An Improved Meta-Heuristic Search for Constrained Interaction Testing

Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE 68588-0115
{*bgarvin, myra, dwyer*}*@cse.unl.edu*

Abstract

Combinatorial interaction testing (CIT) is a cost-effective sampling technique for discovering interaction faults in highly configurable systems. Recent work with greedy CIT algorithms efficiently supports constraints on the features that can coexist in a configuration. But when testing a single system configuration is expensive, greedy techniques perform worse than meta-heuristic algorithms because they produce larger samples. Unfortunately, current meta-heuristic algorithms are inefficient when constraints are present.

We investigate the sources of inefficiency, focusing on simulated annealing, a well-studied meta-heuristic algorithm. From our findings we propose changes to improve performance, including a reorganized search space based on the CIT problem structure. Our empirical evaluation demonstrates that the optimizations reduce run-time by three orders of magnitude and yield smaller samples. Moreover, on real problems the new version compares favorably with greedy algorithms.

1. Introduction

Software development is shifting to producing families of related products [2, 19]. Developing these families as integrated, highly-configurable systems offers significant opportunities for cost reduction through systematic component reuse. Unfortunately, highly configurable systems are even more difficult to validate than traditional software of comparable scale and complexity; faults may lie in the interactions between features. These *interaction faults* only appear when the corresponding features are combined, and it is generally impractical to validate every feature combination as that means testing all possible configurations [5, 6, 24].

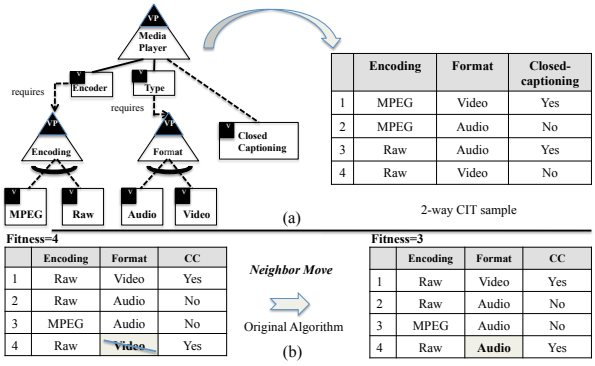
Combinatorial interaction testing (CIT) tempers the cost of validation by limiting the degree of feature interaction considered. Only a sample of the set of possible configurations is tested, but that sample contains at least one occurrence of every t-way interaction, where t is called the strength of testing [4].

CIT sampling is dominated by approximating algorithms of two varieties: greedy techniques such as the Automatic Efficient Test Case Generator (AETG) [4] or the In Parameter Order (IPO[G]) algorithm [14, 23] and meta-heuristic algorithms like simulated annealing [8, 21, 22]. Both greedy and meta-heuristic approaches construct solutions by making small, elementary decisions, but a greedy algorithm's selections are permanent, whereas meta-heuristic search may revisit its choices. Intuitively, greedy techniques should be faster because each decision occurs just once; for the same reason, meta-heuristic strategies should discover better answers when the consequences of selections are difficult to anticipate. In CIT this impression is accurate: greedy algorithms tend to run faster, but meta-heuristic searches usually yield smaller sample sizes [8]. Thus, the former are better when building and testing a configuration is inexpensive, but the latter become superior as these costs increase.

However, these observations ignore an inherent problem: highly-configurable systems typically have feature constraints—restrictions on the features that can coexist in a valid configuration. While earlier work efficiently handled constraints in a greedy algorithm [7], they remain a roadblock to meta-heuristic search [6]. Hence, if testing is expensive and feature constraints are present, neither approach is cost-effective.

In this paper we study the sources of inefficiency for the only published meta-heuristic algorithm for constrained CIT—a variation on simulated annealing [6] described in Section 3. In Section 4 we identify eight algorithmic improvements, two of which show significant promise: (1) modifying the global strategy for selecting a sample size and (2) changing the neighbor-

hood of the search. Then, in Section 5, we evaluate the benefits of these improvements on five real highly-configurable systems and 30 synthesized systems that share their characteristics. Our results show that simulated annealing can be cost-effective for constrained CIT problems: the changes reduce sample generation time by three orders of magnitude. Perhaps more importantly, when each configuration takes non-trivial time to test (for example, more than 30 seconds), the results suggest that simulated annealing's better sample sizes more than compensate for its longer run time, compared to a greedy algorithm. This makes it the superior choice for CIT on real systems where executing a test suite typically takes tens or hundreds of minutes [16].



