

The density algorithm for pairwise interaction testing

Renée C. Bryce^{1,*},[†] and Charles J. Colbourn²

¹*School of Computer Science, University of Nevada at Las Vegas,
4505 Maryland Parkway, Las Vegas, NV 89154, U.S.A.*

²*Department of Computer Science and Engineering, Arizona State University,
P.O. Box 878809, Tempe, AZ 85287, U.S.A.*



SUMMARY

There are many published algorithms for generating interaction test suites for software testing, exemplified by AETG, IPO, TCG, TConfig, simulated annealing and other heuristic search, and combinatorial design techniques. Among these, greedy one-test-at-a-time methods (such as AETG and TCG) have proven to be a reasonable compromise between the needs for small test suites, fast test-suite generation, and flexibility to accommodate a variety of testing scenarios. However, such methods suffer from the lack of a worst-case logarithmic guarantee on test suite size, while methods that provide such a guarantee at present are less efficient or flexible, or do not produce test suites that are competitive in size for practical testing scenarios. In this paper, a new algorithm establishes that efficient, greedy, one-test-at-a-time methods can indeed produce a logarithmic worst-case guarantee on the test suite size. In addition, this can be done while still producing test suites that are of competitive size, and in a time that is comparable to the published methods. It is deterministic, guaranteeing reproducibility. It generates only one candidate test at a time, permits users to 'seed' the test suite with specified tests, and allows users to specify constraints of combinations that should be avoided. Further, statistical analysis examines the impact of five variables used to tune this density algorithm for execution time and test suite size: weighting of density for factors, scaling of density, tie-breaking, use of multiple candidates, and multiple repetitions using randomization. Copyright © 2007 John Wiley & Sons, Ltd.

Received 2 March 2005; Revised 24 September 2006; Accepted 2 November 2006

KEY WORDS: covering array; mixed-level covering array; one-row-at-a-time greedy algorithm; pairwise coverage; software interaction testing

*Correspondence to: Renée C. Bryce, School of Computer Science, University of Nevada at Las Vegas, 4505 Maryland Parkway, Las Vegas, NV 89154, U.S.A.

[†]E-mail: reneebyrce@cs.unlv.edu

Contract/grant sponsor: Consortium for Embedded and Internetworking Technologies
Contract/grant sponsor: ARO; contract/grant number: DAAD 19-1-01-0406



1. INTRODUCTION

Faults arise in component-based systems when unexpected interactions among components occur. For instance, consider an Internet-based software system in which customers may use a variety of components, including Web browsers, operating systems, connection types, and printer configurations, as shown in Table I. By choosing one of the available options for each of the components, a *test* is obtained. For the system in Table I, one possible test is {Browser=IE, OS=Linux, Connection=LAN, PrtConf=Screen}. Each such test can be executed, with the result that the test *passes* or *fails*; in the latter case, the presence of a *fault* is indicated. A *test suite* is a set of such tests. *Exhaustive testing* executes all possible tests; for the system in Table I, an exhaustive test suite contains $3^4 = 81$ tests.

Executing an exhaustive test suite may be feasible for such a small system, but the combinatorial growth of larger systems prohibits exhaustive testing. With ten components each having four possible options, $4^{10} = 1\,048\,576$ tests are needed! In view of the infeasibility of exhaustive testing, interaction testing has been proposed as a means to reduce the number of tests [1–4]. To make this precise, consider the possible causes of faults. It may happen that a particular option for one component is always faulty, and then every test in which this component is assigned the specified option must fail. Evidence of such faults will be seen using any test suite in which for every option of every component, there is at least one test in which the specified component is assigned the specified option. In general, this is not sufficient, since options for components interact. A fault may result, for example, whenever a screen printer is used with IE, while with other options both the screen printer and the IE browser function correctly. A *t-way interaction* is a choice of t of the components, and one option for each component. For example, {Browser=IE, PrtConf=Screen} is a two-way (or *pairwise*) interaction, while {OS=Linux, Connection=LAN, PrtConf=Screen} is a three-way interaction. *Interaction testing* is concerned with faults that arise from such interactions of t or fewer components; *pairwise testing* is the specific case when $t = 2$. The goal of interaction testing is to ensure, for some small, fixed value of t , that every t -way interaction appears as a part of at least one test in the test suite; this would ensure that if the interaction causes a fault, the fault will be represented in the outcomes of the tests in that test suite. In general, any test for a system with k components will contain (or ‘cover’) one t -way interaction for each of the $\binom{k}{t}$ ways to choose t components from the k components. The t -way interactions must all be covered, using tests that each involve all k components.

A pairwise interaction test suite for the input in Table I contains only nine different tests (see Table II). With ten components each with four possible options, all pairs of interactions can be covered using at most 25 tests.

Interaction testing has been applied in a number of studies. Dalal *et al.* present empirical results to argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults [5]. Berling and Runeson use interaction testing for a target identification system that identifies both real and false targets [6]. In further work, Burr and Young provide more empirical results to show that this type of test coverage is effective [7]. Dunietz *et al.* link the effectiveness of these methods to software code coverage. They show that high code block coverage is obtained when testing all two-way interactions, but higher strength is needed for good path coverage [2]. Lazić and Velašević integrate interaction testing with modelling and simulation to improve the productivity of an automated target-tracking radar system [8]. Their work uses orthogonal arrays, specific examples of covering arrays. Kuhn *et al.* examined fault reports for several systems [9,10]. They show that 70% of



Table I. Four components, each with three options.

Web browser	Operating system	Connection type	Printer configuration
Netscape	Windows	LAN	Local
IE	Macintosh	PPP	Networked
Mozilla	Linux	ISDN	Screen

Table II. Test suite covering all pairs from Table I.

Test	Browser	OS	Connection	Printer
1	Netscape	Windows	LAN	Local
2	Netscape	Linux	ISDN	Networked
3	Netscape	Macintosh	PPP	Screen
4	IE	Windows	ISDN	Screen
5	IE	Macintosh	LAN	Networked
6	IE	Linux	PPP	Local
7	Mozilla	Windows	PPP	Networked
8	Mozilla	Linux	LAN	Screen
9	Mozilla	Macintosh	ISDN	Local

faults can be discovered by testing all two-way interactions, while 90% can be detected by testing all three-way interactions. Six-way coverage was required in these systems to detect 100% of the faults reported [9]. A similar study is reported in [10], and the relationship between test suite minimization and effectiveness is examined in [11].

Other applications of interaction testing in the software and hardware domains include regression testing through the graphical user interface (GUI) [12–14], fault localization [15], compiler testing [16], and benchmark testing [17]. Indeed, there are many possible applications of interaction testing that can benefit from tools that automatically construct such test suites. The density algorithm developed here can be used for such tools.

To generate interaction test suites, a combinatorial object called a covering array is often used. A *covering array* $CA_\lambda(N; t, k, v)$ is an $N \times k$ array on v symbols such that every $N \times t$ subarray contains all tuples from v symbols of size t at least λ times. Here an $N \times t$ subarray is any $N \times t$ array obtained by selecting t of the columns, and deleting the remaining $k - t$ columns. In this subarray every row is a *tuple*, or list of t elements. In essence, the subarray contains all of the t -way interactions arising from these t components that are covered in the set of tests (rows). In the application of interaction testing, $\lambda = 1$, which means that every t -tuple only needs to be covered at least once. The notation $CA(N; t, k, v)$ can be used.

The variable t is the *strength* of coverage, k is the number of components (or *factors*), and v is the number of options (or *levels*) associated with each component. A test suite is an $N \times k$ array where



each row is a test case. Each column represents a component and the value in the column is the level chosen. An *orthogonal array* is a covering array in which every t -tuple is covered exactly λ times. In Table II, $t = 2$ (pairwise coverage), $k = 4$ (four components), $v = 3$ (three options associated with each component), and $N = 9$ (the size of the test suite). All pairwise interactions between every two columns are tested in this test suite.

These (*fixed-level*) covering arrays only apply when all factors have the same number of levels. Several authors have suggested use of the mixed level covering array for software testing (see [1,18–21]). A *mixed-level* covering array, $MCA(N; t, k, (v_1, v_2, \dots, v_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^k v_i$, with the following properties.

