Test Prioritization for Pairwise Interaction Coverage

Renée C. Bryce and Charles J. Colbourn
Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-8809
{rcbryce,colbourn}@asu.edu

ABSTRACT

Interaction testing is widely used in screening for faults. In software testing, it provides a natural mechanism for testing systems to be deployed on a variety of hardware and software configurations. Several algorithms published in the literature are used as tools to automatically generate these test suites; AETG is a well known example of a family of greedy algorithms that generate one test at a time. In many applications where interaction testing is needed, the entire test suite is not run as a result of time or cost constraints. In these situations, it is essential to *prioritize* the tests. Here we adapt a "one-test-at-a-time" greedy method to take importance of pairs into account. The method can be used to generate a set of tests in order, so that when run to completion all pairwise interactions are tested, but when terminated after any intermediate number of tests, those deemed most important are tested. Computational results on the method are reported.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Biased covering arrays, covering arrays, greedy algorithm, mixed-level covering arrays, pairwise interaction coverage, software interaction testing

1. INTRODUCTION

Software testing is an expensive and time consuming activity that is often restricted by limited project budgets. Accordingly, the National Institute for Standards and Technology (NIST) reports that software defects cost the U.S. economy close to $60 billion a year [28]. They suggest that approximately $22 billion can be saved through more effective testing. There is a need for advanced software testing techniques that offer a solid cost-benefit ratio in identifying defects. Interaction testing is one such method that may offer a benefit when used to complement current testing methods [11, 14]. Interaction testing implements a model based testing approach using combinatorial design. In this approach, all t-tuples of interactions in a system are incorporated into a test suite. This testing method has been applied in numerous examples including [3, 2, 4, 14, 15, 25, 26, 27, 34, 37]

The need for prioritization arises in many different phases of the software testing process [17, 19, 20, 21, 30, 31, 36]. For instance, requirements need to be prioritized; tasks on schedules need to prioritized; and testers may have to prioritize the tests that they run.

A prioritized interaction test suite may be particularly useful to testers who want to test key areas of concern first, or for those that wish to run their tests in order of importance until they exhaust their available resources. For instance, a tester may request a complete test suite and attempt to run as many tests as they can budget. In this context, it is important to test the most important items first. An example in testing web services is described in [13]. In order to construct these test suites, biased covering arrays are proposed here.

This paper is organized as follows: Section 2 presents definitions and background. Section 3 describes a method for constructing biased covering arrays. Section 4 provides initial computational results.

2. COVERING ARRAYS

2.1 Definitions and Background

A *covering array*, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array. In every $N \times t$ subarray, each t-tuple occurs at least λ times. In our application, t is the *strength* of the coverage of interactions, k is the number of factors (degree), and v is the number of symbols for each factor (order). We treat only the case when $\lambda = 1$, i.e. that every t-tuple must be covered at least once.

This combinatorial object is fundamental when all factors have an equal number of levels. However, software systems are typically not composed of components *(factors)* that each have exactly the same number of parameters *(levels)*. Then the mixed-level covering array can be used.

A *mixed level covering array*, $MCA_\lambda(N; t, k, (v_1, \ldots, v_k))$,

```
1.  start with empty test suite
2.  while uncovered pairs remain do
3.      for each factor
4.          compute factor interaction weights
5.          initialize new test with all factors not fixed
6.          while a factor remains whose value is not fixed
7.              select such a factor $f$ that has the largest factor interaction weight,
8.                  using a factor tie-breaking rule
9.              compute factor-level interaction weights for each level of factor $f$
10.             select a level $\ell$ for $f$ which offers the largest increase in weighted density
11.                 using a level tie-breaking rule
12.             fix factor $f$ to level $\ell$
13.         end while
14.     add test to test suite
15. end while
```

Figure 1: Pseudocode for a greedy algorithm to generate biased covering arrays

is an $N \times k$ array. Let $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$, and consider the subarray of size $N \times t$ obtained by selecting columns $i_1, \ldots, i_t$ of the MCA. There are $\prod_{i=1}^{t} v_i$ distinct t-tuples that could appear as rows, and an MCA requires that each appear at least once.

Some interactions are more important to cover than are others; the covering array does not distinguish this, as it assumes that all rows will be run as tests. When only some rows are to be used as tests, certain tests are more desirable than others.

