*

Definition 3. *A t-way covering array $CA(t, M=<O,V>)$ is a set of configurations, in which each t-tuple appears at least once, i.e., $CA(t, M)=\{c_1, c_2, \ldots, c_N\}$, such that $\forall\, \phi_t \in \Phi_t\, \exists\, c_i \supseteq \phi_t$, where $c_i \in C$ for $i=1, 2, \ldots, N$.*

Standard covering arrays are computed, such that all t-tuples are covered (i.e., appear at least once) in a minimum number of configurations. A 2-way CA with 5 binary options ($V_i=\{0, 1\}$ for all o_i) is given in Figure 1. As it can be seen, every possible 2-way option-value is covered at least once by a configuration.

o_1	o_2	o_3	o_4	o_5
0	1	1	1	1
1	0	0	0	0
0	0	0	1	0
1	1	1	0	0
1	0	0	0	1
0	0	1	0	0
1	1	0	1	0

Fig. 1. A 2-way covering array with 5 binary options.

2.2 Cost-aware covering arrays

Standard CAs (and the tools generating them) aim to reduce the actual testing cost by minimizing the size of the CA. They assume that the testing cost of all the configurations are the same. Unlike standard CAs, cost-aware covering arrays take actual cost of testing into account while computing the covering arrays. Consequently, cost-aware covering arrays take as input a standard configuration space model $M=<O,V>$ augmented with a cost function $cost(.)$ [13, 12].

Definition 4. *The cost function $cost(c)$ computes the expected cost of a given configuration c as follows:*

$$cost(c) = intercept + \sum_{\phi_1 \in \Phi_1} cost(\phi_1) + \sum_{\phi_2 \in \Phi_2} cost(\phi_2) + \ldots + \sum_{\phi_f \in \Phi_f} cost(\phi_f)$$

where $\phi_m \in \Phi_m$ is a **costly** *m-tuple in c that increases the cost of the configuration with an additional cost ($cost(\phi_m) > 0$) such that $1 \leq m \leq f \leq k$. The intercept is the base cost of the configuration in the absence of any costly tuples ($intercept > 0$).*

Definition 5. *The cost of a covering array, $CA=\{c_1, c_2, \ldots, c_N\}$ is the sum of the costs of all configurations in the CA, i.e.,*

$$cost(CA) = cost(\{c_1, c_2, \ldots, c_N\}) = cost(c_1) + cost(c_2) + \ldots + cost(c_N)$$

Definition 6. *Given a configuration space model $M=<O,V,cost(.)>$, a t-way cost-aware covering array CCA is a t-way CA with the "minimal" $cost(CA)$. The lower bound for the cost of a CA with N configurations is $intercept \times N$.*

We manipulated four independent variables that can affect the cost model:

1. **Cardinality of costly tuples** (f in Definition 4) is the number of option-value pairs in each costly tuple for that model. For $f > t$ such costly tuples can be avoided and not added to the CA at all. However, for $f \leq t$, as each *t-tuple* will be added to the CA, such costly tuples will also be included, increasing the cost of the resulting CA.
2. **Number of costly tuples (b)** represents how many tuples increase the cost of a configuration if included in it. For a given configuration, $b=|\Phi_1| + |\Phi_2| + \ldots + |\Phi_f|$ (see Definition 4). The cost of a configuration, hence the cost of a CA, is expected to be higher for larger b values.
3. **Impact of costly tuples (i)** represents the percentage of how much a costly tuple increases the base cost if included in the configuration. If $i \leq 100\%$, decreasing the size of the CA may also be beneficial. For i=0, the minimum cost can be achieved by generating a CA with the minimum number of rows. However for $i \geq 100\%$, a less costly CA may be achieved by increasing the size of the CA and avoiding such costly tuples.
4. **Percentage of costly options (c)** for a given cost model determines how many distinct options will appear in the costly tuples of that cost model. For example, for a system with 25 distinct options, c=20% means that the costly tuples will only contain combinations of 5 different options.