	Encoding	Format	Closed-captioning
1	MPEG	Video	Yes
2	MPEG	Audio	No
3	Raw	Audio	Yes
4	Raw	Video	No

	Encoding	Format	CC
1	Raw	Video	Yes
2	Raw	Audio	No
3	MPEG	Audio	No
4	Raw	Video	Yes

	Encoding	Format	CC
1	Raw	Video	Yes
2	Raw	Audio	No
3	MPEG	Audio	No
4	Raw	**Audio**	Yes

Figure 1. Example of SPL and Neighbors

2. Background

We begin by introducing a small example of a software product line (SPL). An SPL is one type of configurable system, a family of related products defined through a well managed set of commonalities and points of variability [2, 20]. SPLs are used for development in many large companies and present unique challenges for validation [5, 17]. Our example is an SPL for a media player and appears on the left side of Figure 1(a) in the Orthogonal Variability Modeling language (OVM) [20]. In OVM variation points are shown as triangles, and rectangles represent each variant (or feature). Required variants are connected by solids lines, optional features use dashed lines, and arcs indicate alternative choices. Additionally, dashed arrows labeled "requires" indicate hierarchical relationships between variants and variation points. Because the modeling language is hierarchical, the selections of leaf features uniquely determine the final products.

There are two variation points of interest in this family, *encoding* and *format*, and a single optional variant, *closed-captioning*. The *encoding* is either MPEG or RAW, and *format* is AUDIO or VIDEO. Because *closed-captioning* is optional we represent its inclusion or exclusions with YES or NO. Hence, there are a total of eight possible products in this SPL.

If we plan to test the *family* for interaction faults then each of the eight products must be constructed and tested individually. Although this is reasonable in a small example, most realistic product lines have thousands or millions of possible products because the size of the family grows exponentially with the number of variation points. For instance, an unconstrained SPL with 10 variation points, each having 3 variants, describes 3^{10} or 59,049 distinct products.

Interaction faults in software usually depend on a small number of interacting features [13]; in CIT we sample to cover these interactions. Thereby we save testing effort: in the SPL with 10 variation points we only need 14 products to exercise all 2-way interactions, and in the example media player SPL we need just the 4 on the right side of Figure 1(a). A CIT sample is usually encoded as a covering array, which we define next.

2.1. Constrained Covering Arrays

A *covering array* (CA) is an array having N rows and k columns (factors). The elements in the i^{th} column are chosen from a set of v_i symbols (values). We adopt the convention that columns do not share symbols. The key property of a CA is that for any choice of t columns, all possible sets of t symbols (t-sets) appear in at least one row; the t-sets are said to be covered.

In CIT, a covering array's rows describe a sample of the configuration space; columns represent variation points, so each symbol in a row indicates the alternative chosen at the corresponding point. For example, $k = 3$ in the media player SPL, and each v is size 2. To test all pairwise interactions we would set $t = 2$ and obtain the sample of four products in Figure 1(a).

But there are also constraints such as hardware or software incompatibilities or developer/market preferences. Returning to the media player example, picking YES for *closed-captioning* implies the choice of VIDEO. The third configuration in our CIT sample (Figure 1(a)) violates this constraint—it is an invalid product.

The presence of such constraints in CIT was for a long time a major impediment to the sampling process [4, 6]. However, recent work [6] has shown that we can incorporate constraint information into the greedy sampling algorithms without increasing computational cost. This work led to the formalization of *constrained covering arrays* (CCAs) in [7].

In CCAs we define a Boolean variable for each symbol that could appear in the covering array. Given

a row, a variable is true if the corresponding symbol appears in that row, false otherwise. We then encode constraints on the rows as a propositional formula and restrict the covering array to rows that satisfy this formula. For instance, the formula VIDEO $\vee \neg$YES would capture the constraint where closed captioning implies video support. Once we've added constraints, fewer t-sets need to be covered, but the space of valid covering arrays has a more complicated structure.

2.2. Simulated Annealing

The objective in CIT is to find a CCA that minimizes N. Unfortunately, a tight bound is not known for general CAs, let alone CCAs [11], so approaches such as greedy algorithms and meta-heuristic search that approximate minimality are most prevalent. We concentrate on simulated annealing because it has produced small arrays for unconstrained problems [8, 22].