Call the factors $f_1, \ldots, f_k$. Suppose that, for each i, f_i has ℓ_i possible values $c_{i,1}, \ldots, c_{i,\ell_i}$. For each $c_{i,j}$, we assume that a numerical value between 0 and 1 is available as a measure of importance, with 1 as most important and 0 as unimportant. Every value τ for f_i has an importance $t_{i,\tau}$. A *test* consists of an assignment to each factor f_i of a value τ_i with $1 \leq \tau_i \leq \ell_i$. The first task is to quantify the preference among the possible tests. In order to capture important interactions among *pairs* of choices, importance of pairs is defined by a weight which can take on values of 0 to 1, where 1 is the strongest weight. Specifically, the importance of choosing τ_i for f_i and τ_j for f_j together is $t_{i,\tau_i} t_{j,\tau_j}$.

The *benefit* of a test (in isolation) is $\sum_{i=1}^{k} \sum_{j=i+1}^{k} t_{i,\tau_i} t_{j,\tau_j}$. Every pair covered by the test contributes to the total benefit, according to the importance of the selections for the two values. However in general we are prepared to run many tests. Then rather than adding the benefits of each test in the suite, we must account a benefit only when a pair of selections has not been treated in another test. Let us make this precise. Each of the pairs (τ_i, τ_j) covered in a test of the test suite may be covered for the first time by this test, or can have been covered by an earlier test as well. Its *incremental benefit* is $t_{i,\tau_i} t_{j,\tau_j}$ in the first case, and zero in the second. Then the incremental benefit of the test is the sum of the incremental benefits of the pairs that it contains.

The total benefit of a test suite is the sum, over all tests in the suite, of the incremental benefit of the test.

A *ℓ-biased covering array* is a covering array $CA(N; 2, k, v)$ in which the first ℓ rows form tests whose total benefit is as large as possible. That is, no $CA(N'; 2, k, v)$ has ℓ rows that provide larger total benefit.

Although precise, this definition should be seen as a goal rather than a requirement. Finding an ℓ-biased covering array is NP-hard, even when all benefits for pairs are equal [12]. Worse yet, the value of ℓ is rarely known in advance. For these reasons, we use the term *biased covering array* to mean a covering array in which the tests are ordered, and for every ℓ, the first ℓ tests yield a "large" total benefit.

Covering arrays have been extensively studied [11]. Specifically, techniques have been introduced in the areas of combinatorial constructions [5, 16, 22, 23, 35], heuristic search [9, 10, 29], and greedy algorithms [1, 6, 7, 8, 12, 32, 33]. The overriding criteria for evaluation of these techniques has been the size of the test suites and the execution time for such constructions. Further consideration has been given to other practical concerns such as permitting *seeds*, or user specified tests, and blocking *constraints* of combinations that are not permitted. Recently, *prioritized ordering* of tests [13] has also been considered. The concept is straightforward – testers may have priorities that they assign to different levels for factors. The inclusion of the highest importance levels should occur as early as possible in a test suite. The algorithm presented here constructs pair-wise covering arrays while covering the most important values early. This work may be extended to t-way coverage.

3. THE ALGORITHM

We consider the construction of a test suite with k factors, adapting the Deterministic Density Algorithm [12]. For factors i and j, the *total benefit* β_{ij} is $\sum_{a=1}^{\ell_i} \sum_{b=1}^{\ell_j} t_{i,a} t_{j,b}$, while the *remaining benefit* ρ_{ij} is the same sum, but of the incremental benefits. Define the *local density* to be $\delta_{i,j} = \frac{\rho_{i,j}}{\beta_{ij}}$. In essence, $\delta_{i,j}$ indicates the fraction of benefit that remains available to accumulate in tests to be selected. We define the *global density* to be $\delta = \sum_{1 \leq i < j \leq k} \delta_{i,j}$. At each stage, we endeavor to find a test whose incremental benefit is at least δ.

To select such a test, we repeatedly fix a value for each factor, and update the local and global density values. At each stage, some factors are *fixed* to a specific value, while others remain *free* to take on any of the possible values. When all factors are fixed, we have succeeded in choosing the next test. Otherwise, select a free factor f_s. We have

$\delta = \sum_{1\le i<j\le k} \delta_{i,j}$, which we separate into two terms:

$$\delta = \sum_{\substack{1\le i<j\le k\\ i,j\ne s}} \delta_{i,j} + \sum_{\substack{1\le i\le k\\ i\ne s}} \delta_{i,s}.$$

Whatever level is selected for factor f_s, the first summation is not affected, so we focus on the second.