Below, an example cost model is given for a configuration c in a configuration space with binary options, $intercept$=100, b=3, f=2 and i=50%.

$$cost(c) = 100+(o_2 = 1 \wedge o_3 = 1)50+(o_3 = 1 \wedge o_6 = 0)50+(o_7 = 0 \wedge o_9 = 0)50 \quad (1)$$

Since it is generally hard and impractical for the experts of the system to express the cost in terms of costly tuples [2], we have been also working on automated approaches for cost model discovery [18].

2.3 Some existing CA-generation tools

As CIT is getting used more widely in practical cases, several tools have been developed to construct CAs effectively [16] including Pairwiser, a tool that utilizes parallelization [17]. We investigate and make comparisons with two well known tools: ACTS [19] and Jenny [15].

ACTS can generate CAs with strengths 2-way through 6-way and also supports constraints and variable-strength tests. `Jenny` is another well known, open-source tool for CA generation which also supports constraints and variable-strength CAs. It takes the configurable options, the strength t, and the constraints as input and produces a CA as the output. Experiments on `Jenny` showed that its average CA size is acceptable for many cases, and it can be faster than many state-of-art tools especially when many constraints exist.

`Jenny` uses a greedy, iterative approach; at each iteration, a set of τ configurations are generated by consecutively setting each option to a value in a greedy fashion. In other words, it uses a hillclimbing approach to generate τ configurations. Then, the configuration with the maximum *coverage* (the number of additional tuples that may be covered by this configuration) among these random configurations is chosen and added to current state of CA until no tuples left uncovered. A high level pseudocode for Jenny is given in Algorithm 1.

We implement cost-awareness at multiple steps within `Jenny` to generate cost-aware CAs. We also parallelize `Jenny` to generate CAs in much shorter time.

3 Cost-awareness for free

The current CA generation tools try to reduce the testing time by reducing the size of the covering array, i.e., the number of its configurations. However, in most cases, the costs of testing different configurations are not the same, so minimizing the size does not necessarily minimize the cost. We have modified `Jenny` to minimize the actual cost instead of the CA size by making it cost-aware.

The original `Jenny` algorithm (Algorithm 1) randomly generates τ candidate tests. For each test, it tunes each option-value pair such that their coverage counts are maximized and finally selects one of them according to their overall coverage counts. In our modified version, however, after generating τ candidate configurations, the configuration with the lowest *cost* is selected instead of the configuration with the highest *coverage* count. Moreover, while generating τ candidate tests, each column is tuned such that the cost is decreased instead of increasing the coverage count. The cost-aware algorithm is given in Algorithm 2.

Algorithm 1 JENNY

Input: S: configuration space, t: strength
Output: CA(S): an N x k covering array

```
1: N ← 0
2: CA ← ∅ (an empty CA with k columns)
3: while true do
4:     tuple ← SelectUncoveredTuple(CA, S)
5:     bestCoverage ← −1
6:     for i = 1 to τ do
7:         (test, coverage) ← GenTest(S, tuple)
8:         if coverage > bestCoverage then
9:             bestTest ← test
10:            bestCoverage ← coverage
11:    N ← N + 1
12:    CA(N, :) ← bestTest
13:    if CountUncoveredTuples(S) = 0 then
14:        break
```

Algorithm 2 COST-AWARE JENNY

Input: S: configuration space, t: strength
Output: CA(S): an N x k covering array

```
1: N ← 0
2: CA ← ∅ (an empty CA with k columns)
3: while true do
4:     tuple ← SelectUncoveredTuple(CA, S)
5:     bestCost ← INT_MAX
6:     bestCoverage ← −1
7:     for i = 1 to τ do
8:         test ← GenTest(S, tuple)
9:         (test, cost, coverage) ← ImproveConfiguration(S, test)
10:        if cost < bestCost then
11:            bestTest ← test
12:            bestCost ← cost
13:            bestCoverage ← coverage
14:        else if cost = bestCost then
15:            if coverage > bestCoverage then
16:                bestTest ← test
17:                bestCost ← cost
18:                bestCoverage ← coverage
19:    N ← N + 1
20:    CA(N, :) ← bestTest
21:    if CountUncoveredTuples(S) = 0 then
22:        break
```