- (1) Each column i ($1 \leq i \leq k$) contains only elements from a set S_i with $|S_i| = v_i$.
- (2) The rows of each $N \times t$ subarray cover all t -tuples of levels from the t columns at least once.

A shorthand notation is used to describe mixed-level covering arrays by combining equal entries in $(v_i : 1 \leq i \leq k)$. For example, three entries each equal to two can be written as 2^3 . Then an $MCA(N; t, k, (v_1, v_2, \dots, v_k))$ can be written as an $MCA(N; t, (w_1^{r_1}, w_2^{r_2}, \dots, w_s^{r_s}))$ where $k = \sum_{i=1}^s r_i$ and $(w_j : 1 \leq j \leq s) \subseteq \{v_1, v_2, \dots, v_k\}$.

In the subsequent sections, the main contributions are: the development of an efficient, one-test-at-a-time greedy algorithm for constructing mixed-level covering arrays (pairwise interaction test suites) that ensures a logarithmic worst-case guarantee on test suite size, which previous efficient greedy methods do not; a specific implementation of this algorithm that, despite the requirement for such a strong worst-case guarantee, remains competitive with other greedy methods in terms of execution time and test suite size; and a statistical technique for optimizing the performance of the algorithm when considering a number of implementation decisions. In Section 2, criteria for algorithms to generate the test suites are discussed. Existing algorithms are evaluated according to these criteria; while each of the available algorithms exhibits some desirable properties, none meets all of the criteria identified. However, a comparison of published data show that greedy algorithms exhibit relatively good results. In Section 3, the focus moves to greedy algorithms and descriptions of AETG and Test Case Generator (TCG) algorithms are provided. In Section 4, a new efficient greedy algorithm is introduced that ensures a logarithmic worst-case guarantee on the size of test suites in relation to the number of factors (while such a logarithmic guarantee is known, what is new is that it can be ensured by an efficient, greedy one-test-at-a-time method). In Section 5, the new algorithm is described precisely and, in Section 6, computational results are compared with algorithms from the literature. The method proposed often produces smaller sized test suites than existing techniques, and in general produces test suites of competitive size. In Section 7, variants of the new method are examined statistically to optimize algorithm performance. Finally, in Section 8, the contributions are summarized.

2. CRITERIA FOR CONSTRUCTING INTERACTION TEST SUITES

A variety of methods for generating test suites for pairwise coverage arise from a number of different objectives to be addressed. These may include (1) *size of test suites*, (2) *execution time to generate test suites*, (3) *consistency of test suites generated*, and (4) *accommodation of seeds and constraints* (defined shortly).



The smallest possible test suite that covers all t -way interactions may be desired since every extra test adds to the cost of testing. No published algorithm provides the smallest test suite for every possible input. Naturally, both good performance in the average case *and* a *worst-case guarantee* on test suite sizes are desirable. At the same time, a short execution time to generate test suites is needed to reduce the time spent constructing, rather than executing the test suite. When an algorithm uses randomization to generate a test suite, different testers can have quite different experiences with the same method. A test suite size that is reported once may not be easily reproduced at a later time.

It is rarely the case, even with automatic test generation, that a tester does not wish to impose seeds or constraints. For example, the ability to *seed* a test suite by specifying the inclusion of certain tests is often necessary. In addition, *constraints* among the factors can dictate that the test suite *avoid* certain tests; these ‘avoids’ can be in the form of pairs that need not be covered, or of pairs that cannot be covered [22].

These many criteria have led to a wide variety of approaches. Algebraic and combinatorial constructions for covering arrays appear in the literature (see [18,23–28] for example). Combinatorial constructions are only known for some parameter sets; this severely limits the applicability to practical testing problems. Nevertheless, Williams and Probert [3,21] developed a strategy, *TConfig*, for employing a recursive construction based on orthogonal arrays to construct test suites. Their method provides a worst-case guarantee on test suite size that is optimal up to a constant factor. It does not provide for seeds or avoids. While it shares the speed of other combinatorial constructions and permits more general applications, it generates test suites that are often much larger than needed. This problem is especially prevalent in mixed-level covering arrays (when individual factors have different numbers of levels), since the method is designed for fixed-level problems.

Computational methods have been explored to address the concerns with test suite size. Exhaustive generation is intractable, and optimization methods such as linear programming have proved successful only on small problems [28]. Computational search techniques take a covering array through a series of transformations, computing the cost of a change, and accepting the change according to an acceptance criterion. Examples include techniques such as hill climbing, tabu search, and simulated annealing; other sophisticated search techniques have proved less successful until this time [20]. Simulated annealing is discussed in [29,30], and tabu search is examined in [31]. Both hill climbing and simulated annealing provide general methods that appear to produce the smallest test suites across a wide range of problems [30]. They provide for seeds. However, predictability is a major concern, and the time to construct a test suite can be prohibitive. They also fail to produce a worst-case guarantee on test suite size. Nevertheless, if minimizing the size of the test suite is of paramount concern, simulated annealing currently appears to be the best method available. Techniques for specific instances can also be found in [32–34].

Greedy algorithms have also been designed to generate test suites [1,4,35–37]. Greedy algorithms can address seeding and constraints [22], can generate test suites quickly, and can produce relatively accurate results. For instance, the AETG system and TCG [1,37,38] use a greedy technique. Each test suite is built one test at a time. For each subsequent test to be added, many are created and then the best is chosen (see [1,37,38]). The greedy portion of these algorithms lies in the step of determining which level to add to each factor of each test. This is chosen to be a local optimum.

In each algorithm, information is maintained about which interactions are still uncovered and is used as a heuristic to provide a better chance of finding the missing interactions. AETG uses a random approach to finding a pool of tests. Tung and Aldiwan [37] suggest a deterministic algorithm, TCG.



Table III. Published accuracy and time for implementations of TCG, AETG, and simulated annealing (SA) as reported in [24].

Input	Our-TCG (size/time in seconds)	Our-AETG (size/time in seconds)	SA (size/time in seconds)
$5^1 3^8 2^2$	18/6	20/58	15/214
$7^1 6^1 5^1 4^5 3^8 2^3$	42/57	44/489	42/874
$5^1 4^4 3^{11} 2^5$	25/33	28/368	21/379
$6^1 5^1 4^6 3^8 2^3$	32/42	35/376	30/579
10^{20}	213/1333	198/6001	183/10 833

Table IV. Published accuracy for algorithms that construct covering arrays (results in italics are not previously reported in the literature).

Input	TConfig [40]	IPO [39]	AETG [38]
3^{13}	<i>15</i>	19	15
4^{40}	40	49	<i>42</i>
4^{100}	43	52	<i>51</i>
2^{100}	14	15	10
10^{20}	231	212	180

The authors begin with a deterministic ordering of the factors. Another greedy algorithm, *In-Parameter-Order (IPO)*, has the benefit of reusing old tests when new factors are added. It does this by generating test suites and assigning levels to factors in a vertical and horizontal order [39].

Each of the algorithms discussed thus far have strengths and weaknesses in regard to the size of solutions (*accuracy*), consistency of results, and execution time. Published results for the algebraic, greedy, and heuristic search algorithms are briefly summarized by accuracy in Tables III–V. In some cases, sizes for inputs were not available in publication and these results are shown in italics. Missing data for AETG and TConfig are reported based on results from their publicly available tools. Missing data reported for TCG are based on an implementation in [30]. The discrepancy in the AETG results for input 10^{20} appear to result from the fact that, while the primary algorithm in AETG is a greedy, one-row-at-a-time method, the full algorithm also permits combinatorial methods to be used. While the size 180 is published [1], the commercial AETG tool reports a value of 198; this variation is quite large, but not entirely unexpected. The results in Table III show examples in which simulated annealing produces the smallest solutions, but this is at the cost of additional execution time. Results in Tables IV and V compare solutions from algebraic and greedy methods. Sometimes one method outperforms the others, but not for all inputs.



Table V. Published accuracy for algorithms that construct mixed-level covering arrays (results in italics are not previously reported in the literature).

