One-Test-at-a-Time Heuristic Search for Interaction Test Suites

Renée C. Bryce
Computer Science
University of Nevada at Las Vegas
Las Vegas, Nevada
reneebryce@cs.unlv.edu

Charles J. Colbourn
Computer Science
Arizona State University
Tempe, Arizona
colbourn@asu.edu

ABSTRACT

Algorithms for the construction of software interaction test suites have focussed on the special case of pairwise coverage; less is known about efficiently constructing test suites for higher strength coverage. The combinatorial growth of t-tuples associated with higher strength hinders the efficacy of interaction testing. Test suites are inherently large, so testers may not run entire test suites. To address these problems, we combine a simple greedy algorithm with heuristic search to construct and dispense one test at a time. Our algorithm attempts to maximize the number of t-tuples covered by the earliest tests so that if a tester only runs a partial test suite, they test as many t-tuples as possible. Heuristic search is shown to provide effective methods for achieving such coverage.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Experimentation

Keywords

covering arrays, great flood, heuristic search, hill climbing, simulated annealing, tabu search, t-way interaction coverage, software interaction testing, test suite prioritization

1. INTRODUCTION

Software testing is an expensive but imperfect process. Software testers often test for defects that they anticipate while less foreseen defects are overlooked. Systematic approaches to testing have been suggested to complement current testing methods in order to improve rates of fault detection. One such systematic testing technique is software interaction testing [1, 2, 6, 15, 17, 20, 21, 30].

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

A *software interaction test suite* of *strength* t is an $N \times k$ array with N rows each representing a *test*, k columns each representing a *factor*, and v number of symbols permitted in each column representing allowed *levels* of the factor. Each t-tuple (i.e. selection of t columns, and one of the v levels for each of the t corresponding factors) occurs at least once. The size, t, of a tuple is referred to as the strength. *Higher strength* is the term when $t > 2$. Table 1(a) shows four *factors* that each have three *levels*. The factors (f_0,f_1,f_2, and f_3) represent components, or parameters for a system. The numerical values for levels map to unique options for the parameters of a system. An exhaustive test suite would have 81 ($= 3^4$) tests. Pairwise interaction testing tests each 2-way interaction at least once. This requires only 9 tests as in Table 1(b).

f_0	f_1	f_2	f_3
0	3	6	9
1	4	7	10
2	5	8	11

(a)

	f_0	f_1	f_2	f_3
1	0	3	6	9
2	0	4	7	10
3	0	5	8	11
4	1	3	8	10
5	1	4	6	11
6	1	5	7	9
7	2	3	7	11
8	2	4	8	9
9	2	5	6	10

(b)

Table 1: (a) Example system of four components with three levels each. (b) A pairwise interaction test suite

The example provides tests for all pairwise interactions. Pairwise testing has been useful in several applications (see [13] and therein). Interaction testing techniques can also be applied with higher strength interaction coverage. The most cost-effective value for the strength, t, is unknown. Limited research shows that strength two is often not sufficient and that strengths up to six can be beneficial. For instance, Kuhn *et al.* compare reported bugs for the Mozilla web browser against results of interaction testing [20]. More than 70% of bugs are identified with 2-way interactions; approximately 90% of bugs are identified with 3-way interactions; and 95% of errors are detected by 4-way interactions. This study is one of only a few that evaluates beyond the special case of pairwise coverage. (Also see [15, 21] for two other studies on higher strength interaction testing.)

Most studies focus on pairwise coverage. Available tools work well to construct pairwise test suites, but the results

for higher strength test suites are generally unacceptably large and take significant time to generate. The problem is NP-hard (see [4], for example). Pairwise coverage has been implemented with greedy methods [2, 5, 9, 26, 27], algebraic methods [8, 13, 18, 29], constraint programming [19], and heuristic search [11, 22, 25]. For higher strength, simulated annealing has reported results for up to 3-way coverage [12]; Combinatorial Test Services (CTS) is implemented for any t-way coverage with results published for up to 4-way coverage [18]; the Automated Efficient Test Generator (AETG) can be used for any t-way coverage, but results are reported for up to 3-way coverage [7]; and mathematical solutions have been published for any t-way coverage for fixed-level covering arrays [13]. A genetic algorithm and an ant colony algorithm are implemented for up to 3-way coverage [24].