Write $\rho_{i,s,\sigma}$ for the ratio of the sum of incremental benefits of those pairs involving some level of factor f_i, and level σ of factor f_s to the sum of (usual) benefits of the same set of pairs. Then rewrite the second summation as

$$\sum_{\substack{1\le i\le k\\ i\ne s}} \delta_{i,s} = \frac{1}{\ell_s}\sum_{\sigma=1}^{\ell_s}\sum_{\substack{1\le i\le k\\ i\ne s}} \rho_{i,s,\sigma}.$$

We choose σ to maximize $\sum_{\substack{1\le i\le k\\ i\ne s}} \rho_{i,s,\sigma}$. It follows that $\sum_{\substack{1\le i\le k\\ i\ne s}} \rho_{i,s,\sigma} \ge \sum_{\substack{1\le i\le k\\ i\ne s}} \delta_{i,s}$. We then fix factor f_s to have value σ, set $v_s = 1$, and update the local densities setting $\delta_{i,s}$ to be $\rho_{i,s,\sigma}$. In the process, the density has not been decreased (despite some possible – indeed necessary – decreases in some local densities).

We iterate this process until every factor is fixed. The factors could be fixed in *any order at all*, and the final test has density at least δ. Of course it is possible to be greedy in the order in which factors are fixed.

This method ensures that each test selected furnishes at least the *average* incremental benefit. This may seem to be a modest goal, and that one should instead select the test with maximum incremental benefit. However, even when all importance values are equal, it is NP-hard to select such a test (see [12]).

Next we illustrate the operation of the method, before reporting experimental results.

3.1 Algorithm Walk-through

The overall goal in constructing a covering array is to create a 2-dimensional array in which all t-tuples are covered. The secondary constraint is that as much weight be incorporated into the test suite as early as possible. In our implementation, each level for every factor is assigned a weight value between 0 and 1, where 1 is the highest importance.

Using a greedy approach (see Figure 1), this collection is built one row at a time by fixing each *factor* with a *level* (value). A *test* consists of an assignment to each factor f_i of a value τ_i with $1 \le \tau_i \le \ell_i$, where ℓ_i is the number of levels (or values) associated with a factor. The order in which factors are assigned values is based on factor interaction weights. Levels are selected based on a weighted density formula that calculates a likelihood of covering as much uncovered weight as possible for a row. Once all t-tuples have been covered, the covering array is complete.

An example of building a row

Consider the input in Table 1 with three factors. The first factor (f_0) has four levels, the second (f_1) has three levels, and the third (f_2) has two levels. Each level value is labelled in Table 1 with a unique ID, and a weight for each in parenthesis.

Step 1 - Identify the order to assign levels to factors

To construct a row in the biased covering array, factors are assigned values one at a time in order of decreasing factor interaction weights. Factors that have been assigned values are *fixed* while those that have not been assigned values are called *free*. Factor interaction weights are calculated between two factors by multiplying the weights between all levels of the two factors. The maximum weight is denoted as w_{max}.

Factor	v_0	v_1	v_2	v_3
f_0	0 (.2)	1 (.1)	2 (.1)	3 (.1)
f_1	4 (.2)	5 (.3)	6 (.3)	-
f_2	7 (.1)	8 (.9)	-	-

Table 1: Input of three factors and their levels and weights

Factor Interaction Weight	f_0	f_1	f_2	**Total Weight**
f_0	-	.4	.5	.9
f_1	.4	-	.8	1.2
f_2	.5	.8	-	1.3

Table 2: Factor interaction weights

Table 2 shows the factor interaction weights for the input from Table 1. The algorithm assigns levels for factors in decreasing order of their factor interaction weights: f_2, f_1, and f_0.