Input	TConfig [40]	IPO [39]	AETG [38]	TCG [37]
$5^1 3^8 2^2$	<i>21</i>	<i>21</i>	19	20
$7^1 6^1 5^1 4^5 3^8 2^3$	<i>91</i>	<i>48</i>	45	45
$5^1 4^4 3^{11} 2^5$	<i>32</i>	<i>28</i>	30	30
$6^1 5^1 4^6 3^8 2^3$	<i>50</i>	<i>35</i>	34	33
$4^{15} 3^{17} 2^{29}$	<i>40</i>	<i>36</i>	41	35
$4^1 3^{39} 2^{35}$	<i>30</i>	<i>29</i>	28	27

Each of the published methods work well under certain criteria. In the remainder of this paper, one-row-at-a-time greedy algorithms are examined due to their relatively good results for accuracy and execution time.

3. ONE-ROW-AT-A-TIME GREEDY ALGORITHMS

Two one-row-at-a-time greedy algorithms include AETG and TCG. Descriptions of these algorithms are provided and then a new algorithm that improves upon these methods is presented.

3.1. The AETG algorithm

In AETG, covering arrays are constructed one row at a time. To generate a row, the first t -tuple is selected based on the one involved in most uncovered pairs. Remaining factors are assigned levels in a random order. Levels are selected based on the one that covers the most new t -tuples. For each row that is actually added to the covering array, there are a number, M , candidate rows that are generated and only a candidate that covers the most new pairs (or t -tuples) is added to the covering array. Once a covering is constructed, a number, R , of test suites are generated and the smallest test suite generated is reported. This process continues until all pairs are covered. Figure 1 provides pseudocode for this algorithm.

3.2. The TCG algorithm

In TCG, one row is added at a time to a covering array until all pairs are covered. Before each row is added, a number of up to M candidate rows are generated and the best candidate (covering the most new pairs) is added. M is defined to be the maximum cardinality of factors (the maximum number of levels associated with any factor). To construct each row, factors are assigned levels in an order based on a non-ascending order of each factor's cardinality. Each level for the factor is evaluated and a count of the number of pairs that are covered is used to determine whether or not to select a level for a factor. Figure 2 shows pseudocode for TCG.



```

set MinArray to  $\infty$ 
repeat R times
  start with no tests in T
  N =  $\infty$ 
  while there are uncovered t-tuples in T
    start with an empty test C and an empty test BestCandidate
    repeat M times
      select the first pair that appears in the largest number of uncovered pairs
      while free factors remain
        randomly select a factor f
        select a level v that is in the largest number of uncovered pairs with fixed factors
      end while
      if C covers more t-tuples than BestCandidate
        BestCandidate = C
    end repeat
    add test BestCandidate to T
    N++
  end while
  if T has N < MinArray tests, set MinArray = N, BestArray = T
end repeat

```

Figure 1. AETG pseudocode.

```

start with no tests in T
sort factors in non-ascending order of cardinality
while there are uncovered t-tuples in T
  for i = 1 . . . M candidates
    assign  $k_{0_{v_i}}$  to  $k_0$ 
    for j = 1 . . . (k - 1)
      select a level for  $k_i$  that covers the largest number of
        uncovered t-tuples in relation to fixed factors
      break ties by selecting the least recently used level
    end for
  add the candidate that covers the most uncovered t-tuples to T
end while

```

Figure 2. TCG pseudocode.



4. A LOGARITHMIC GUARANTEE

It is well known (see [41]) that there exists a test suite whose number of tests grows as a logarithmic function of the number of factors, and that there is also a logarithmic *lower* bound on the test suite size. Indeed, in [1] it is shown that a greedy method equipped with the ability to select the ‘best’ next test can achieve such a logarithmic bound. Until this time, however, no greedy one-test-at-a-time method has exhibited an efficient (polynomial-time) strategy for selecting the next test that leads to such a logarithmic guarantee. One of the contributions of this article is to establish that such an *efficient*, greedy one-test-at-a-time method exists. This is explored next.

A tester desires the smallest possible test suite in practice and may also desire a guarantee on the size of test suites. AETG and TCG do not have a guarantee. Consider the strategy proposed by AETG in more detail. Suppose that a system with k factors f_1, \dots, f_k is the system under test. The factor f_i is permitted to take on any of v_i levels, denoted by $\{\sigma_{i,j} : j = 1, \dots, v_i\}$. The objective is to produce an $MCA(N; 2, k, (v_1, \dots, v_k))$.

The AETG system attempts to make a ‘small’ covering array using a greedy strategy. It selects a single test at a time, repeating this until all pairs are covered in at least one of the selected tests. Since the objective is to minimize the number of tests, AETG concentrates on the selection of each test to maximize the number of previously uncovered pairs that are covered by this test. This paper makes two main contributions [1].

- (1) It shows a *logarithmic* upper bound on the number of tests needed as a function of k that matches the trivial logarithmic lower bound (see [41]) up to a constant factor.
- (2) It describes a (greedy) heuristic for the selection of tests.

The first relies on a conceptually simple construction method for covering arrays. Having selected some (partial) collection of tests, record the pairs \mathcal{P} yet to be covered. Among the $\prod_{i=1}^k v_i$ possible tests, there can be substantial variation in the number of pairs in \mathcal{P} that the test covers. Select a test that covers that largest number of pairs in \mathcal{P} , add it to the collection of tests, and repeat this step until $\mathcal{P} = \emptyset$; at this point, the covering array has been produced. This is a greedy method, and by no means guarantees the minimum possible size of the test suite. However, it does ensure that at each stage, at least $|\mathcal{P}|/L$ new pairs are covered where L is the product of the two largest of the sizes $\{v_i\}$. This, in turn, ensures that the size of the test suite constructed is bounded by a logarithmic function of the number k of factors (see [1] for details).

However, the authors do not propose an algorithm for finding the test that covers a maximum number of uncovered pairs, and instead adopt a greedy heuristic to produce each new test in turn. Their method is reviewed here. Each test is selected from a pool of M candidate tests, where M is a constant chosen in advance. To generate each candidate, first select a factor f_i and a level σ_i for this factor so that the choice of σ_i for f_i appears in the maximum number of uncovered pairs. Set $\pi(1) = i$. Then choose a random permutation of the indices of the remaining factors to form a permutation $\pi : \{1, \dots, k\} \rightarrow \{1, \dots, k\}$. Now assume that levels τ_1, \dots, τ_i have been selected for factors $f_{\pi(1)}, \dots, f_{\pi(i)}$. Select a level τ_{i+1} for factor $f_{\pi(i+1)}$ by selecting that level which yields the maximum number of new pairs with the selected levels for the i factors already fixed.

Repeating this process M times exploits the randomness in the factor ordering, and yields different tests from which to select. Naturally, one selects the best in terms of newly covered pairs, and adds it to the test suite.



While the authors note that this appears to exhibit a logarithmic performance in practice, their earlier guarantee does not apply because the test selection does not ensure that a selected test covers the maximum possible number of new pairs. In view of the next result, this is not surprising.

Given a collection \mathcal{P} of uncovered pairs, and a specified number p , it is NP-complete to determine whether there exists a test covering at least p pairs. Seroussi and Bshouty [42] prove a similar result, and it is proved in this form in [36]. In plain terms, what this says is that an efficient technique to select a test covering the maximum number of uncovered pairs is unlikely to exist. However, this leaves an unsatisfactory situation. The current proof of logarithmic growth hinges on selecting such a test! Fortunately, the situation is not as bad as it first appears. Indeed a more careful reading of the proof of logarithmic growth establishes that one does not need to find a test which covers the maximum number of uncovered pairs. All one needs is to find a test that covers the *average* number of uncovered pairs.

It appears that the test selection in AETG does not ensure this, although for practical purposes, unless M is quite small, the likelihood that at least one of the M candidates has this property is high. Nevertheless, it is reasonable to ask for a test selection technique that *guarantees* to cover at least the average number. This is pursued next.

The lack of a guarantee results primarily from the greedy nature of the test selection. In particular, when selecting a level for the i th factor, only its interaction with the first $i - 1$ factors is considered. This can (and does) result in a selection which make selections for the later factors less desirable. Indeed if there are 100 factors, and a level for the fifth is being selected, its interaction with the later 95 factors is arguably more important than its interaction with the first four. This intuition suggests an alternate approach.