In a single iteration, the SelectUncoveredTuple function selects an uncovered tuple (line 4). After selecting one, the GenTest function is called for τ times to generate random configurations that cover the selected tuple. The configuration with the lowest cost is selected (line 10) and added to the CA (line 20). If the costs of two configurations are the same, then their coverages are compared and the configuration which covers more additional tuples is selected (line 15). If all tuples are covered after an iteration then the algorithm stops.

Although this approach decreases the costs considerably, it still leaves room for improvement. By decreasing the costs of candidate configurations, we can obtain configurations with lower costs to choose from, which would further reduce the cost. In Algorithm 2 we use the GenTest function (line 8) to generate τ candidate tests. This function places a tuple to be covered inside a configuration and fills the rest of the configuration randomly. After that ImproveConfiguration function (line 9) is called on this configuration to reduce the cost of the given configuration and increase the coverage. This is accomplished by using a hill climbing approach as shown in Algorithm 3.

The ImproveConfiguration function takes a configuration (test) as input and iterates through all the options (line 1). At each iteration, the corresponding option is set to one of its possible values (line 6) and if the cost decreases and coverage increases the change is saved (line 10). If multiple values decrease the cost or increase the coverage, one of them is accepted randomly (line 12). The CountTuples and Cost functions compute the additionally covered tuples and the additional cost caused by the updated test at each iteration.

3.1 Generating Covering Arrays in Parallel

As stated before, generating a CA efficiently is important to start the testing process earlier. However, the CA generation may take hours for the systems with a large number of configurable options. Although even commodity processors have multiple cores and are capable of parallel processing, surprisingly, most of the state-of-the art tools do not fully exploit the computation power on these CPUs. We use coarse-grain parallelism on the modified, cost-aware Jenny to analyze the benefits of parallel computing on CA generation in detail.

We carefully profiled `Jenny` with different inputs and as expected, we observed that the bottleneck, i.e. most time consuming part, for the Cost-Aware Jenny algorithm is the test generation part (lines 7-18 of Algorithm 2), in which τ random tests are generated and the test with the least cost is selected. In fact, on average 84% and 89% of the total time is spent on this part for t=2 and t=3, respectively. This process is pleasingly parallel since the generation of each test is independent from the generation of others. Hence, we followed a coarse-grain parallel approach to concurrently generate τ tests. Unlike the original algorithm, we store each generated test, their cost and coverage values until the generation phase ends. Then, we choose the best test among these. The pseudocode of the parallel Cost-Aware Jenny algorithm is given in Algorithm 4.

This parallelization approach allows us to utilize n threads where $n \leq \tau$, effectively. Since the targeted bottleneck occupies 84% and 89% of the total

Algorithm 3 ImproveConfiguration

Input: S: configuration space, *test*: test, k: number of options
Output: t: test with minimal cost

```
1: for i = 0 to k do
2:     cost ← Cost(test)
3:     cov ← CountTuples(test)
4:     n ← 0
5:     for j = 0 to values[i] do
6:         test[i] ← j
7:         nCost ← Cost(test)
8:         nCov ← CountTuples(test)
9:         if nCost ≤ cost and nCov ≥ cov then
10:            best[n] ← j
11:            n ← n + 1
12:    t[i] ← best[rand(0, n)]
```