Under simulated annealing the search space comprises all $N \times k$ arrays populated with the values for each factor. The fitness function for the CIT problem is the number of uncovered t-sets in the sample. For instance, in Figure 1(b) on the left, we see an intermediate state of the algorithm when N is 4 (ignoring constraints for now). Of the twelve 2-sets that must be covered we are missing [MPEG, YES], [MPEG, VIDEO] [AUDIO, YES] and [VIDEO,NO]; the fitness is 4. A smaller fitness means that we are closer to the optimum, zero.

The algorithm starts with an initial, random array (the start state) and then makes a series of state transitions. Each transition is a random mutation of a single location in the solution. This mutation defines the search neighborhood. In Figure 1(b) a transition to a new solution is illustrated. The cost of the new solution, 3, is better.

The simulated annealing algorithm accepts a new solution if its fitness is the same or closer to optimal than the current one. When this is not the case, it uses a pre-defined temperature and cooling schedule to set the probability for making the bad move at $e^{f(S)-f(S')/T}$, where T is the current temperature, S and S' are the old and new solution, and $f(S)$ represents the fitness of solution S. As the algorithm proceeds, the temperature is cooled by a decrement value (the number of iterations between decrements is a parameter of the algorithm). As the temperature cools the probability drops and we are less likely to allow a bad move.

3. Base Algorithm

The original application of simulated annealing to covering array problems is described by Stevens in [22]. Though that algorithm requires each column to have the same number of symbols, Stardom [21] extended it to remove this restriction. In [8] we added optimizations and other features, and [6] introduced support for constraints. We used a satisfiability (SAT) solver to determine whether or not rows of the covering array satisfy the constraints at each move. Our work continues from this last version; we refer to it as the *base algorithm*.

Compared with our implementation of the AETG greedy algorithm, mAETG, the base algorithm met with limited success in [6]. For small 2-way problems it found constrained CIT samples that matched the quality of mAETG, but failed to compete at higher strengths both in size and time; the search had little information about constraints so it spent too many iterations undoing transitions to infeasible arrays. Moreover, though both programs took longer to solve constrained problems, the base algorithm's run time grew at a much faster rate.

The following subsections first explain the base algorithm's overall approach and then discuss its two primary components: an outer search and an inner search. Along the way we highlight some drawbacks that warrant remedy.

3.1. Design

Because the minimum covering array size cannot be known ahead of time, the base algorithm repeatedly calls simulated annealing, each time with a different N. Hence, there are two layers to the design. The *outer search* chooses values for N and either accepts or rejects them according to the results of an *inner search*, simulated annealing proper.

3.2. Outer Search

The outer search is shown in Figure 2. It takes an upper and lower bound on the size of the covering array and performs a binary search within this range.

There are two points of interest. First, at each size simulated annealing attempts to build an array (line 4), and the return value is the last array found, whether it is a solution or not, so its coverage must be checked (line 5). Second, the outer search is responsible for returning the smallest covering array constructed, so it must keep a copy of the best solution (line 6).

Though this approach works, binary search is a poor fit for this problem because its fundamental assumptions are violated. Our first two criticisms of the base algorithm are:

1. The binary search supposes that the inner search accurately evaluates whether an array of some size can be built. However, the inner search is stochas-

binarySearch$(t,k,v,C,lower,upper)$

```
1  let A ← ∅;
2  let N ← ⌊(lower + upper)/2⌋;
3  while upper ≥ lower do
4  |   let A' ← anneal(t,k,v,C,N);
5  |   if countNoncoverage(A') = 0 then
6  |   |   let A ← A';
7  |   |   let upper ← N − 1;
8  |   else
9  |   |   let lower ← N + 1;
10 |   end
11 |   let N ← ⌊(lower + upper)/2⌋;
12 end
13 return A;
```

Figure 2. The Base Outer Search, a Binary Search

tic and might terminate before finding a solution that does in fact exist.

2. The binary search expects the minimum size to lie within the given bounds. But the quantities *lower* and *upper* are merely estimates and may not bound the final answer, especially in constrained problems.

3.3. Inner Search

Figure 3 shows the specifics of the inner search. At each step, the code randomly chooses a location in the array and a symbol to put there (lines 4–6). If the replacement causes the row to violate constraints, it discards that move and tries again (line 9), following an unsophisticated policy called the death penalty [3]. Consequently, the search never enters an infeasible region. But if constraints are satisfied, it computes the change in coverage (line 10). When the alteration increases or maintains coverage, the algorithm applies it and continues (lines 11–13), but it only accepts bad moves with a probability that depends on the quality of the move and the current temperature (lines 14–16). The iterations continue until a stabilization criterion is met (line 3); that is, the algorithm has found a solution or seems not to be making further progress. Then the array is returned (line 21).