Consider the construction of a test suite with k factors. The number of levels for factor i is denoted by v_i . For factors i and j , the *local density* is $\delta_{i,j} = r_{i,j}/v_i v_j$ where $r_{i,j}$ is the number of uncovered pairs involving a level of factor i and a level of factor j . In essence, $\delta_{i,j}$ indicates the fraction of pairs of assignments to these factors which remain to be tested. The *global density* is $\delta = \sum_{1 \leq i < j \leq k} \delta_{i,j}$. At each stage, a test covering at least δ uncovered pairs is sought.

To select such a test, repeatedly fix a level for each factor, and update the local and global density values. At each stage, some factors are *fixed* to a specific level, while others remain *free* to take on any of the possible levels. When all factors are fixed, the test has been chosen. Otherwise, select a free factor f_s . Now $\delta = \sum_{1 \leq i < j \leq k} \delta_{i,j}$, which can be separated into two terms:

$$\delta = \sum_{\substack{1 \leq i < j \leq k \\ i, j \neq s}} \delta_{i,j} + \sum_{\substack{1 \leq i \leq k \\ i \neq s}} \delta_{i,s}$$

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

Write $\rho_{i,s,\sigma}$ for $1/v_i$ times the number of uncovered pairs involving some level of factor f_i , and level σ of factor f_s . Then rewrite the second summation as

$$\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \delta_{i,s} = \frac{1}{v_s} \sum_{\sigma=1}^{v_s} \sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma}$$

Choose σ to maximize $\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma}$. It follows that $\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma} \geq \sum_{\substack{1 \leq i \leq k \\ i \neq s}} \delta_{i,s}$. 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).



```
start with empty test suite
while uncovered pairs remain do
  compute factor density for each factor
  initialize new test with all factors not fixed
  while a factor remains whose level is not fixed
    select such a factor  $f$  with largest density,
    using a factor tie-breaking rule
  compute level density for each level of factor  $f$ 
  select a level  $\ell$  for  $f$  with maximum density
  using a level tie-breaking rule
  fix factor  $f$  to level  $\ell$ 
  recompute densities for each factor
  end while
  add test to test suite
end while
```

Figure 3. DDA pseudocode.

This process is iterated 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 has some practical value to be seen later, but does not affect the logarithmic growth.

Applying this method to the case where each factor has the same number of levels, the density is the average number of uncovered pairs in a test that could be selected. The selection of a test with at least this number of uncovered pairs is guaranteed.

5. A DETERMINISTIC DENSITY ALGORITHM

While the density argument developed establishes that greedy methods can indeed yield a worst-case guarantee that is logarithmic, it does not address the question of whether controlling the worst case has a negative impact on the expected size of test suites. In this section a preliminary implementation of the deterministic density algorithm (DDA) is developed; later a more general implementation is discussed.

The DDA constructs one row of a covering array at a time using a steepest ascent approach. Factors are dynamically fixed one at a time in an order based on density. New rows are continually added until all interactions have been covered. In order to make the previous discussion precise, consider the algorithm in Figure 3.

Four decisions (shown in boxes) must be made to instantiate this prototype: (1) *factor density*, the manner in which densities are computed for factors; (2) *factor tie-breaking rule*, what tie-breaking is done when two or more maximum densities for factors are equal; (3) *level density*, the manner in which densities are calculated for levels; and (4) *level tie-breaking rule*, what tie-breaking is done when two or more maximum densities for levels are equal.



Table VI. DDA density formulae.

		Number of levels k_i	Number of levels k_j	Density formula
Factor density formulae	Equation (1)	>1	>1	$\delta = \left(\frac{r_{i,j}}{\ell_{\max}^2}\right)^\alpha$
	Equation (2)	1	>1	$\delta = \left(\frac{r_{i,j}}{\ell_{\max}}\right)^\alpha$
	Equation (3)	>1	1	$\delta = \left(\frac{r_{i,j}}{\ell_{\max}}\right)^\alpha$
	Equation (4)	1	1	If uncovered = 1.0, Otherwise = 0.0
Level density formulae	Equation (5)	1	>1	$\delta = \left(\frac{r_{i,j}}{\ell_{\max}}\right)$
	Equation (6)	1	1	If uncovered = 1.0, Otherwise = 0.0

The density calculations are done as in Table VI; scaled versions of these formulae are treated in a later section.

5.1. Factor density

The density formulae are computed for several scenarios in Table VI. First consider factor density. Define the *current factor* as the factor for which the factor density is being calculated, $r_{i,j}$ as the number of uncovered pairs between the current factor i and another factor j , and ℓ_{\max} as the largest cardinality of the factors. The density δ_{ij} for factors i and j is as follows.

- Equation (1): if both factors have more than one level left, then

$$\delta_{ij} = \left(\frac{r_{i,j}}{\ell_{\max}^2}\right)^2$$

- Equations (2) and (3): if only one factor has one level left and the other has greater than one level remaining, then

$$\delta_{ij} = \left(\frac{r_{i,j}}{\lambda_{\max}}\right)^2$$

- Equation (4): if both factors have exactly one level left, then if a new pair is covered, $\delta_{ij} = 1.0$; otherwise, $\delta_{ij} = 0.0$.

The *density* of factor i is the summation of these local densities over each factor $j \neq i$.

Table VII. Example input $4^13^12^1$.

	Factor		
	1	2	3
Levels	0, 1, 2, 3	4, 5, 6	7, 8

5.2. Factor tie-breaking

For factor tie-breaking, a number of rules are explored in Section 6. However, initially here, tie-breaking simply chooses the lexicographically first among the maxima.

5.3. Level density

Once a factor is selected, a level needs to be assigned to it. Table VI shows the density formula. Level density is computed for a specific level, v_i , in relation to an individual factor, k_j , as follows.

- Equation (5): if k_j has more than one level left that is involved in uncovered pairs with v_i , then $r_{i,j}$ is the number of levels that are in uncovered pairs with v_i . This is divided by the maximum number of levels associated with any factor, ℓ_{\max} . The larger that this density ($r_{i,j}/\ell_{\max}$) is, the more likely that selecting this level will cover a new pair.
- Equation (6): if k_j has only one level left that is involved in uncovered pairs, and a new pair is covered, $\delta = 1.0$; otherwise, $\delta=0$.

5.4. Level tie-breaking

For level tie-breaking, a number of rules are explored in Section 6. However, initially here, tie-breaking simply chooses the lexicographically first among the maxima.

5.5. Example trace

Consider the example input in Table VII, $4^13^12^1$ (read as one factor has four levels, one has three levels, and one has two levels).

The density is calculated in relation to both factors that have already been fixed (assigned levels) and those that have yet to be fixed. Consider the generation of the sixth test in Figure 4. Factor density values are shown at this point in time. Factor f_0 has the largest density, so it is assigned a level first. There are four levels to consider and the densities are shown in Figure 4. Levels 1, 2, and 3 tie with the largest density values, so the tie is broken by selecting the first level. The process is repeated until the row is completely generated. New rows are added until all pairs are covered.

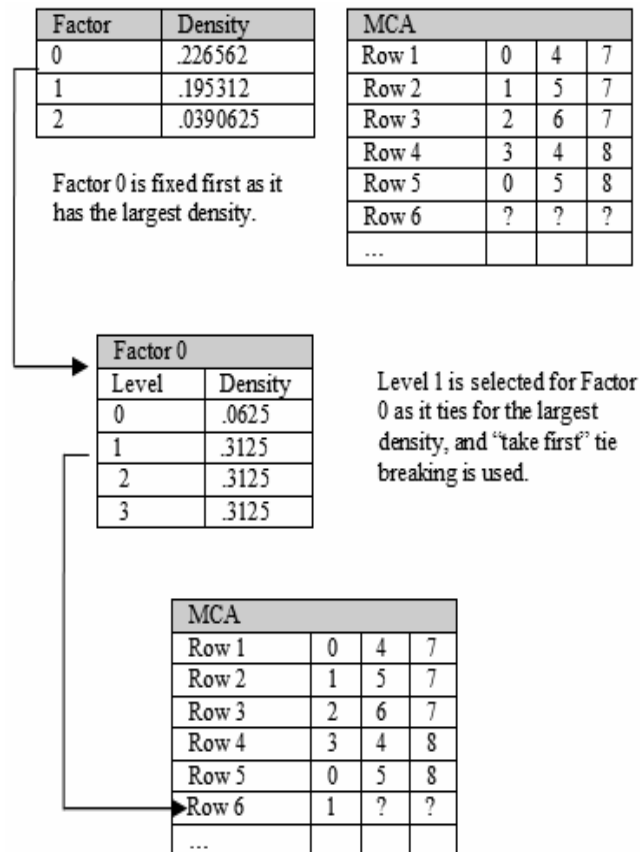


Figure 4. An example trace through DDA: generating the sixth test for input $4^1 3^1 2^1$.