Algorithm 4 Parallel Cost-Aware Jenny

Input: S: configuration space, t: strength
Output: CA(S): an N x k covering array

```
1: N ← 0
2: CA ← ∅ (an empty CA with k columns)
3: costs[.] ← an integer array of size τ
4: covs[.] ← an integer array of size τ
5: tests[.] ← a test array of size τ
6: while true do
7:     tuple ← SelectUncoveredTuple(S)
8:     for i = 1 to τ in parallel do
9:         tests[i] ← GenTest(S, tuple)
10:        (tests[i], covs[i], costs[i]) ← ImproveConfiguration(S, tests[i])
11:    bestCoverage ← −1
12:    bestCost ← INT_MAX
13:    for i = 1 to τ do
14:        if costs[i] < bestCost then
15:            bestTest ← tests[i]
16:            bestCost ← costs[i]
17:            bestCoverage ← coverages[i]
18:        else if costs[i] = bestCost then
19:            if coverages[i] > bestCoverage then
20:                bestTest ← tests[i]
21:                bestCost ← costs[i]
22:                bestCoverage ← coverages[i]
23:    N ← N + 1
24:    CA(N, :) ← bestTest
25:    if CountUncoveredTuples(S) = 0 then
26:        break
```

time for t=2 and t=3 respectively, the maximum speed up that can be observed can be calculated as in Equation 2.

$$t = 2 : \frac{1}{0.16 + \frac{0.84}{n}} \qquad t = 3 : \frac{1}{0.11 + \frac{0.89}{n}} \tag{2}$$

This value is 1.8 for n=2, 3 for n=4 and 4.5 for n=8 when t=2 and it's 1.7 for n=2, 2.7 for n=4 and 3.7 for n=8 when t=3. Note that these are the maximum speed up values which assume that there are no parallelization overheads.

4 Experimental Results

In the CA generation problem, generating CAs both effectively and efficiently is important. The former is crucial for keeping the testing time short, whereas the latter would allow starting the testing process earlier. We have carried out two sets of experiments evaluating both the effectiveness and efficiency of our construction approach. First, we focus on minimizing the actual testing time by reducing the cost of CAs. Then, we aim to shorten the CA generation time by exploiting the computation power of multicore processors.

We formed configuration spaces for the experiments using strength: $t \in \{2, 3\}$, number of options: $k \in \{25, 35, 45, 55, 65, 85, 100\}$ and impact of costly tuples: $i \in \{50\%, 100\%\}$. Additionally, for each configuration space, all the performance and cost results presented in the following subsections are calculated as the arithmetic means of 72 experiments: 3 different number of costly tuples $\{4, 5, 6\}$, 4 different cardinality of costly tuples $\{1, 2, 3, 4\}$, 2 different percentage of costly options $\{50\%, 100\%\}$, and 3 executions of the same problem with different random seeds. Moreover, for `Jenny` and `Cost-Aware Jenny`, $n \in \{1, 2, 4, 8\}$ threads are used for parallelism. Thus, a total of over 15K experiments have been carried out.

The experiments are performed on a machine running on 64 bit CentOS 6.5 with an Intel Xeon E7-4870 v2 clocked at 2.30 GHz. The codes are compiled with `gcc 4.4.7` with `-O3` flag and `OpenMP 4.0` is used for parallelism.

4.1 Generating Cost-Aware Covering Arrays

Testing a software system in a short amount of time is crucial as it would allow shipping the product earlier. Therefore, in order to reduce the actual cost of testing, we aim to generate low-cost CAs for given configuration spaces. Our earliest work [11] to compute CCAs was a greedy algorithm for one simple type of cost function, where the goal was minimizing the number of unique compile-time configurations included in the CA. In this work as well as our recent heuristic approach to compute CCAs [12], the cost function is more general modeling the cost at the level of option-value combinations. This work is a parallel greedy approach whereas previous work [12] was a simulated annealing based approach.