Step 2 - Assign levels for each factor

To select a value for f_2, either of its two levels v_0 or v_1 may be selected. The first level, v_0, has a value of 7 in Table 1, while the second (v_1) has a value of 8. According to the level selection criteria, the one with the largest factor-level interaction weight is chosen. There are two scenarios to calculate the weighted density for level selection. If a level is being selected in relation to a factor that has already been fixed, then the contributing weight of selecting the new level is 0.0 if no new weight is covered; otherwise the contributing weight is the product of the current level under evaluation's weight and the fixed level's weight. However, when selecting a level value in relation to factors that have not yet been assigned values, weighted density is calculated as the factor-level interaction weight. Factor-level interaction weight is calculated for a level, ℓ, and a factor i that has a number of levels called v_{max}, as: $\sum_{j=1}^{v_{max}}(\frac{w_{f_{i_j}} * w_\ell}{w_{max}})$,

For instance, the factor-level interaction weights used to select a value for f_2 are:

$f_{2_{v_0}}$ = (.05/1.3)+(.08/1.3) = .1
$f_{2_{v_1}}$ = (.45/1.3)+(.72/1.3) = .9

Since the second level has a larger factor-level interaction weight, the level v_1 (8) is selected for factor f_2.

The second factor to receive a value is f_1 since it has the second largest factor interaction weight. Factor f_1 has three levels to choose from. For each of these levels, weighted density is calculated in relation to the other factors. Since f_2 has already been fixed with a value, density is increased by the product of the weight of f_2's fixed level and the weight of the level being evaluated for selection. Factor f_0 on the other hand has not yet been fixed with a level so weighted density is increased by the formula that includes the likelihood of covering weight with this factor in the future.

$f_{1_{v_0}}$ = (.1/1.3)+(.2*.9) = .2569
$f_{1_{v_1}}$ = (.15/1.3)+(.3*.9) = .38538

Row Number	f_0	f_1	f_2
1	0	5	8
2	1	6	8
3	2	4	8
4	3	5	8
5	0	6	7
6	0	4	7
7	1	5	7
8	2	5	7
9	3	6	7
10	2	6	7
11	1	4	7
12	3	4	7

Table 3: Output of the prioritized greedy algorithm for the case study example

Row Number	Unweighted Density	Weighted Density
1	4.71%	30.00%
2	4.12%	22.94%
3	4.12%	17.06%
4	17.06%	7.06%
5	30.00%	6.47%
6	22.94%	3.53%
7	6.47%	4.12%
8	2.35%	2.35%
9	3.53%	2.35%
10	1.18%	1.76%
11	1.76%	1.18%
12	1.76%	1.18%

Table 4: Weight covered in each row using Unweighted and Weighted density

$f_{1_{v_2}} = (.15/1.3)+(.3*.9) = .38538$

In this case, there is a tie between v_1 and v_2, broken by taking the lexicographically first, v_1.

Finally, the third factor to fix is f_0. The weighted density for f_0's levels is straightforward as it is the increase of weight offered in relation to the other fixed factors.

$f_{0_{v_0}} = (.2*.3)+(.2*.9) = .24$
$f_{0_{v_1}} = (.1*.3)+(.1*.9) = .12$
$f_{0_{v_2}} = (.1*.3)+(.1*.9) = .12$
$f_{0_{v_3}} = (.1*.3)+(.1*.9) = .12$

Since v_0 offers the largest increase in weight, it is selected as the level for f_0 in this test.

The generation of this one row demonstrates each type of decision made in this algorithm for ordering factors and assigning levels to factors. When the algorithm is given the opportunity to complete, the output is shown in Table 3.

4. EMPIRICAL RESULTS

The weighted density algorithm generates test suites that order rows in decreasing order of importance. Two algorithms are compared using the data in Table 1. The first utilizes unweighted density, and the second uses weighted density to construct a biased covering array. The amount of weight covered in each row is shown in Table 4. Weighted

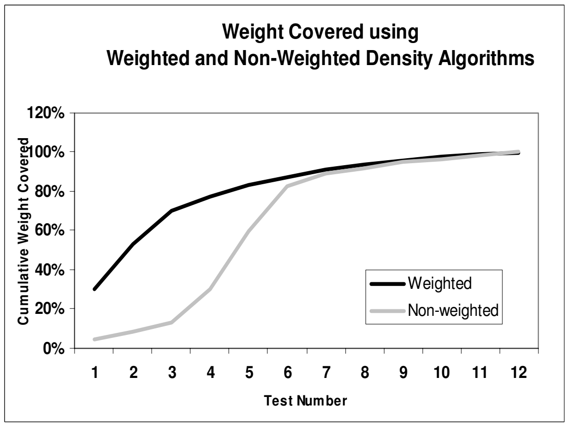

Figure 2: Cumulative weight covered using Unweighted and Weighted density

density covers more weight earlier. Figure 2 shows the difference in cumulative weight covered after each row.