5.6. Seeding and constraints

To accommodate seeds, tests are processed and any pairs covered in these tests are recorded to avoid redundant coverage of pairs. Constraints of pairs that should be avoided can be addressed by weighting the individual pairs to cover as presented in [22]. Pairs that are desired to be covered are given positive weights, while constraints of pairs that are less desirable are given negative weights. The modified density algorithm then addresses constraints by modifying density formulae to emphasize the covering weight instead of the pairs [22].



Table VIII. Comparison of sizes with published results (results in italics are not previously reported in the literature).

Parameters	Minimum size of a test suite				
	DDA	AETG	TCG	IPO	TConfig
$5^1 3^8 2^2$	21	19	20	<i>21</i>	<i>21</i>
$7^1 6^1 5^1 4^5 3^8 2^3$	43	45	45	48	<i>91</i>
$5^1 4^4 3^{11} 2^5$	27	30	30	28	32
$6^1 5^1 4^6 3^8 2^3$	34	34	33	35	<i>50</i>
$4^{15} 3^{17} 2^{29}$	35	41	<i>35</i>	36	40
$4^1 3^{39} 2^{35}$	27	28	27	29	30
3^{13}	18	15	<i>20</i>	19	<i>15</i>
2^{100}	15	10	<i>16</i>	15	14
4^{40}	43	<i>42</i>	<i>46</i>	49	40
4^{100}	51	<i>51</i>	55	52	43
10^{20}	201	180 (<i>198</i>)	<i>218</i>	212	231

6. COMPUTATIONAL RESULTS FOR DDA

In order to assess the practicality of DDA, a simple C program is used to compare the sizes of test suites obtained against those in the literature for TCG [37], AETG [1,38,43], IPO [4,39], and TConfig [3,40]. In some cases, results are not available in the published literature. Missing results for AETG and TConfig are reported based on tools made available by the original authors. Missing results for TCG are reported from an independent implementation [30] and missing results for IPO are drawn from a publicly available implementation built into the TConfig tool. These results are used to provide an assessment of the relative practicality of DDA.

Table VIII presents the sizes of test suites for a collection of mixed-level covering arrays, and for a few fixed-level covering arrays, for which results appear in the literature. Results that are independent of the published literature are shown in italics in the table. TConfig, as expected, appears to be quite effective in fixed-level cases when the number of levels is a prime or prime power, but does not fare as well in other cases. AETG constructs $M = 50$ candidates for each test and selects the best; the method by which the candidates is chosen involves randomly permuting the factors. TCG deterministically builds a number of candidates equal to the maximum number of levels for a factor, and chooses the best. Nevertheless, while DDA constructs only a single candidate for each test, the results shown suggest that it is competitive; an extension of DDA is explored in the following to maintain multiple candidates and select the best.

It would be incorrect to think that the minimum size of the test suite is the only parameter of importance. Indeed, if this were the case, one could in theory employ an exhaustive search method rather than a greedy one. More realistically, one could employ hill-climbing or simulated annealing [30]. For example, for the mixed-level covering array $5^1 4^4 3^{11} 2^5$, simulated annealing yields



a solution with only 21 tests (compared with 27 for DDA, and 30 for TCG and AETG in Table VIII). Nevertheless, this simulated annealing result took 579 seconds of compute time [30], while (on a different machine) DDA took 0.16 seconds. The point here is not to compare these running times directly, as different platforms and implementations are involved. Rather, the point is to emphasize that greedy methods are intended for speed. With this in mind, some execution times on a SunBlade 1000 system are reported for DDA. The longest running time observed was 24.9 seconds for the CA(2, 100, 4). The case 2^{100} took 7.41 seconds, and the case $4^13^{29}2^{35}$ took 5.81 seconds, and the case $4^{15}3^{17}2^{29}$ took 4.12 seconds. The case 10^{20} took 1.1 seconds, and all other cases took less than 0.2 seconds each. These results demonstrate that DDA can produce test suites of a reasonable size in a modest amount of time.

7. VARIATIONS ON THE THEME

In order to ensure determinism and speed, DDA has been presented as a specific density algorithm for generating test suites. However, several variables can have an impact on the time and accuracy. These include the weighting of the density formula for factors, scaling of density, and tie-breaking. The importance of these variables is assessed here by a standard analysis of variance (ANOVA) test [44]. In addition, as with TCG and AETG, density algorithms can employ multiple candidates and repetitions. These variations suggest a host of possible density algorithms, of which DDA is just one example.

7.1. Fundamental variables: density weighting, density scaling, and tie-breaking

The density is computed based on the number of pairs that are covered with fixed factors and the probability of covering pairs with free factors. The amount of preference given to free factors with the density formula is called *look-ahead* and can be scaled. Consequently, *look-back* is the preference given to fixed factors and can also be scaled in the density formulae.

Scaling densities and staying within the logarithmic guarantee is ensured by selecting a next test that covers at least the average number of uncovered pairs. However, the intuition is that better results will be obtained by covering the largest number. While this is NP-hard, it nonetheless indicates that choices should be made to improve the density of coverage as much as possible. First, consider the selection of factors. The density of a factor f_i can be taken as $\delta_i = \sum_{1 \leq j \leq k, j \neq i} \delta_{i,j}$. At every stage, the factor having the largest density is selected. Once a factor is chosen, a level for this factor must be selected. To do this, for each possible level of the factor, calculate the resulting density for the factor when fixed to the specified level. Then select the level that yields the largest increase in density, and update the densities. In essence, density is a surrogate for the number of pairs that become covered; in this way, the issue of not knowing which specific pairs are covered until after the test is selected can be avoided.

Making the 'best' selection locally ensures that a set of choices that is at least as good as the average is always retained. However, a few moments' thought reveals some concerns, particularly for mixed-level arrays. The measure of local density scales the number of uncovered pairs by the initial number to be covered. Hence, between two factors with two levels each, every pair contributes $\frac{1}{4}$ to the global density, while if the two factors have ten levels each, a pair contributes only $\frac{1}{100}$. This runs counter to the expectation that the pairs in the first situation are much more easily covered than those in the second.



To address this, local density is redefined in a way that does not affect the logarithmic guarantee. Let ℓ_{\max} be the largest number of levels for any factor. In order to handle two factors each having more than one level, in the local density replace the denominator by ℓ_{\max}^2 , making every pair equally important from a density viewpoint. When one factor has had its level chosen (meaning that it has only one level now) and the other remains to be chosen, use the denominator ℓ_{\max} . When both factors have been fixed, use the denominator. This corrects the method to focus on factors with many levels rather than factors with few.

The algorithm using local densities defined in this way has been implemented. Using densities in this way does not exhibit a preference among the pairs to be covered. Consider two factors. Considering a possible level for the first, it may have many or few uncovered pairs with the second factor; local density as defined does not provide a greater reward for covering those pairs involving a level appearing in many uncovered pairs. Hence, as a practical matter, the notion of local density is refined further.

The objective is to revise the definition of density so that between one factor and another, each pair as it becomes covered makes a smaller reduction in the density. In this way, the selection of the best improvement in density leads to a preference to cover pairs involving levels having many remaining uncovered pairs. Implementing this is straightforward. Choose an *inflation factor* α . For two factors f_i and f_j , and a level σ of f_i , calculate the ratio of uncovered pairs involving σ and a level in f_j to the value ℓ_{\max} (or 1 if f_j has already been fixed). Summing over all levels of σ and dividing by ℓ_{\max} (or 1 if factor f_i has been fixed), and then raising the result to the power α , gives the local density $\delta_{i,j}$. The resulting formulae are given in Table VI.