	$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	3^{13}	11^{16}
t=2	1,320	702	14,520
t=3	18,150	7,722	745,360
t=4	157,773	57,915	26,646,620
t=5	902,055	312,741	703,470,768
t=6	3,416,930	1,250,954	1,301,758,600
...	...	...	...
t=k	3,628,800	1,594,323	45,949,729,863,572,200

Table 2: Increase in number of tuples with higher strength (t) coverage

Generating covering arrays of higher strength consumes more computational resources and produces inherently larger solutions than pairwise coverage. Table 2 shows the combinatorial growth of tuples for three different combinations of factors and levels as the strength t increases. The input 3^{13} (read as 13 factors have 3 levels each) includes 702 pairs, 7,722 triples, and reaches over a million 6-tuples. As the size of the tuples and the numbers of them increase, the size of test suites grow as well. Coping with this growth to minimize test suite size within reasonable execution time has not been well addressed in general. More importantly, *if a test suite is inherently large and a tester can not run the entire test suite, how should one prioritize tests?* This work proposes a straightforward solution to prioritizing tests. Arguably, the real goal is not to minimize the number of tests to achieve t-way coverage. Rather it is to generate and dispense one test at a time so that many t-tuples are covered as early as possible. Then if a tester stops testing at any time, they nevertheless cover a "large" fraction of the t-tuples. The trade-off in execution time and test suite size is a serious issue. A tester may prefer quicker turn-around over a solution that may cover more t-tuples, or may be willing to wait longer for a test while running other tests. Therefore, we develop a hybrid approach to combine the speed of a greedy method with the potential improvement in test suite size from a heuristic search technique. We discuss this contribution shortly.

In Section 2, we demonstrate that the very natural idea of first constructing a test suite of small size, and then ordering the tests to obtain early coverage of t-tuples, may not perform as well as the approach that we propose. Section 3 describes a hybrid technique to generate tests using a one-test-at-a-time greedy method and heuristic search. Section 4 provides empirical results on the hybrid technique. In these experiments, our goal is not to identify the best instantiation of the hybrid technique, but rather to examine possible instantiations that serve as a proof-of-concept that the hybrid technique can generate solutions that have a higher rate of t-tuple coverage than either a greedy or heuristic search algorithm alone. Finally, Section 5 examines the hybrid technique based on a different greedy algorithm and Section 6 compares to an exhaustive one-test-at-a-time algorithm.

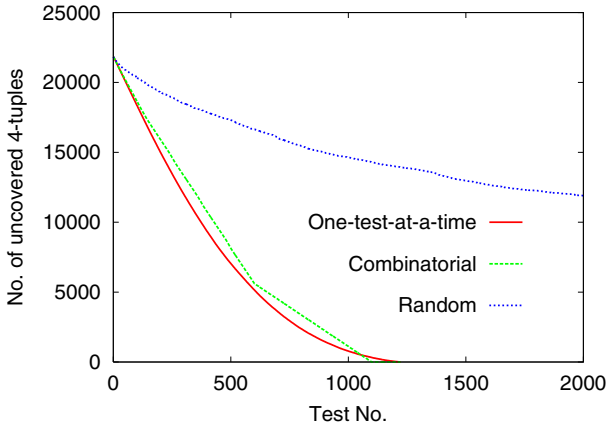


Figure 1: Rate of 4-tuple coverage for input 5^7.

2. MINIMUM TEST SUITE SIZE OR EARLY COVERAGE?

Is minimum test suite size strongly correlated with coverage of many t-tuples in the initial tests? If it is, rather than generating a test suite one test at a time to maximize early coverage of t-tuples, one could use published methods to generate a test suite with as few tests as possible, and reorder the tests to obtain early coverage. However, we show that smaller test suite size may not support early coverage of t-tuples, no matter what ordering is chosen.

In order to do this, we first explicitly describe a test suite with 1100 tests for strength four of type 5^7 that is smaller than any previously published test suite for these parameters. The first 600 tests are formed by taking $(a, a+b+c+d, a+2b+4c+3d, a+3b+4c+2d, a+4b+c+4d, d, d)$ for $a, b, c, d \in \{0, 1, 2, 3, 4\}$ and either $c \neq 0$ or $d \neq 0$, with all arithmetic done modulo 5. The last 500 tests are formed by taking $(a, a+b, a+2b, a+3b, a+4b, c, d)$ for $a, b, c, d \in \{0, 1, 2, 3, 4\}$ and $c \neq d$, with all arithmetic done modulo 5. It is possible to check that this covers all of the 21,875 4-tuples through exhaustive examination. On the other hand, a simple greedy one-test-at-a-time algorithm [3] provides a solution of size 1,222. The ultimate sizes of the test suites are strikingly different! Figure 1 shows the rate of 4-tuple coverage for the two test suites. The greedy test suite appears to be much more effective at covering 4-tuples early on; whenever the number of tests chosen is between 25 and 1,054, it covers more 4-tuples. After 775 tests, the greedy method has covered 1,000 more 4-tuples.