Initially, one may have thought that the prioritization would result in a significant increase in the number of tests. For instance, in one example we ran 7^8 with half of the factors assigned the highest priority and the other half a low priority. This resulted in 85 tests as opposed to 77 when everything was weighted equally. Remarkably, though, the example of unequal weights in Table 1 did not adversely affect the overall size of the test suite. We explore this further, considering several schemes for assigning weights.

Consider the input $20^2 10^2 3^{100}$ (this is a shorthand for 2 factors of 20 values each, 2 of 10 each, and 100 of 3 each), with four different weight distributions:

- **Distribution 1 (Equal weights)**- All levels have the same weight,
- **Distribution 2 ($\frac{50}{50}$ split)**- Half of the weights for each factor are set to .9 and the other half to .1,
- **Distribution 3 ($(\frac{1}{v_{max}})^2$ split)**- All weights of levels for a factor are equal to $(\frac{1}{v_{max}})^2$, where v_{max} is the number of levels associated with the factor,
- **Distribution 4 (Random)**- Weights are randomly distributed

The rate of cumulative weight coverage for an input differs depending on the associated weight distribution. Figure 3 shows the percentage of cumulative weight covered in the first 20 tests for each of the four examples of different distributions. When all weights are equal (Distribution 1), the result is a (non-biased) covering array. This exhibits the slowest growth of weight coverage early on. However, when there is more of a difference in the distribution of weights, a biased covering array can often move more weight to occur in the earliest rows.

For instance, the $\frac{50}{50}$ split shows the most rapid coverage of growth earliest. This may be expected because half of the levels with a weight of .9 comprise the majority of the weight and are quickly covered in the early rows. The $\frac{1}{v_{max}}^2$ split and Random are similar to each other, and intermediate between the two other two extremes considered.

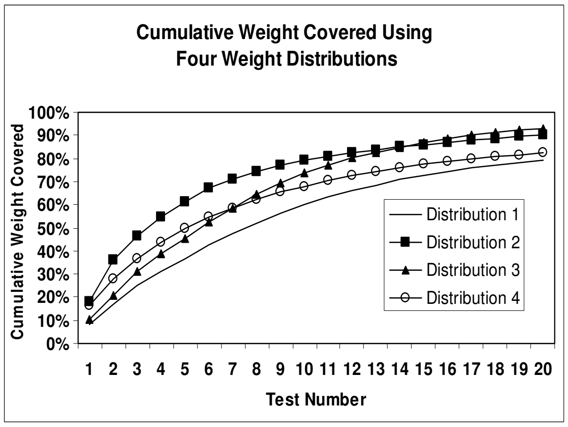

Figure 3: Cumulative weight covered in the first 20 test using input $20^2 10^2 3^{100}$

In addition to the rate of cumulative weight coverage, the size of the test suites generated vary:

- **Distribution 1 (Equal weights)**- 401 rows
- **Distribution 2 ($\frac{50}{50}$ split)**- 400 rows
- **Distribution 3 ($(\frac{1}{v_{max}})^2$ split)**- 416 rows
- **Distribution 4 (Random)**- 405 rows

Distribution 3 has the effect of emphasizing the importance of pairs involving factors with few levels, and in this example yields a larger covering array than unweighted density. In a more extreme example, we distributed weight as $(\frac{1}{v_{max}})^{10}$, producing a result of 433 rows.

The evidence from this scenario is that weighted density does generate tests of greater weight early, and that the weights themselves cause substantial variation in the size of covering array produced. We consider several more scenarios to examine this further. Table 5 shows the results in seven cases: three have all factors with the same number of levels and four are mixed level. Unweighted density usually produces the smallest sized covering array. Figures 4, 5, 6, and 7 show results of the cumulative weight distribution for each weight distributions. Naturally, in a typical application we cannot specify the weights, and hence these figures merely illustrate the algorithm's ability to accumulate large weight early. For instance, the first half of tests using any of the four weight distributions for 3^{100}, over 90% of the weight is covered; the first half of tests using any of the four weight distributions for $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$ covers between 75% to 99% of the total weight; and in $3^{50} 2^{50}$, the first half of tests covers over 95% of the total weight. When simply generating a covering array, we can control the weights used to minimize the size of covering array generated. The variation in sizes is reported in Table 5.