In this way, the contribution of a pair to the density, when the pair involves factors between which many uncovered pairs remain, is much larger. Initially, an inflation factor $\alpha = 2$ is selected (that is, squaring the previous contributions); later, other values are considered. A change in α does not negate the logarithmic guarantee, but rather changes the guarantee by a constant factor.

In order to assess the relative importance of ‘look-back’ to factors whose levels are fixed versus ‘look-ahead’ to factors that are free (yet to be fixed), the density formulae can be revised as follows. Choose a value β with $0 < \beta < 1$. Partition the factors into two classes, those whose levels have been fixed and those whose levels remain to be chosen. Multiply each density for a ‘fixed’ factor by β and each for an unfixed factor by $1 - \beta$. Summing these weighted local densities, divide by β times the number of fixed factors plus $1 - \beta$ times the number of factors yet to be fixed, to determine the global density. In this way, more weight can be placed on the interaction with fixed factors (thereby ‘looking back’ more) by setting β larger, or placed on ‘look ahead’ to unfixed factors more by setting β smaller.

In essence, schemes determining what to select based only on the fixed factors are using $\beta = 1$, while DDA as described until this point uses $\beta = 0.5$. The exponent α used in the density calculation instead aims, when large, to emphasize the selection of uncovered pairs between factors in which a large fraction of pairs remain to be covered; when α is less than one, pairs of factors that are closer to complete coverage receive higher weight instead. Both can be viewed as tuning parameters for the density calculation. Therefore, a broad spectrum of values for α and β is examined.

7.1.1. Scaling and tie-breaking experiment

The effect of scaling and tie-breaking is explored with the following values.

- Look-ahead scaling: $\alpha = 0.05, 0.10, 0.20, 0.30, 0.50, 0.75, 0.80, 0.90, 1.00, 1.25, 1.50, 1.75, 2.00, 2.50, 3.00,$ and 10.00 .



Table IX. Minimum and average size, by scaled level density, β , and five tie-breaking rules for input $7^1 6^1 5^1 4^5 3^8 2^3$ (see Section 7.1.1 for the tie-breaking rules (1)–(5)).

Look-back (β)	Tie-breaking rule									
	(1)		(2)		(3)		(4)		(5)	
	Minimum	Average	Minimum	Average	Minimum	Average	Minimum	Average	Minimum	Average
0.05	70	87.24	72	86.76	74	89.76	72	90.53	73	88.59
0.10	73	84.47	73	83.94	72	83.65	72	86.71	73	84.00
0.20	71	78.59	71	77.94	73	79.18	73	80.12	71	79.06
0.25	73	77.82	71	77.35	72	78.47	72	78.65	72	77.41
0.30	72	76.71	70	76.00	71	76.71	72	77.71	72	76.35
0.50	71	73.71	71	73.06	71	73.29	72	74.35	71	73.06
0.75	70	72.65	70	72.00	70	72.29	72	73.00	70	72.29
0.80	71	72.71	70	72.47	70	72.53	71	72.94	71	72.29
0.90	70	72.41	70	71.94	71	72.53	71	73.18	70	72.71
0.99	70	72.71	70	72.12	70	72.12	72	73.47	71	72.82

- Look-back scaling: $\beta = 0.05, 0.10, 0.25, 0.30, 0.50, 0.75, 0.80, 0.90, 0.99$.
- Tie-breaking:
 - *Tie-break (1)* resolves ties in selecting factors and levels in favor of the lexicographically first;
 - *Tie-break (2)* resolves ties by selecting levels in favor of the least-frequently used;
 - *Tie-break (3)* resolves ties to select the least-recently used;
 - *Tie-break (4)* breaks ties in favor of factors and levels for which the largest number of pairs remain to be covered;
 - *Tie-break (5)* selects at random.

First we analyse the data at a high level with an ANOVA. Size is used as the response variable and the ANOVA indicates that the choice of β is dominant (54.6% of the variance), the interaction of choices for α and β is next most significant (23.8%), the choice of α next at 16.2%, and the three-way interaction (α , β , and tie-break) explains 3.8% of the variance. Together these account for 98.5% of the variation in size, and the other factors and interactions are considered insignificant. In plain terms, emphasis in selecting parameters should be placed on the weighting for look-back versus look-ahead. Indeed looking at the data in more detail as shown in Table IX, restricting to cases with $\beta > 0.5$ is desirable. Intuitively, this supports the belief that speculative gains from factors to be fixed later are not as important as ensuring gains with factors whose levels have been chosen.

The interaction of α and β can be seen in Table X; when β is small (so that look-ahead is emphasized), choosing α larger has a significant effect on reducing size. This also is to be expected; when look-ahead is weighted heavily, increasing α places substantial weight on pairs between factors for which many pairs remain uncovered. The effect of this is less pronounced as β increases. To explore this further, attention in Table XI is restricted to $\beta \in \{0.75, 0.8, 0.9, 0.99\}$.



Table X. Minimum and average size, scaled by factor ordering density, α , and five tie-breaking rules for input $7^1 6^1 5^1 4^5 3^8 2^3$ (see Section 7.1.1 for the tie-breaking rules (1)–(5)).

Look-ahead (α)	Tie-breaking rule									
	(1)		(2)		(3)		(4)		(5)	
	Minimum	Average	Minimum	Average	Minimum	Average	Minimum	Average	Minimum	Average
0.05	72	80.50	71	78.10	72	80.20	72	81.10	71	79.30
0.10	70	79.60	70	79.00	71	80.00	72	80.90	72	79.00
0.20	72	80.80	72	79.10	71	80.10	72	81.80	71	80.20
0.25	71	80.50	71	80.60	71	80.80	71	83.00	71	81.30
0.30	72	80.30	71	79.80	71	81.00	73	82.00	70	80.50
0.50	71	78.70	72	79.40	72	81.00	72	81.30	72	80.20
0.75	72	78.40	71	78.40	71	78.00	72	79.40	71	77.50
0.80	72	77.20	71	77.20	72	77.80	72	78.90	71	77.40
0.90	70	76.80	70	77.40	70	77.40	72	78.10	70	77.20
1.00	72	77.60	70	75.80	71	77.40	72	77.50	72	77.90
1.25	72	75.90	70	73.90	71	75.50	71	76.00	71	75.70
1.50	70	74.40	71	74.70	72	74.60	73	75.30	71	74.40
1.75	71	73.90	71	73.60	71	74.00	72	75.00	71	73.30
2.00	71	73.80	71	73.10	70	73.00	72	76.00	72	73.80
2.50	71	73.40	70	72.70	71	73.40	72	73.90	72	73.00
3.00	71	72.80	71	72.90	70	72.30	72	73.50	71	73.00
10.00	70	72.70	71	72.40	71	73.40	72	73.40	71	72.90

In Table XI, what is striking is that once look-ahead is deemphasized, the value of α chosen has a very limited effect. Earlier $\alpha = 2$ was chosen, and its performance is comparable to other selections. Look-ahead should not be emphasized at the expense of look-back, or *vice versa*; if look-ahead is weighted substantially, then selecting the impact α to be larger reduces the sizes obtained. However, when look-back is emphasized, the variation in this case is too small to draw conclusions about the value of α or the tie-breaking strategy.

Tie-breaking appears to have little impact, as shown in the statistical analysis and in Tables IX–XI. Thus, it could be argued that tie-breaking is primarily useful as a technique that can be varied to produce different arrays, rather than as a means to provide significant reductions in their sizes.

These results do not suggest that an ‘optimal’ value for α and β can be chosen. Apart from this analysis, a response surface method (RSM) [44] was used in an attempt to identify global optimal values for α and β . However, a single global optimum is not likely to exist. Therefore, it is reasonable to employ numerous different values, generate a candidate test using each, and select the best. Consequently DDA is extended to create two variations in the next section: Candidate-DA and Random-DA.