Can one reorder the test suite with 1100 tests to improve coverage of the 4-tuples early? Naturally, trying all $1,100!$ orderings is infeasible. Despite this, we can show that *no*

ordering of this suite is as good as the much larger greedy solution. To do this, call a particular 4-tuple *private* to a test if the test is the unique one that covers the 4-tuple. Every test should have at least one private 4-tuple, or the test can be removed. Often, tests have more than one. Indeed a simple but lengthy computation shows that of the 1,100 tests, 100 have 25 private 4-tuples each, 500 have 21 each, and 500 have 10 each. Now consider an *arbitrary* ordering of the 1,100 tests. Whenever a test is not in the initial set of x tests, each of its private 4-tuples must be uncovered. Hence after the first x tests in this ordering, no more than $21,875-10(1,100-x)-11\max(600-x,0)-4\max(100-x,0)$ 4-tuples can be covered, whatever ordering is chosen. This bound suffices to show that the greedy test suite covers more 4-tuples than *every* ordering of the much smaller test suite whenever $436 \leq x \leq 1,045$.

Small test suite size does not ensure early coverage of tuples, even when the test suite can be reordered arbitrarily. Indeed, in Section 6, it is shown that the converse also does not hold: Early coverage does not guarantee the smallest overall test suites.

One might further suspect that a simple method that selects tests uniformly at random would afford good coverage among initial tests. However, Figure 1 shows that selection of tests uniformly at random from the set of all possible tests, except when very few tests are chosen, is not at all competitive. Because the objective of early coverage diverges from that of minimum test suite size, and random selection does not provide a sensible alternative, we treat methods that generate one test at a time, attempting to choose a best next test at each step.

3. A HYBRID TECHNIQUE

Although the algorithms developed herein ultimately cover all t-way interactions, the focus is on covering as many t-tuples as possible in the earliest tests. Consider the scenario in which tests are generated on demand. The approach is straightforward: Use a one-test-at-a-time greedy algorithm to initialize tests and then apply heuristic search to increase the number of t-tuples in a test. A greedy method can initialize a test quickly; then heuristic search can attempt to increase the number of t-tuples in a test for as long as time permits. Indeed, testers may not run an entire test suite, so rather than spending time to generate an entire test suite, time may be spent to increase the number of t-tuples covered in the subset of the test suite that is actually run.

3.1 An algorithm for test initialization

Existing one-test-at-a-time greedy algorithms for constructing covering arrays fall into a framework [5]. With small modification, this framework can be employed here. The first portion of Figure 2 shows the greedy component of the hybrid test generation technique, giving three major decision points [5]. For each test, a number M of ***candidate tests*** is constructed. The candidate test that covers the most new t-tuples is chosen. Within the construction of a single test, ***factor ordering*** is the order in which factors are assigned levels. For each factor, a ***level selection*** rule specifies criteria for assigning a level to a factor.

Any one-test-at-a-time greedy algorithm may be used with the heuristic search algorithm that we develop. For our preliminary experiments, we use a specific instantiation of the framework as follows. Only one candidate test is constructed each time. We begin the construction of a test by selecting a t-tuple that has not yet been covered. This ensures that any test that we generate covers at least one previously uncovered t-tuple. Each remaining factor is assigned a level. The order in which factors are assigned levels is random. A factor that has been assigned a level is referred to as *fixed*; one that hasn't as *free*. For each factor, the level that covers the largest number of previously uncovered t-tuples in relation to fixed factors is selected. This algorithm is similar to AETG [10].

```
// Select a test with an n-way greedy algorithm
for M candidate tests
   choose an uncovered t-tuple T at random
   fix factors of T to the specified values; other factors are free
   while a free factor remains
      using a factor selection rule select a free factor f
      count t-tuples covered containing each level of factor f
      use a level selection rule to pick a level ℓ for f
      fix factor f to level ℓ
   end while
end for
store BestTest, the candidate test covering the most new t-tuples
// Refine the test using heuristic search
for μ iterations do
   select a factor at random
   select a new level for the factor at random
   accept the change according to an acceptance criterion
   if the change produces a test that covers the
      most t-tuples seen, store it as BestTest
end for
return BestTest
```

Figure 2: Pseudocode to generate a test - each test is initialized with a greedy algorithm and refined with heuristic search.

3.2 One-test-at-a-time search

The greedy algorithm initializes tests rapidly. For pairwise coverage greedy methods can be competitive, yet heuristic search often yields the smallest covering arrays at the cost of higher execution times [28]. For instance, a comparison of a greedy method and simulated annealing for limited time periods shows that simulated annealing often produces the best results, but typically only after longer execution time [28].