Unlike a typical implementation of simulated annealing, only the first start state is entirely random. Instead, because we are repeatedly invoking the inner search for nearly the same problem we use a single large array for every search and ignore the rows at index N and higher.

We point out four criticisms of the inner search and add them to our list:

3. Constraints decrease the connectivity of the state space, so the average path to a solution is longer in constrained problems. Thus the search needs many more iterations for the same probability of success.
4. In the extreme case, constraints disconnect the search space, rendering the problem unsolvable from some start states.
5. The initial state generator keeps no record of the symbols it has tried while creating random rows. Therefore, it may waste time making choices already known to violate constraints.
6. Although progress is saved from one run of simulated annealing to the next, the choice of rows to use is arbitrary. Difficult-to-obtain rows may be lost while redundant rows are preserved.

anneal(t,k,v,C,N)

```
1  let A ← initialState(t,k,v,C,N);
2  let temperature ← initialTemperature;
3  until stabilized(countNoncoverage(A)) do
4  |   choose row from 1...N;
5  |   choose column from 1...k;
6  |   choose symbol from v_column;
7  |   let A' ← A;
8  |   let A'_{row,column} ← symbol;
9  |   if SAT(C, A'_{row,1...k}) then
10 |   |   let δ ← countCoverage(A') − countCoverage(A);
11 |   |   if δ ≥ 0 then
12 |   |   |   let A ← A';
13 |   |   else
14 |   |   |   with probability e^{δ/temperature} do
15 |   |   |   |   let A ← A';
16 |   |   |   end
17 |   |   end
18 |   |   let temperature ← cool(temperature);
19 |   end
20 end
21 return A;
```

Figure 3. The Base Inner Search, Element-wise Simulated Annealing

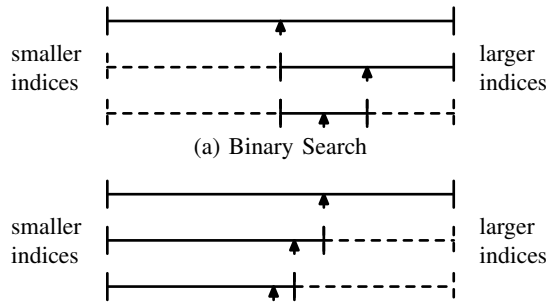
4. Modifications

For each shortcoming we detail a solution. Our ideas are organized into two categories: the two most influential changes appear as major modifications; the rest are listed under minor modifications.

4.1. Major Modifications

We discovered that our most significant alterations stemmed from Criticisms 1 and 3. We introduce one-sided narrowing, a meta-heuristic, to accommodate inaccuracy from the inner search and t-set replacement, a state space restructuring, to mitigate constraints' impact on state space traversal.

One-Sided Narrowing. In the previous section we pointed out that the binary search assumes simulated



(a) Binary Search

(b) One-Sided Narrowing

Figure 4. One-Sided Narrowing Chooses Partitions so that Larger Sizes are Eliminated

```
outermostSearch(t, k, v, C, lower, upper)
 1  let A ← ∅;
 2  while upper ≥ lower do
 3  |  let A′ ← binarySearch(t, k, v, C, lower, upper);
 4  |  if A′ = ∅ then
 5  |  |  break;
 6  |  else
 7  |  |  let A ← A′;
 8  |  |  let upper ← rows(A′) − 1;
 9  |  end
10  end
11  return A;
```

Figure 5. The New Outer Search, One-Sided Narrowing

annealing can decide whether a covering array is constructable (Criticism 1). This assumption does not hold because simulated annealing cannot conclusively show the nonexistence of a solution at a given size. Constraints make the problem harder, so the inner search is even more likely to return a false negative. Therefore, one-sided narrowing keeps the essence of a binary search, but abandons this faulty assumption.

The idea behind binary search is still valid: we want to restrict the range of candidate sizes as much as possible. Rather than narrowing from both sides though, the algorithm should only improve the upper bound because soundness is guaranteed. So at each step the code must either find a covering array of size $N \in [lower \ldots upper]$ and refine its search or give up.