We conducted several experiments to generate CAs with `ACTS`, `Jenny` and `Cost-Aware Jenny` on our configuration spaces. Note that both `ACTS` and `Jenny` are not cost-aware. Then, we calculated the cost of each generated CA. They are presented in Table 1 and Table 2 for i=100% and i=50%, respectively. We

also give the cost reduction percentages of our approach compared to the best result of either `Jenny` or `ACTS` for each case.

Even in the worst case scenarios, `Cost-Aware Jenny` generates 21% and 9% lower cost CAs for t=2 and t=3, respectively. Moreover, `Cost-Aware Jenny` can generate 32% and 21% lower cost CAs on average for t=2 and t=3, respectively compared to the best result of either `Jenny` or `ACTS`. We can also state that as the impact of costly tuples i increases, the percentage of cost reduction increases.

Table 1. Average costs for ACTS, Jenny and Cost-Aware Jenny for i=100%.

	t=2				**t=3**			
k	ACTS	JENNY	CAJENNY	Cost reduction (%)	ACTS	JENNY	CAJENNY	Cost reduction (%)
25	7.97	6.88	4.13	39.97	24.09	27.12	19.06	20.88
35	8.23	7.04	4.41	37.36	28.96	32.64	21.67	25.17
45	8.56	7.60	4.69	38.29	32.33	34.46	23.70	26.69
55	9.29	8.39	4.94	41.12	36.04	39.04	25.62	28.91
65	10,01	8.02	5.18	35.41	38.91	40.97	27.43	29.50
85	10.85	9.37	5.50	41.30	44.12	47.81	31.21	29.26
100	11.21	9.74	5.73	41.17	47.42	50.29	33.20	29.99

Table 2. Average costs for ACTS, Jenny and Cost-Aware Jenny for i=50%.

	t=2				**t=3**			
	ACTS	JENNY	CAJENNY	Cost reduction (%)	ACTS	JENNY	CAJENNY	Cost reduction (%)
25	4.91	5.11	3.66	25.46	18.32	20.57	16.65	9.12
35	5.63	5.28	4.02	23.86	22.06	24.44	19.23	12.83
45	5.92	5.67	4.26	24.87	24.46	26.27	20.94	14.39
55	6.39	6.38	4.52	29.15	27.12	29.45	22.72	16.22
65	6.86	5.97	4.69	21.44	29.36	30.80	24.22	17.51
85	7.46	6.59	4.97	24.58	33.16	34.65	26.84	19.06
100	7.73	7.04	5.19	26.28	35.52	36.56	28.39	20.07

4.2 Generating covering arrays faster

Generating a CA fast is important to start the testing process earlier. Hence, most of the state-of-the-art tools we have today aim to generate CAs efficiently, i.e., in a short amount of time. However, surprisingly, parallel programming is not used in most of these tools, despite its increasing popularity.

In this set of experiments we use parallel programming on multicore processors to reduce the CA generation time, without increasing the costs of resulting CAs. We use τ=8 to avoid load-imbalances, or in another words to distribute the workload equally among the processors. As mentioned in Algorithm 4, we have parallelized the step where τ candidate tests are generated. Since this is a pleasingly parallelizable step, we aim to have speed-up values close to the corresponding possible maximum values, calculated in Section 3.1.

The execution times for original `Jenny` and `Parallel Cost-Aware Jenny` is given in Table 3 where n is the number of threads used for parallelization.

Table 3. Execution Times for Parallel Cost-Aware Jenny for t=2 (ms) and t=3 (sec).

	t=2 (ms)					t=3 (sec)				
		Parallel CA Jenny					Parallel CA Jenny			
k	Jenny	$n=1$	$n=2$	$n=4$	$n=8$	Jenny	$n=1$	$n=2$	$n=4$	$n=8$
25	13	14	10	9	8	0.40	0.38	0.23	0.15	0.11
35	22	24	16	12	11	1.43	1.35	0.79	0.50	0.36
45	32	34	22	16	15	4.18	3.55	2.07	1.32	0.96
55	48	49	31	22	19	9.55	8.44	5.03	3.17	2.36
65	62	66	41	29	24	21.30	19.37	11.79	7.87	5.82
85	102	106	63	42	33	136.41	119.30	76.95	52.35	42.58
100	139	142	85	56	44	404.2	624.1	390.5	280.4	218.9