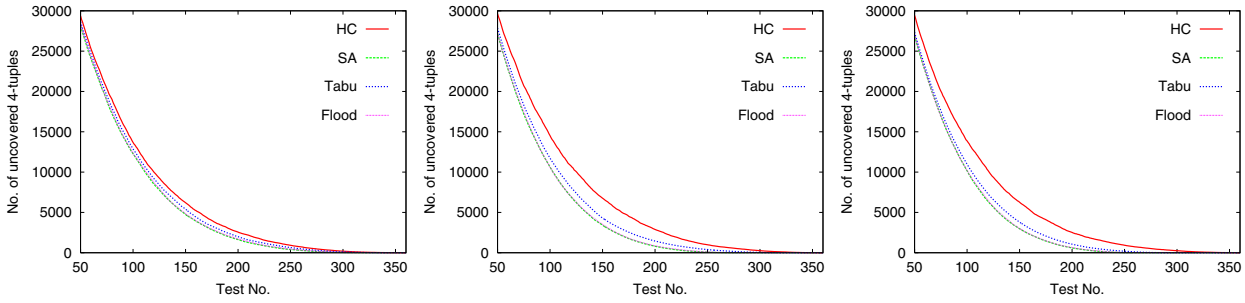
We apply search to individual tests with hill climbing, simulated annealing, tabu search, and great flood. All of the search strategies implemented here have the same goal, to maximize the number of t-tuples covered in a test. Each test is initially chosen using the greedy algorithm and then modified using local search. During the search iterations, one factor is selected at a time using a *factor selection rule*. Random first-order improvement (for factor selection) is used to diversify the search. For each factor selected, a level is assigned using a *level selection rule*. For each of these possible tests, neighborhood pruning can be effected by using *multiple candidate tests* for each test that is added. Choices of candidates are avoided here in order to focus on the impact of search iterations. Figure 2 shows pseudocode of this process which we instantiate using four heuristic search techniques.

3.2.1 Hill climbing

Using hill climbing [23], a test goes through a series of

	Greedy	HC	SA	Tabu	Flood
	No. iterations	No. iterations	No. iterations	No. iterations	No. iterations
Test No.	0	1,000	1,000	1,000	1,000
25	41,692	41,629	40,719	40,947	40,845
50	29,365	29,460	26,895	27,359	27,007
75	20,497	20,056	16,989	17,634	16,952
100	14,039	13,868	10,165	10,986	10,187
125	9,383	9,364	5,641	6,595	5,724
150	6,271	6,267	2,976	3,795	3,002
175	4,154	4,101	1,453	2,075	1,488
225	1,545	1,592	180	492	214
250	916	932	10	188	13
275	484	523	0	32	0
300	201	252	0	0	0
325	59	81	0	0	0
350	6	12	0	0	0

Table 3: Rate of 4-way coverage for input 3^{13}.



(a) 10 iterations (b) 100 iterations (c) 1,000 iterations

Figure 3: Rate of remaining 4-tuples for input 3^{13}

transformations. A factor and its level are selected at random. The cost of the move is measured as the number of t-tuples not covered in the test. If the cost $c(S') \leq c(S)$, then the transformation from S to S' is accepted. We record a final test that covers at least as many t-tuples as the initial test.

3.2.2 *Simulated Annealing*

Simulated annealing applies transformations to single tests. Factors and their levels are selected at random. To determine whether to accept the change, the cost is measured and accepted according to a temperature schedule. The current cost is the number of t-tuples not covered. If the cost $c(S) \leq c(S')$, then the transformation from S to S' is accepted. If this inequality is not satisfied, a cooling schedule is applied. The transformation is accepted with probability based on a temperature, T. During the search iterations, we record the test that covers the largest number of t-tuples and record this best test once the iterations are complete. This ensures that a final test recorded never covers fewer t-tuples than the best test encountered during the search iterations. Simulated annealing has been applied to generate covering arrays in [11].

3.2.3 *Tabu search*

In this implementation of tabu search, transformations are applied to tests. In a single iteration, we select one factor at random and a level for it at random. Cost is again the number of uncovered t-tuples. Moves are only permitted when not *tabu*. A tabu move is one that has occurred during the last T moves, where T is the length of history maintained. A tabu list of length $T = 10$ is recommended in [22], however, we experiment with list sizes that are proportional to the size of the inputs (ie: list sizes are a percentage of the number of levels for an input). Throughout the transformations, we record a test that covers the largest number of t-tuples; at the end of the search iterations, we select this "best test" encountered. Tabu search has been applied to generate complete covering arrays in [22, 25].