5. CONCLUSIONS

Previous algorithms used in tools to generate software interaction test suites have been evaluated on criteria of accuracy, execution time, consistency, and adaptability to seeding and constraints. This paper discussed a further impor-

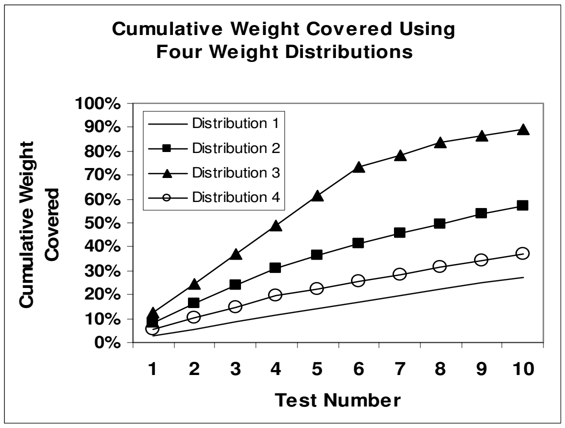

Figure 4: Cumulative weight covered in the first 10 test using input $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$

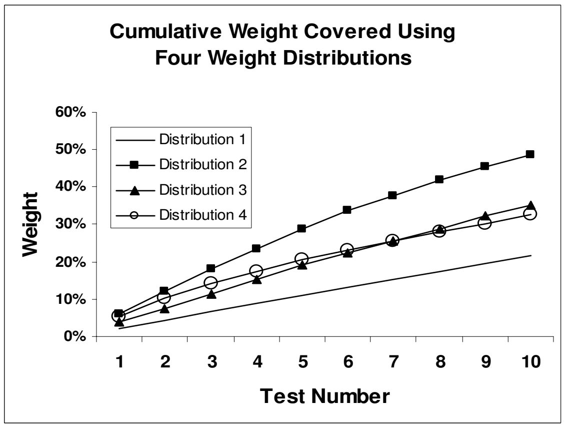

Figure 5: Cumulative weight covered in the first 10 test using input $8^2 7^2 6^2 2^4$

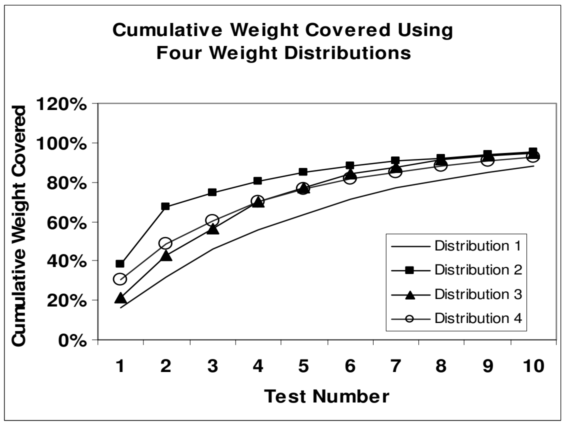

Figure 6: Cumulative weight covered in the first 10 test using input $3^{50} 2^{50}$

Weighting	Equal	$\frac{1}{v_{max}}^2$	$\frac{50}{50}$ Split	Random
3^4	9	13	9	13
10^{20}	206	314	225	223
3^{100}	32	38	31	32
$10^1 9^1 8^1$ $7^1 6^1 5^1$ $4^1 3^1 2^1$	94	125	98	101
$8^2 7^2 6^2 2^4$	70	98	77	81
$15^1 10^5 5^1 4$	175	238	185	188
$3^{50} 2^{50}$	28	35	28	28

Table 5: Sizes of biased covering arrays with different weight distributions

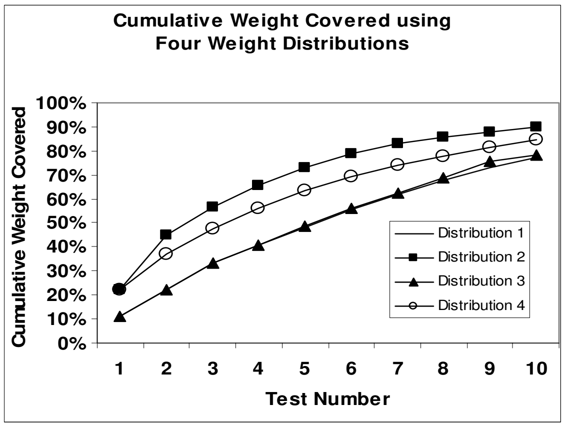

Figure 7: Cumulative weight covered in the first 10 test using input 3^{100}

tant criterion: prioritization based on user specified importance. An algorithm was described. Computational results suggest that the method provides a useful, and simple, mechanism for generating prioritized test suites. Greedy methods for constructing biased covering arrays can be useful when testers desire a prioritized ordering of tests.