Table XI. Minimum and average size when level density is scaled by $\beta \geq 0.75$, scaled by factor ordering density, α , and five tie-breaking rules for input $7^1 6^1 5^1 4^5 3^8 2^3$ (see Section 7.1.1 for the tie-breaking rules (1)–(5)).

Look-ahead (α)	Tie-breaking rule									
	(1)		(2)		(3)		(4)		(5)	
	Minimum	Average	Minimum	Average	Minimum	Average	Minimum	Average	Minimum	Average
0.05	72	73.00	71	72.75	72	73.25	72	72.75	71	72.50
0.10	70	72.75	70	71.00	71	72.00	72	72.25	72	73.00
0.20	72	73.75	72	72.50	71	72.75	72	72.50	71	72.25
0.25	71	72.50	71	73.00	71	71.50	71	72.25	72	73.25
0.30	72	73.25	71	72.25	71	71.75	73	73.50	70	71.25
0.50	71	72.50	72	72.75	72	73.75	72	74.00	72	72.75
0.75	72	72.75	71	72.75	71	72.00	72	73.50	71	72.50
0.80	72	73.25	71	72.00	72	73.00	72	73.00	71	72.50
0.90	70	72.00	70	72.00	70	71.75	72	72.50	70	71.50
1.00	72	72.75	70	71.00	71	72.25	72	72.75	73	73.75
1.25	72	73.00	70	71.25	71	72.00	71	72.00	71	72.50
1.50	70	71.25	71	72.75	72	72.50	73	73.75	71	72.50
1.75	71	71.75	71	72.50	71	72.25	72	74.25	71	71.50
2.00	71	72.25	71	72.00	70	71.00	72	74.50	72	73.00
2.50	71	71.75	70	71.25	72	73.25	73	73.50	72	72.25
3.00	71	73.00	71	72.25	70	71.50	72	72.50	72	72.75
10.00	72	73.00	71	72.25	72	73.75	72	74.00	71	73.25

7.2. Candidates and repetitions

Earlier, cases in which DDA did not match or beat the performance of AETG and TCG were attributed to the fact that TCG maintains multiple candidate solutions, and AETG performs a number of repetitions randomly. Here, density algorithms are developed to exploit candidates and repetitions in order to assess their impact on accuracy and execution time.

Candidate-DA stores M candidates during the construction of a row. Once all M candidate rows have been constructed, the candidate that covers the most new pairs is accepted. Candidates include variations of tie-breaking (take first, take lexicographically first, take least-recently used, take least-frequently used) and inflation factor α .

Random-DA stores M candidates during the construction of a row, using the same candidates as before, but also introduces a random tie-breaking rule. The entire covering array is constructed N times; since randomization has been introduced, one can expect different behaviour in each repetition.

The three density algorithms are compared. The implementations are written in C++ and run on a Dell Latitude C400, Pentium II 1200 MHz with 256 MB RAM. In DDA, take-first tie-breaking and $\alpha = 2$ is used; in Candidate-DA, 15 candidates are used with take-first, take least-recently used, and take least-frequently used tie-breaking, and inflation values of 0.5, 1.0, 1.5, and 2.0; and Random-DA uses 50 candidates and 100 covering array generation repetitions. The test suite size produced by DDA



Table XII. DDA with one candidate, DDA with multiple candidates, and DDA with both multiple candidates and repetitions (size/time in seconds).

Input	DDA		Candidate-DA <i>M</i> = 15		Random-DA <i>M</i> = 50, <i>rep</i> = 100	
	Size	Time (s)	Size	Time (s)	Size	Time (s)
6^{13}	70	0.09	67	1.32	68	68.47
6^{40}	96	2.21	93	38.2	94	10 922
10^{40}	251	9.47	245	156.3	248	461
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	93	0.04	97	0.64	91	91.9
$8^2 7^2 6^2 5^2$	74	0.03	71	0.43	70	71.54
$6^6 5^5 3^4$	58	0.07	57	1.24	56	57.85
$3^4 4^5$	25	0.01	23	0.13	24	24.49

Table XIII. DDA with one candidate, DDA with multiple candidates, and DDA with both multiple candidates and repetitions compared with previously published results for AETG, IPO, TCG, and TConfig.

Input	DDA	Candidate-DA	Random-DA	AETG	TCG	IPO	TConfig
$5^1 3^8 2^2$	21	21	20	19	20	<i>21</i>	<i>21</i>
$7^1 6^1 5^1 4^5 3^8 2^3$	43	45	44	45	45	<i>48</i>	<i>91</i>
$5^1 4^4 3^{11} 2^5$	27	31	28	30	30	28	32
$6^1 5^1 4^6 3^8 2^3$	34	37	36	34	33	35	50
$4^{15} 3^{17} 2^{29}$	35	37	37	41	35	36	40
$4^1 3^{39} 2^{35}$	27	26	28	28	27	29	30
3^{13}	18	19	17	15	20	19	<i>15</i>
2^{100}	15	15	15	10	16	15	14
4^{40}	43	44	44	42	46	49	40
4^{100}	51	54	56	<i>51</i>	55	52	43
10^{20}	201	200	198	180 (<i>198</i>)	218	212	231

can often be improved using multiple candidates and/or multiple repetitions. Data sets in Table XII show that the cost in time increases with more repetitions and candidates. In addition, data sets in Table XIII compare the minimum size results from the three versions of DDA to previously published results for AETG, TCG, IPO, and TConfig. Data that were unavailable in the previous literature for AETG, TCG, IPO, and TConfig are shown in italics in Table XIII and were derived from available algorithms that were also used in Section 1.



8. CONCLUSIONS

Cohen *et al.* [1] established that a greedy method that always selects the next test to maximize newly covered pairs gives a test suite whose size grows logarithmically with the number of factors. They designed AETG as an heuristic method to attempt to select a next test that covers close to this maximum number, but there is no theoretical guarantee that their method indeed achieves this objective (although the practical results suggest that it typically does). TCG employs a deterministic strategy that often, but not always, outperforms AETG for accuracy but also provides no such guarantee. Indeed, of the methods discussed in this paper, only the recursive combinatorial method TConfig provides such a guarantee; however, TConfig appears to perform well in practice on a limited variety of inputs.

These observations motivate two questions. The first is to select a test that covers the maximum number of uncovered pairs, and in a strong sense an efficient algorithm to do this is unlikely to exist; the problem is NP-hard. The second question is to determine what is needed to get a logarithmic bound on the number of tests in a test suite. Employing the method of Cohen *et al.* [1], it suffices to cover the *average* number of uncovered pairs. A density calculation has been developed, and used to ensure that at every stage, the selection of factors and levels is made that ensures that the average number of pairs covered among the candidate tests that remain at each stage never declines. Consequently, when all factors have been assigned levels, the test covers at least the average number of new pairs. This guarantees the logarithmic bound.

While the logarithmic bound is of theoretical interest, and answers a question implied in [1], the practical value of the method that results is not addressed by this theoretical bound. Therefore, a deterministic density algorithm (DDA) for generating test suites has been developed; instead of choosing factors and levels so as not to decrease the density, a ‘steepest ascent’ technique is chosen, at each stage making selections to increase the density as much as possible. In a comparison of DDA with four published methods, it is competitive with respect to reported sizes of test suites. One might argue that it should be more than ‘competitive’; however, the essential point is that it provides a worst-case guarantee while enjoying the same benefits as greedy algorithms already in existence. In addition, there is with no apparent degradation in average-case performance.

Moreover, the theory underlying DDA indicates that a wide variety of candidate selections can be made without losing the logarithmic bound. Varying the focus on look-ahead versus look-back suggests that concentrating on look-back, while still retaining look-ahead, reduces array size. On the other hand, varying the definition of density to more heavily weight those pairs between factors having many uncovered pairs is most effective when look-ahead is itself emphasized; when look-back is emphasized, varying the impact α is less valuable, although perhaps still worthwhile. Further, the manner in which tie-breaking is done does not appear to be significant.

It is also possible to randomly select one of the choices that does not decrease density, or to maintain a set of candidate partial tests for exploration. AETG uses both of these techniques [1] and TCG uses the latter [37]. A randomized approach based on densities has been shown to provide an improvement in accuracy. The value of maintaining multiple candidates has also been established.