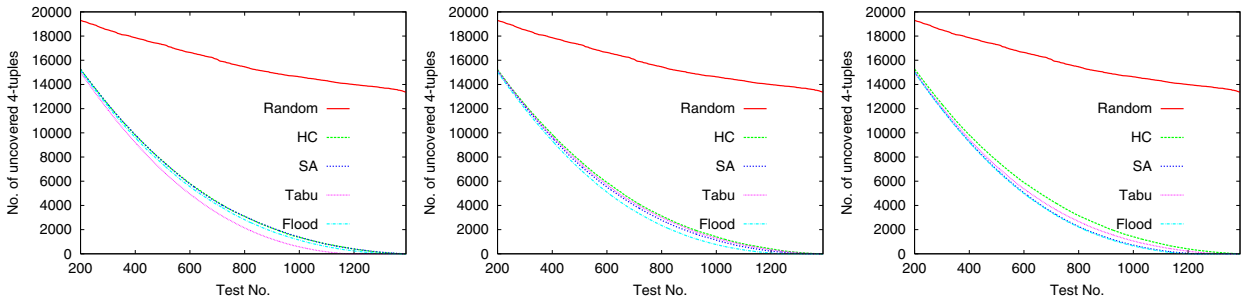
3.2.4 *Great Flood*

The Great Flood (or "Great Deluge") algorithm was introduced in [16]. In our implementation, transformations are applied to individual rows. In each iteration, a factor and its level are selected at random. Cost is now the number of covered t-tuples. Moves are only permitted when the cost does not fall below a *rising threshold*. Throughout the transformations, we record the "best test" encountered during the search iterations as one that covers the largest number of t-tuples, and ultimately report this test.

4. EXPERIMENTS

Our initial experiments explore two basic questions. Do increased search iterations improve the rate of t-tuple coverage over a greedy algorithm alone? How do the four search techniques compare?

To address these, we develop a set of experiments that serve as a proof-of-concept for our hybrid technique. A greedy algorithm (with one candidate) initially generates each test. Heuristic search then refines each test using 10,



(a) 10 iterations (b) 100 iterations (c) 1,000 iterations

Figure 4: Rate of remaining 4-tuples for input 5^7

	Greedy	HC	SA	Tabu	Flood
	No. iterations:	No. iterations:	No. iterations:	No. iterations:	No. iterations:
Test No.	0	1,000	1,000	1,000	1,000
25	47,357	45,010	44,420	45,049	44,422
50	29,766	25,764	24,625	25,580	24,491
75	19,724	15,485	14,438	15,318	14,340
100	13,143	9,496	8,522	9,425	8,493
150	6,051	3,822	3,029	3,734	3,079
200	2,786	1,510	1,065	1,480	1,059
250	1,146	566	345	543	333
300	376	177	101	151	93
350	96	28	16	11	8

Table 4: Rate of 4-way coverage for input $2^{10}3^34^25^1$.

100, or 1,000 iterations. The following heuristic search techniques are initially explored:

1. Hill climbing: If the cost $c(S') \leq c(S)$, then the transformation from S to S' is accepted. This permits sideways moves.

2. Simulated annealing: We use an initial temperature of 10% of the total number of levels and the cooling schedule reduces by 1 degree for every 10% of the number of iterations specified. Fewer iterations mean faster cooling. At the end of the search iterations, the best test encountered is dispensed.

3. Tabu search: Tabu lists have size equal to 25% of the number of levels. The tabu list does not always include an even distribution of levels for all factors.

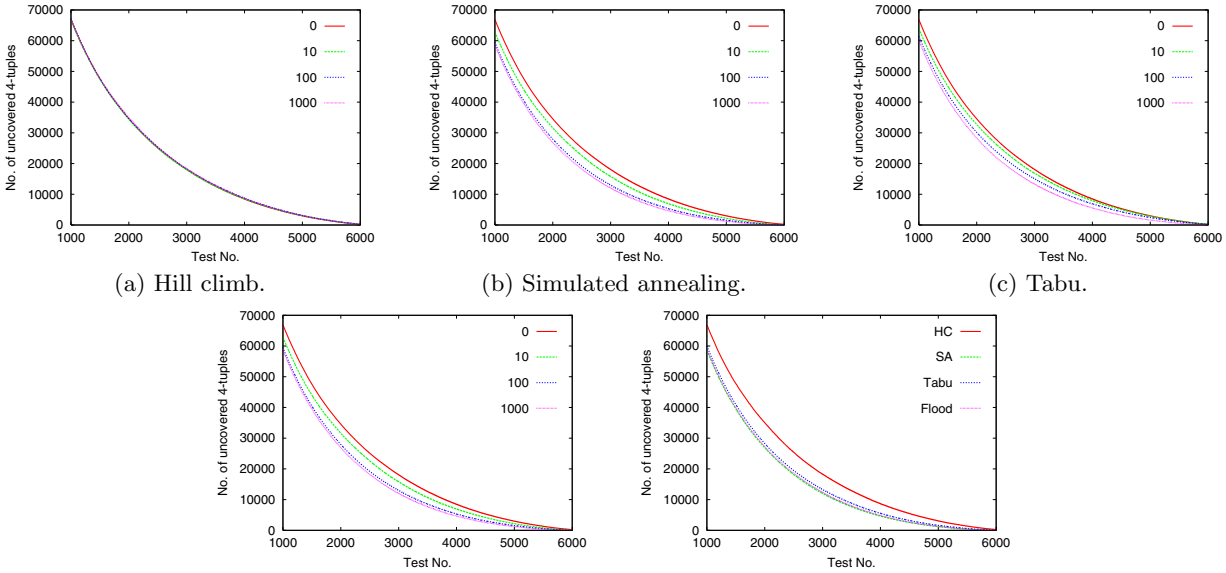
4. Great flood: The rising threshold is 90% of the best solution encountered after the first iteration, and raises at a period of once every 10% of the number of iterations. The increases are to 95%, 98%, 99%, and 100%. We incorporate a tabu list of size 25% of the total number of levels. Larger inputs maintain longer tabu lists. As before, the best test encountered is dispensed.