The difference is illustrated by Figure 4. A solid range represents a subspace being considered; a dashed range has been eliminated. Arrowheads indicate the points of partition. In an ordinary binary search, part (a), the algorithm begins with a partition and then decides which half to discard. With one-sided narrowing, part (b), the program decides that the upper half will be eliminated, then it looks for a satisfactory partition.

Naïvely, the algorithm could always choose $N = upper$ as the partition, but our experience discourages a linear search. With rows being reused from one attempt to the next, larger fluctuations in size help knock the inner search out of local optima. Moreover, it is wasteful to decrease $upper$ by just one row when larger cuts might be possible.

Instead, we recognize that though the binary search cannot serve as the entire outer search, it does accomplish a single step: it returns an $N \in [lower \ldots upper]$ at which a covering array can be constructed or determines that it cannot find such an N. Therefore, one-sided narrowing uses binary search to locate each partition.

We now have a three-layer search. The new, outermost search shown in Figure 5 invokes the old outer search from Figure 2, which calls simulated annealing.

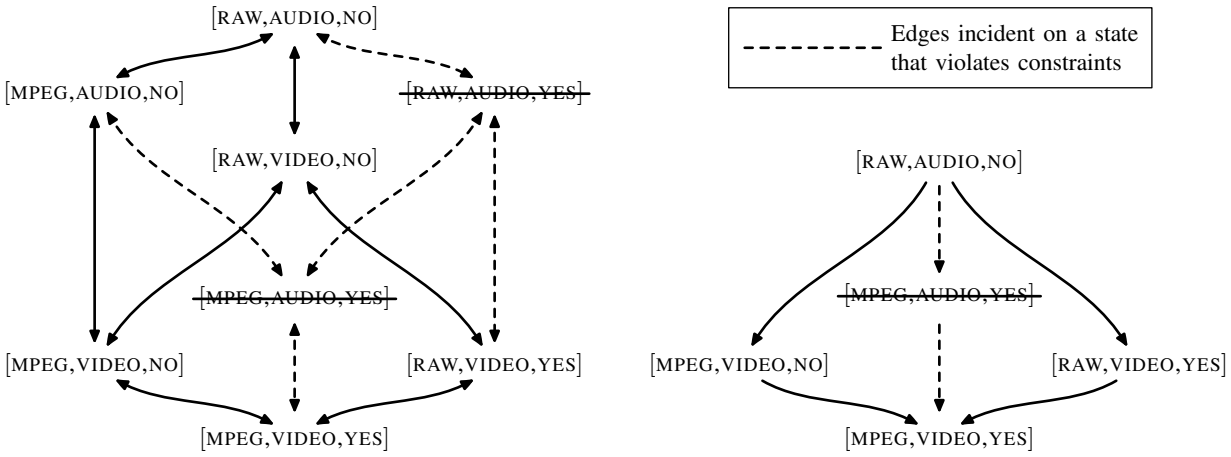
At first glance this seems to be an expensive proposition—finding the first partition means executing the entire base algorithm! Nevertheless, we expect to recover the cost by needing fewer iterations each time the inner search is called.

***t*-set Replacement.** Each iteration of simulated annealing attempts a single state transition; if we represent the state space as a graph, simulated annealing is tracing a path through the graph until it runs out of iterations or reaches a solution vertex. States that violate constraints form obstacles to search progress, leading to the problem in Criticism 3.

For example, suppose that we are testing 2-way interactions in the media player presented earlier. To make the search space small enough to illustrate, we will assume that every pair except [MPEG, VIDEO], [MPEG, YES], and [VIDEO, YES] is covered in the first rows of the covering array and that simulated annealing will only change the last row. Trivially, this row should be [MPEG, VIDEO, YES], so we will put its initial state as far away as possible, at [RAW, AUDIO, NO]. This gives the state space in Figure 6(a). Solid lines depict legal transitions; dashed lines indicate transitions that violate constraints. Note that every state also has three self-loops, not shown for the sake of readability.

Looking at just the upper right part of the graph, the path from [RAW, AUDIO, NO] through [RAW, AUDIO, YES] to [RAW, VIDEO, YES] was valid without constraints. But if simulated annealing attempts this route with constraints, it fails on the first transition and must instead detour via [RAW, VIDEO, NO].