ACKNOWLEDGEMENTS

This research is supported by the Consortium for Embedded and Internetworking Technologies and by ARO grant DAAD 19-1-01-0406. Preliminary work on DDA appeared in [36].



REFERENCES

1. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7):437–444.
2. Dunietz IS, Ehrlich WK, Szablak BD, Mallows CL, Iannino A. Applying design of experiments to software testing. *Proceedings of the International Conference on Software Engineering (ICSE '97)*, Boston, MA, 1997. ACM Press: New York, 1997; 205–215.
3. Williams AW, Probert RL. A practical strategy for testing pair-wise coverage of network interfaces. *Proceedings of the 7th International Symposium on Software Reliability Engineering*, San Jose, CA, 2000. IEEE Computer Society Press: Piscataway, NJ, 2000; 246–254.
4. Yu L, Tai KC. In-parameter-order: A test generation strategy for pairwise testing. *Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium*, Washington, DC, 1998. IEEE Computer Society Press: Piscataway, NJ, 1998; 254–261.
5. Dalal SR, Karunanithi AJN, Leaton JML, Patton GCP, Horowitz BM. Model-based testing in practice. *Proceedings of the International Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, 1999. ACM Press: New York, 1999; 285–294.
6. Berling T, Runeson P. Efficient evaluation of multifactor dependent system performance using fractional factorial design. *IEEE Transactions on Software Engineering* 2003; **29**(9):769–781.
7. Burr K, Young W. Combinatorial test techniques: Table-based automation, test generation and code coverage. *Proceedings of the International Conference on Software Testing Analysis and Review*, Orlando, FL, 1998; 503–513.
8. Lazić LJ, Velašević D. Applying simulation and design of experiments to the embedded software testing process. *Software Testing, Verification and Reliability* 2004; **14**:257–282.
9. Kuhn D, Reilly M. An investigation of the applicability of design of experiments to software testing. *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, Greenbelt, MD, 2002. IEEE Computer Society Press: Piscataway, NJ, 2002; 91–95.
10. Kuhn R, Wallace D, Gallo A. Software fault interactions and implications for software testing. *IEEE Transactions of Software Engineering* 2004; **30**(6):418–421.
11. Wong WE, Horgan JR, London S, Mathur AP. Effect of test set minimization on fault detection effectiveness. *Proceedings of the 17th IEEE International Conference on Software Engineering*, Limerick, Ireland, 2000. ACM Press: New York, 2000; 41–50.
12. Memon AM, Soffa ML. Regression testing of GUIs. *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, 1–5 September 2003. ACM Press: New York, 2003; 118–127.
13. White L. Regressing testing of GUI event interaction. *Proceedings of the International Conference on Software Maintenance*, Monterey, CA, 1996. IEEE Computer Society Press: Piscataway, NJ, 1996; 350–358.
14. White L, Almezen H. Generating test cases for GUI responsibilities using complete interaction sequences. *Proceedings of the Interactional Symposium on Software Reliability Engineering*, San Jose, CA, 2000. IEEE Computer Society Press: Piscataway, NJ, 2000; 110–121.
15. Yilmaz C, Cohen MB, Porter A. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering* 2006; **31**(1):20–34.
16. Mandl R. Orthogonal Latin squares an application of experiment design to compiler testing. *Communications of the ACM* 1985; **28**(10):1054–1058.
17. Berry RF. Computer bench mark evaluation and design of experiments a case study. *Proceedings of the IEEE Wireless Communications Networking Conference (WCNC03)*. *IEEE Transactions on Computers* 1992; **41**(10):1279–1289.
18. Chateaufneuf MA, Kreher DL. On the state of strength-three covering arrays. *Journal of Combinatorial Designs* 2002; **10**(4):217–238.
19. Moura L, Stardom J, Stevens B, Williams A. Covering arrays with mixed alphabet sizes. *Journal of Combinatorial Designs* 2003; **11**(6):113–132.
20. Stardom J. Metaheuristics and the search for covering and packing arrays. *Master's Thesis*, Simon Fraser University, 2001.
21. Williams AW, Probert RL. A measure for component interaction test coverage. *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, Beirut, Lebanon, 2000. ACS/IEEE, 2000; 301–311.
22. Bryce R, Colbourn CJ. Prioritized interaction testing for pairwise coverage with seeding and avoids. *Information and Software Technology Journal* 2006; **48**(10):960–970.
23. Cohen MB, Colbourn CJ, Ling ACH. Augmented simulated annealing to build interaction test suites. *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)*, White Plains, NJ, 2003. IEEE Computer Society Press: Piscataway, NJ, 2003; 394–405.
24. Cohen MB, Colbourn CJ, Ling ACH. Constructing strength 3 covering arrays with augmented annealing. *Discrete Mathematics*, to appear.



25. Hartman A. Software and hardware testing using combinatorial covering suites. *Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms*, Golumbic M (ed.). Springer: Berlin, 2005; 237–266.
26. Hartman A, Raskin L. Problems and algorithms for covering arrays. *Discrete Mathematics* 2004; **284**:149–156.
27. Hedayat A, Sloane N, Stufken J. *Orthogonal Arrays*. Springer: New York, 1999.
28. Sloane N. Covering arrays and intersecting codes. *Journal of Combinatorial Designs* 1993; **1**(1):51–63.
29. Cohen MB, Colbourn CJ, Collofello JS, Gibbons PB, Mugridge WB. Variable Strength Interaction Testing of Components. *Proceedings of the 27th International Computer Software and Applications Conference (COMPSAC 2003)*, Dallas, TX, 2003. IEEE Computer Society Press: Piscataway, NJ, 2003; 413–418.
30. Cohen MB, Colbourn CJ, Gibbons PB, Mugridge WB. Constructing test suites for interaction testing. *Proceedings of the International Conference on Software Engineering (ICSE 2003)*, Portland, OR, 2003. ACM Press: New York, 2003; 38–48.
31. Nurmela K. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics* 2004; **138**:143–152.
32. Cheng C, Dumitrescu A, Schroeder P. Generating small combinatorial test suites to cover input–output relationships. *Proceedings of the 3rd International Conference on Quality Software (QSIC '03)*, Dallas, TX, 2003. IEEE Computer Society Press: Piscataway, NJ, 2003; 76–82.
33. Dumitrescu A. Efficient algorithms for generation of combinatorial covering suites. *Proceedings of the 14th Annual International Symposium on Algorithms and Computation (ISAAC '03)*, 2003 (*Lecture Notes in Computer Science*, vol. 2906). Springer: Berlin, 2003; 300–308.
34. Kobayashi N, Tsuchiya T, Kikuno T. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters* 2002; **8**:85–91.
35. Bryce RC, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction tests. *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, MO, 2005. ACM Press: New York, 2005; 146–155.
36. Colbourn CJ, Cohen MB, Turban RC. A deterministic density algorithm for pairwise interaction coverage. *Proceedings of the IASTED International Conference on Software Engineering*, Vienna, Austria, 2004. IASTED, 2004; 245–252.
37. Tung YW, Aldiwan WS. Automating test case generation for the new generation mission software system. *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT, 2000. IEEE Computer Society Press: Piscataway, NJ, 2000; 431–437.
38. Cohen DM, Dalal SR, Parelius J, Patton GC. The combinatorial design approach to automatic test generation. *IEEE Software* 1996; **13**(5):83–88.
39. Tai KC, Yu L. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 2002; **28**(1):109–111.
40. Williams AW. Determination of test configurations for pair-wise interaction coverage. *Proceedings of Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000)*, Ottawa, Canada, 2000; 59–74.
41. Colbourn CJ. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)* 2004; **58**:121–167.
42. Seroussi G, Bshouty N. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory* 1988; **34**:513–522.
43. Cohen DM, Dalal SR, Fredman ML, Patton GC. Method and system for automatically generating efficient test cases for systems having interacting elements. *United States Patent Number 5,542,043*, 1996.
44. Montgomery DC. *Design and Analysis of Experiments* (5th edn). Wiley: New York, 2001.