In our experiments, we use two inputs that have factors with equal numbers of levels and two inputs have mixed numbers of levels. These inputs include: 5^7, 3^{13}, $2^{10}3^34^25^1$, and $10^19^18^17^16^15^14^13^12^11^1$. The experiments are run five times each and we report the average of the runs. We caution the reader that we change the starting points for the x-axis when we graph the results in the experiments to emphasize the differences in the results.

Results for input 3^{13} in Figure 3 and Table 3 show that increased search iterations appear to improve the rate of t-tuple coverage. Simulated annealing has the fastest rate of t-tuple coverage among the four search techniques, followed by the great flood. Tabu search has the slowest rate of coverage, followed by hill climbing.

Figure 4 shows the number of uncovered 4-tuples for input 5^7, along with the rate of 4-tuple coverage from tests generated at random. The tests from the hybrid algorithms cover all 4-tuples in approximately 1,200 tests. Increased search iterations appear to improve the rate of 4-tuple coverage for all four search instantiations. Simulated annealing often produces the quickest rate of 4-tuple coverage. Tabu search generally produces the slowest rate of coverage with 10 or 100 search iterations. However, when 1,000 iterations are applied, tabu search ties simulated annealing or produces a slightly faster rate of 4-tuple coverage during the first 300 tests. Beyond the 300^{th} test, simulated annealing is the most competitive with 1,000 iterations. When 1,000 search iterations are applied, all four techniques maintain the same rate of 4-tuple coverage for the first 100 tests.

Table 4 shows 4-tuple coverage for input $2^{10}3^34^25^1$ and 1,000 iterations of each search technique. The first column reports the average results of running the greedy algorithm with no search iterations five times. The next four columns report results for each of the heuristic search techniques. The great flood algorithm often produces the fastest rate of t-tuple coverage, followed by simulated annealing. Hill climbing has the slowest rate of 4-tuple coverage with 1,000 iterations. Not shown in the graphs here, all four search techniques improve the rate of 4-tuple coverage when 10 or 100 iterations are applied. Simulated annealing generally has the fastest rate of coverage when 10 or 100 iterations are



(a) Hill climb. (b) Simulated annealing. (c) Tabu.

(d) Flood. (e) Comparison (100 iterations).

Figure 5: Rate of 4-tuple coverage for input $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$ with 0, 100, and 1,000 search iterations.

applied, Tabu search has the worst rate of 4-tuple coverage when 10 or 100 iterations are applied. Hill climbing and Great Flood are between these two.

Results for input $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$ in Figure 5 indicate that all four heuristic search techniques improve the rate of 4-tuple coverage. More iterations improve results. When only 10 search iterations are applied, simulated annealing and great flood produce the best results. With 10 search iterations, hill climbing has the slowest rate of 4-tuple coverage in the first 650 tests and tabu search has the slowest rate of coverage for the remaining tests. In the case of 100 iterations, simulated annealing has the fastest rate of 4-tuple coverage and tabu search has the slowest rate of coverage. Finally, with 1,000 search iterations, tabu search and hill climbing alternate in producing the slowest rate of coverage.

Each of the search techniques improve the greedy result. However, this improvement costs execution time. Table 5 shows the average time in seconds to generate single tests for the experiments run on a SunBlade 5000 machine. To compute the average time per test, we generate full test suites and divide the time by the number of tests. Having amortized the initialization time across all tests, the time to generate each individual test is impacted in a small way. However, the initialization time is relatively small. We do not report the time to generate entire test suites here since our goal is not necessarily to run an entire test suite. However, for a full test suite, increased iterations can increase or decrease execution time. Shorter overall execution time can arise when more iterations result in fewer tests.