To lessen the effect we allow simulated annealing to take short excursions through the infeasible states such as [RAW, AUDIO, YES]. Instead of checking feasibility after every transition, we have the search apply several transitions, then check. Or, equivalently, we treat this group of transitions as a single step. In the example, we combine pairs of transitions so that [RAW,



(a) Original Space

(b) Revised Space

Figure 6. The Original and Revised Search Space

VIDEO, YES] becomes directly reachable from [RAW, AUDIO, NO], much like what is shown in Figure 6(b).

We must be careful though: if we group too many transitions together adjacent states will be unrelated, and simulated annealing will degenerate to random search. To balance concerns about constraints with consideration for the search space we set the number of old moves per new move at t. There is particular advantage to choosing t. Rather than allowing all sets of t transitions, we can further inform the search by only choosing groups that substitute in a yet-to-be-covered t-set. We show this in Figure 6(b). For instance, [RAW, AUDIO, NO] becomes [RAW, VIDEO, YES] when the 2-set [VIDEO, YES] is added; the subsequent addition of [MPEG, VIDEO] or [MPEG, YES] leads to a solution.

We observe two clear advantages. First, in this new space, simulated annealing has many options when it is far from a solution, but the alternatives dwindle as it nears an answer. For example, in Figure 6(b), [RAW, AUDIO, NO] has three outgoing edges, [MPEG, VIDEO, YES] has one, and [MPEG, VIDEO, YES] has none. Being confined in a solution's vicinity, the algorithm should make a more thorough search there than in the rest of the space. Second, a t-set that by itself violates constraints will never be one that needs to be covered, so simulated annealing avoids these inherently infeasible transitions.

4.2. Minor Modifications

We developed several other changes, each motivated by a criticism from Section 3. These modifications were less effective, but we briefly list them for completeness. Criticisms 1, 2, 4, 5, and 6 are addressed by (1) dynamically bounding the inner search's iteration, (2) choosing binary search partitions more intelligently, (3) checking if the lower or upper bounds are attained, (4) replacing whole rows to escape highly constrained states, (5) adding a simple SAT history to the initial state generator, and (6) sorting the rows after each run of simulated annealing.

Along with these alterations we changed the underlying SAT solver from zChaff [15] in the base algorithm to MiniSAT [10] for comparability with the greedy implementation mAETG.

5. Evaluation

Our evaluation is organized as follows: We present two research questions and establish the scope of investigation. We then describe experiments aimed to answer each research question. Results and analysis follow. The evaluation artifacts are available for download at `http://cse.unl.edu/~bgarvin/research/ssbse2009/`.

5.1. Research Questions

Our first objective is to determine which changes are beneficial in the targeted context—constrained problems. This leads to the research question:

RQ1 (Effect of Modifications). How does each change affect performance on constrained problems?

Using the set of modifications that yield the best performance, we pose a second research question:

RQ2 (Comparison to mAETG). How does the best algorithm from RQ1 compare to a greedy approach, the modified AETG algorithm from [7]?

5.2. Scope

Conducting a full factorial experimentation to quantify the effects of each proposed change would be too costly. Instead, we group the modifications to study their effects. All of the minor modifications are grouped as a single change and we investigate the major changes—t-set replacement and one-sided narrowing—independently. In total there are four variations to the base algorithm: with minor changes (Minor), with minor changes and t-set replacement (TSR), with minor changes and one-sided narrowing (OSN), and with all of the changes (All).

Along with the minor modifications we refactored the implementation to more cleanly support our changes. When adding the minor changes, we kept the simulated annealing parameters from the base algorithm, but these settings were updated when major modifications were introduced; the specific parameter settings are available on the website given earlier.

5.3. Experiments

All of our data are obtained by executing simulated annealing once per problem unless otherwise indicated. The computations were performed on a 2.4GHz Opteron 250 running Linux. We record the final array size and total run time.

Effect of Modifications. In the first experiment we let t be two and compare all five versions on the 35 problems in [7]. Five of these problems are taken from real highly-configurable systems, and the other thirty are based on the characteristics of these five. More detail on these systems is given in [7].

Our dependent variables are the cost in run time and size of the generated sample. With just this raw data, it is not clear whether a small sample found at great expense is better than a large sample discovered quickly. But CIT is ultimately meant to reduce the total time spent selecting and testing configurations; this is the metric we should use.