Acknowledgements

Research is supported by the Consortium for Embedded and Internetworking Technologies and by ARO grant DAAD 19-1-01-0406.

6. REFERENCES

[1] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proc. 27th International Conference on Software Engineering (ICSE2005)*, page to appear, May 2005.

[2] R. F. Berry. Computer bench mark evaluation and design of experiments a case study. In *Proc. IEEE Wireless Communications Networking Conference (WCNC03)*, 41(10):1279-1289, 1992.

[3] T. Berling and P. Runeson. Efficient Evaluation of Multifactor Dependent System Performance Using Fractional Factorial Design. *IEEE Transactions on Software Engineering*, 29(9):769-781, 2003.

[4] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513, October 1998.

[5] C. Cheng, A. Dumitrescu, and P. Schroeder. Generating small combinatorial test suites to cover input-output relationships. *Proceedings of the Third International Conference on Quality Software (QSIC '03)*, pages 76–82, 2003.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–44, October 1997.

[7] D. M. Cohen, S. R. Dalal, M.L.Fredman, and G. Patton. Method and system for automatically generating efficient test cases for systems having interacting elements. *United States Patent, Number 5,542,043*, 1996.

[8] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):82–88, October 1996.

[9] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. *Proc. Intl. Conf. on Software Engineering (ICSE 2003)*, pages 38–48, 2003.

[10] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*, to appear.

[11] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, to appear.

[12] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. *Proc. of the IASTED Intl. Conference on Software Engineering*, pages 242–252, February 2004.

[13] C. J. Colbourn, Y. Chen, W.-T. Tsai. Progressive Ranking and Composition of Web Services Using Covering Arrays. *Tenth IEEE International Workshop of Object-oriented Real-time Dependable Systems (WORDS2005)*, to appear.

[14] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, B.M. Horowitz. Model-based testing in practice. *Proc. Intl. Conf. on Software Engineering (ICSE '99)*, pages 285–294, May 1999.

[15] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. *Proc. Intl. Conf. on Software Engineering (ICSE '97)*, pages 205–215, October 1997.

[16] A. Dumitrescu. Efficient algorithms for generation of combinatorial covering suites. *Proc. 14-th Annual Intl. Symp. Algorithms and Computation (ISAAC '03), Lecture Notes in Computer Science*, pages 300–308, 2003.

[17] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the 7th International Software Metrics Symposium*, pages 169–179 Apr. 2001.

[18] S. Elbaum, A. Malishevsky, and G. Rothermel. Using fault estimation to improve test case prioritization. Technical Report 99-60-13, Oregon State University, Corvallis, OR, Feb. 2000.

[19] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, May 2001.

[20] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 102–112, Aug. 2001.

[21] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 18(2):159–182, Feb. 2002.

[22] A. Hartman. Software and hardware testing using combinatorial covering suites. *Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms*, June 2002.

[23] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math.*, 284:149–156, 2004.

[24] N. Kobayashi, T. Tsuchiya, and T. Kikuno. A new method for constructing pairwise covering designs for software testing. *Information Processing Letters*, 81:85–91, 2002.

[25] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, October 2002.

[26] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Engineering*, 30(6):418–421, October 2004.

[27] R. Mandl. Orthogonal latin squares an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054-1058, 1985.

[28] National Institute of Standards and Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing.* U.S. Department of Commerce, May 2002.

[29] K. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Math.*, 138(9):143–152, March 2004.

[30] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology*, 13(3):277-331, July 2004.

[31] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188, Sept. 1999.

[32] K. Tai and L.Yu. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28:109–111, 2002.

[33] Y. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. *IEEE Aerospace Conf.*, pages 431–37, 2000.

[34] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proc. of the Interactional Symposium on Software Reliability Engineering*,pp.110-121, 2000.

[35] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. *Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications*, pages 301–311, October 2001.

[36] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering.*, pages 230–238, Nov. 1997.

[37] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Intl. Symp. on Software Testing and Analysis*, pages 45–54, July 2004.