5. USING THE DENSITY ALGORITHM

The Density Algorithm (DA) is a one-test-at-a-time greedy algorithm that appears to produce smaller size covering arrays than the greedy algorithm here [4]. We run the experiments again using DA to initialize tests to examine whether it also benefits from the search techniques. Figure 6 shows

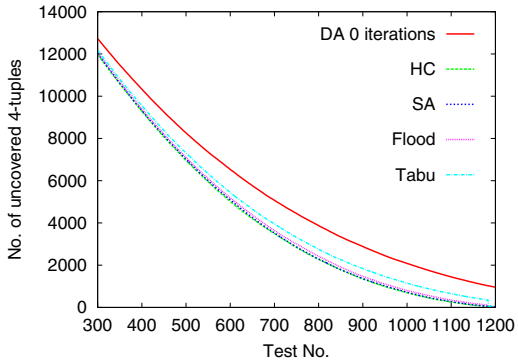


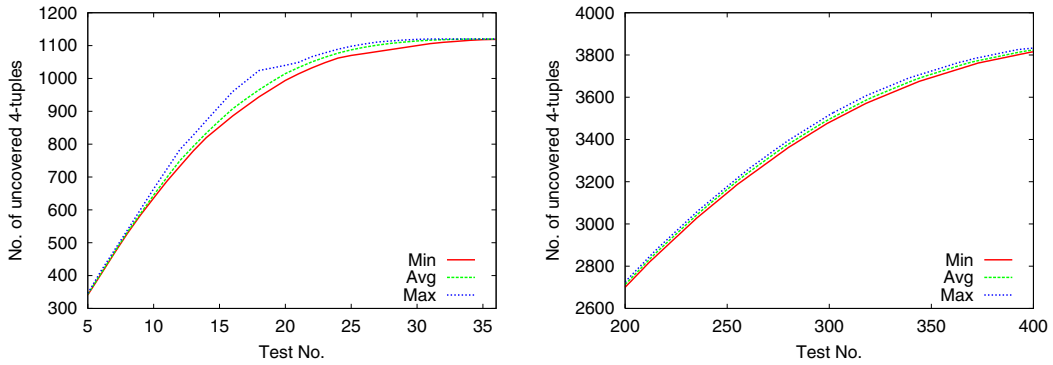
Figure 6: Rate of 4-tuple coverage for input 5^7 using the Density Algorithm and 4 search techniques.

that the heuristic search techniques also improve the rate of t-tuple coverage when tests are initialized with DA. However, the density algorithm produces a faster rate of t-tuple coverage than those earlier.

6. TEST SELECTION FROM EXHAUSTIVE CANDIDATES

At the outset, Figure 1 showed that a greedy method can produce better rates of coverage than a combinatorial method leading to a smaller complete test suite (that covers all 4-tuples). Later in Figure 4, our hybrid approach further improves the rate of coverage, but it nonetheless does not produce the smallest test suite. One may therefore hope that better heuristic search can both achieve fast coverage and produce smallest test suites. This raises a question: Can a one-test-at-a-time approach result in smallest size test suites?

To assess this, we first consider the input 2^8. We enumer-



(a) 2^8 (b) 4^6

Figure 7: The minimum, average, and maximum rate of 4-tuple coverage from a one-test-at-a-time exhaustive algorithm.

	No. of iterations	HC	SA	Tabu	Flood
$2^{10}3^34^25^1$	10	0.06	0.06	0.13	0.13
$2^{10}3^34^25^1$	100	0.21	0.21	0.26	0.24
$2^{10}3^34^25^1$	1,000	1.73	1.73	1.63	1.34
$10^19^18^17^16^1$ $5^14^13^12^11^1$	10	0.03	0.03	0.05	0.05
$10^19^18^17^16^1$ $5^14^13^12^11^1$	100	0.05	0.05	0.06	0.06
$10^19^18^17^16^1$ $5^14^13^12^11^1$	1,000	0.26	0.26	0.24	0.19
3^{13}	10	0.03	0.03	0.07	0.07
3^{13}	100	0.09	0.09	0.13	0.12
3^{13}	1,000	0.69	0.69	0.68	0.58
5^7	10	0.00	0.00	0.00	0.00
5^7	100	0.01	0.01	0.01	0.01
5^7	1,000	0.04	0.04	0.03	0.03

Table 5: Execution time per test (in seconds) using simulated annealing, tabu search, and great flood with 10, 100, and 1,000 search iterations.

ate all of the 2^8 possible tests and repeatedly select one that covers the largest number of uncovered t-tuples. We break ties uniformly at random. The construction of a test suite is repeated 100 times for input 2^8. Figure 7(a) shows the minimum, average, and maximum number of 4-tuples covered. The smallest known test suite for this input and 4-way coverage is 24 [14]. The ultimate size of our test suites here vary from 31 to 37.

For input 4^6, the first 88 tests cover the same minimum, average, and maximum number of 4-tuples. After this, the results differ. The ultimate sizes range from 408 to 425 tests.

Comparing to the best known results [14], any one-test-at-a-time approach that attempts to maximize (or indeed, maximizes) the number of t-tuples covered in tests may not produce the smallest test suite. It appears that improving rate of coverage is essentially different from generating small test suites. In retrospect, the inability of the hybrid approach to produce smallest test suites is not surprising, when even optimal selections of each row do not achieve minimum test suites. We reiterate that this is not the problem addressed; rather rate of coverage is the major concern.