When $n = 1$, in almost every case, `Jenny` generates CAs faster or `Cost-Aware Jenny` performs slightly better than `Jenny`. However, when we increase the number of threads, `Cost-Aware Jenny` surpass the `Jenny` in the execution time with 2.45 and 3.53 speedups on average for t=2 and t=3, respectively. Based on these results, one can conclude that as the number of threads increase, for both t=2 and t=3, the execution times decrease. Therefore, even though we add new functionality to reduce the actual testing cost with the help of parallel computing, the execution time of generating the CAs decreased as well, i.e., it comes for free.

5 Conclusion and Future Work

In this paper, we showed that generation of a cost-aware CA [11, 13] is not necessarily more costly than generating a standard covering array. Moreover, we showed that parallelization can be a great asset for CA generation process, reducing the time spent on CA generation. Both properties are very important for CA construction, since for systems with high testing costs, generating less costly CAs is desirable whereas for systems with low testing costs generating CAs faster is crucial. Our experiments showed that with careful profiling and coarse-grain parallelism, one can obtain significant speedups without changing the CA generation tools much, which are mostly sequential.

As a future work, we believe that there is still some room for improvement in terms of cost reduction. In the current version, the tuples to be covered are enumerated and selected randomly. However, we believe that if costly tuples are enumerated and covered in the earlier phases of the algorithm, it will decrease the cost as such tuples will not be added to the CA anymore. Another approach could be treating costly tuples as constraints, i.e., avoiding them while generating a CA, then covering such tuples with additional rows at the end. We also plan to investigate other approaches such as SAT solvers and branch-and-bound algorithms to compute cost-aware covering arrays.

References

1. D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr, "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*,

vol. 30, no. 6, pp. 418–421, 2004.
2. C. Yilmaz, S. Fouche, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc, "Moving forward with combinatorial interaction testing," *Computer*, no. 2, pp. 37–45, 2014.
3. C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.
4. G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *Information Theory, IEEE Transactions on*, vol. 34, no. 3, pp. 513–522, 1988.
5. Y. Lei and K.-C. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*. IEEE, 1998, pp. 254–261.
6. M. B. Cohen, C. J. Colbourn, and A. C. Ling, "Augmenting simulated annealing to build interaction test suites," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 2003, pp. 394–405.
7. J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, 2006, pp. 419–430.
8. Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog/ipog-d: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.
9. B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.
10. S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 323–334.
11. G. Demiroz and C. Yilmaz, "Cost-aware combinatorial interaction testing," in *In the Proceedings of VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, November 2012, pp. 9–16.
12. G. Demiroz and C. Yilmaz, "Using simulated annealing for computing cost-aware covering arrays," *Applied Soft Computing*, vol. 49, pp. 1129–1144, December 2016.
13. G. Demiroz, "Cost-aware combinatorial interaction testing (doctoral symposium)," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, July 2015, pp. 440–443.
14. R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006, advances in Model-based Testing.
15. B. Jenkins, "jenny: A pairwise testing tool," http://www.burtleburtle.net/bob/index.html, 2005.
16. "Pairwise testing available tools," http://www.pairwise.org/tools.asp.
17. M. F. Johansen, O. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference, SPLC '12*, vol. 1. ACM, 2012, pp. 46–55.
18. G. Demiroz and C. Yilmaz, "Towards automatic cost model discovery for combinatorial interaction testing," in *Proceedings of the 2016 International Workshop on Combinatorial Testing (IWCT 2016)*. IEEE, April 2016.
19. L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 370–375.