7. CONCLUSIONS

Interaction testing provides a systematic approach to testing. The higher the strength of interaction coverage, the closer the testing is to exhaustive. However, higher strength testing is constrained by testing resources. Smaller sized test suites are only of full value when testers run all tests, yet testing budgets can prevent this. To address this, a one-test-at-a-time approach is introduced to cover more t-way interactions in earlier tests. Then while a tester is running other tests, heuristic search techniques attempt to cover more interactions in tests being constructed. A one-test-at-a-time greedy algorithm augmented with heuristic search is used to generate tests that have a high rate of t-tuple coverage. This approach combines the speed of greedy methods with the slower but more accurate heuristic search techniques. The hybrid approach seems to have a more rapid convergence of t-tuple coverage than either greedy or heuristic search alone. Among four heuristic search techniques examined, hill-climbing is effective only when time is severely constrained, but tabu search, simulated annealing, and the great flood make worthwhile further improvements over a longer time. While intended for testers who only run a partial test suite, we also compare the results to an algorithm that focusses on generating "small" test suites (that are intended to be run to completion). The striking conclusion is that smallest test suites do not ensure rapid coverage and that rapid coverage does not lead to smallest test suites. Indeed the algorithms focus on two different goals, each addressing a genuine need in practical testing.

8. REFERENCES

[1] R. C. Bryce, Y. Chen, and C. J. Colbourn. Biased covering arrays for progressive ranking and composition of web services. *International Journal of Simulation and Process Modeling*, to appear.

[2] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and avoids. *Information and Software Technology Journal (IST, Elsevier)*, 40(10):960–970, Oct. 2006.

[3] R. C. Bryce and C. J. Colbourn. A density-based greedy algorithm for higher strength covering arrays. submitted for review.

[4] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Journal of Software Testing, Verification, and Reliability*, to appear.

[5] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction tests. In *Intl. Conference on Software Engineering (ICSE)*, pages 146–155, May 2005.

[6] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Intl. Conference on Software Testing Analysis and Review*, pages 503–513, Oct. 1998.

[7] M. Chateauneuf and D. L. Kreher. On the state of strength-three covering arrays. *J. Combin. Des.*, 10(4):217–238, 2002.

[8] C. Cheng, A. Dumitrescu, and P. Schroeder. Generating small combinatorial test suites to cover input-output relationships. In *Intl. Conference on Quality Software (QSIC)*, pages 76–82, Nov. 2003.

[9] 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 Trans. on Software Engineering*, 23(7):437–44, Oct. 1997.

[10] 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, Oct. 1996.

[11] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Intl. Conference on Software Engineering (ICSE)*, pages 28–48, May 2003.

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

[13] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 58:121–167, 2004.

[14] C. J. Colbourn. Covering array tables, July 2006. `public.asu.edu/~ccolbou/src/tabby/catable.html`, accessed on January 15, 2007.

[15] S. R. Dalal, A. Karunanithi, J. Leaton, G. Patton, and B. M. Horowitz. Model-based testing in practice. In *Intl. Conference on Software Engineering,(ICSE)*, pages 285–294, May 1999.

[16] G. Dueck. New optimization heuristics - the great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104(1):86–92, Jan. 1993.

[17] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Intl. Conference on Software Engineering*, pages 205–215, Oct. 1997.

[18] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math.*, 284(1-3):149–156, Jul. 2004.

[19] B. Hnich, S. Prestwich, and E. Selensky. Constraint-based approaches to the covering test problem. *Lecture Notes in Computer Science*, 3419(1):172–186, Mar. 2005.

[20] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In 27^{th} *Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, Oct. 2002.

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

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

[23] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Chapter 4, 1995.

[24] Toshiaki Shiba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Intl. Conference on Computer Software and Applications Conference (COMPSAC)*, pages 72–77, Sep. 2004.

[25] J. Stardom. *Metaheuristics and the search for covering and packing arrays*. Masters thesis, Simon Fraser University, 2001.

[26] K.C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Trans. on Software Engineering*, 28(1):109–111, Jan. 2002.

[27] Y.W. Tung and W.S. Aldiwan. Automating test case generation for the new generation mission software system. In *IEEE Aerospace Conference*, pages 431–37, Mar. 2000.

[28] R. C. Turban. Algorithms for covering arrays. *Ph.D. Thesis, Arizona State University, Department of Computer Science and Engineering*, May 2006.

[29] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. Testing of communicating systems: Tools and techniques. In *Intl. Conference on Testing Communicating Systems*, pages 59–74, Oct. 2000.

[30] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 31(1):20–34, Jan. 2006.