This total cost is modeled as $c_{gen} + c_{config} \times N$, where c_{gen} is the cost to generate a sample and c_{config} the cost to test a configuration. Clearly, lower values of c_{gen} compensate for larger N. On the other hand, if c_{config} is expensive then expending more time to generate a smaller sample would make sense. Following convention, when comparing two CIT methods we call the point where $c_{gen} + c_{config} \times N = c'_{gen} + c_{config} \times N'$ the *break-even point*.

For this first experiment we use the sums over all the benchmarks to find break-even points. Our discussion of performance is based on these points.

Comparison to mAETG. The second experiment ran the best performing implementation 50 times over the 35 benchmarks. In this way, the averages are directly comparable with the mAETG figures reported in [7]. Our dependent variables are the array size, run time, and break-even point calculated from the mAETG and simulated annealing run times and array sizes.

5.4. Results and Analysis

Throughout the presentation of results we adopt an abbreviated notation for constrained covering arrays. A model term written as $x_1^{y_1} x_2^{y_2} \cdots x_n^{y_n}$ indicates that there are y_i columns with x_i symbols to choose from for each i. A constraint term, written in the same format, means that y_i constraints involve x_i symbols.

Effect of Modifications. The results for the first experiment are listed in Table 1. After the benchmark names and brief descriptions, the table gives the sample sizes and run times for each algorithm. The smallest values are shown in bold. Two samples for the base algorithm did not complete after 27 days (2.3 million seconds); in those cases we list N/A and append + to the sum.

It's clear that t-set replacement dramatically reduces run time. On the other hand, the effect of one-sided narrowing varies with the search space. In the old search space (columns Minor and OSN) it worsens array sizes, but combined with t-set replacement (between columns TSR and All), it always does at least as well. Put together, the All version runs more than a thousand times faster than the base algorithm and produces smaller covering arrays.

When we compute break-even points, only the version TSR ever outperforms All. So there is only one break-even point of interest: variation TSR is best when the per-configuration time is less than 39.79 seconds, and after that the All version dominates. Because few test suites for configurable systems run in under 40 seconds [16], we conclude that including both major modifications is suitable for the common case.

Comparison to mAETG. Table 2 gives the averages for simulated annealing, including both changes, and mAETG over 50 runs. For every row we choose the mAETG variant with the best break-even point compared to simulated annealing, breaking the one tie (on benchmark 22) by array size.

The contrast is sharp. Simulated annealing gives better array sizes in all but one case, averaging 11.16 fewer rows; over the set of 35 samples we only need 75% as many configurations to achieve the same CIT goals. If we had chosen different mAETG variants for each problem that produce the smallest sizes rather than the best break-even point, the average difference would

Table 1. Sizes and Times for Constrained Two-Way Problems

			Size					Run Time (s)				
Name	Model	Constraints	Base	Minor	TSR	OSN	All	Base	Minor	TSR	OSN	All
SPIN-S	$2^{13}4^5$	2^{13}	24	21	20	26	**19**	1513.8	1827.2	**1.7**	140.1	32.2
SPIN-V	$2^{42}3^24^{11}$	$2^{47}3^2$	36	33	39	**32**	36	22278.2	22206.0	**18.5**	122355.0	122.2
GCC	$2^{189}3^{10}$	$2^{37}3^3$	21	**19**	25	**19**	**19**	49856.7	50981.0	1250.4	61013.0	**850.6**
Apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$	33	**31**	36	47	34	47465.2	31638.4	**60.6**	12686.9	137.8
Bugzilla	$2^{49}3^14^2$	2^43^1	17	**16**	18	**16**	17	972.6	123.1	**4.1**	2081.4	4.4
1.	$2^{86}3^34^15^56^2$	$2^{20}3^34^1$	44	40	40	63	**38**	41055.5	9154.3	**31.6**	7047.5	74.3
2.	$2^{86}3^34^35^16^1$	$2^{19}3^3$	32	**30**	33	44	32	80020.5	6185.9	24.1	6650.4	**9.2**
3.	$2^{27}4^2$	2^93^1	**18**	**18**	**18**	**18**	**18**	284.4	3718.5	4.0	327.2	**2.8**
4.	$2^{51}3^44^25^1$	$2^{15}3^2$	21	**20**	22	23	21	16968.6	1096.8	6.2	2725.7	**5.8**
5.	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	60	48	65	81	**47**	69107.0	69003.4	**77.0**	62702.1	337.0
6.	$2^{73}4^36^1$	$2^{26}3^4$	**24**	**24**	**24**	**24**	**24**	62706.3	1777.6	**7.7**	16454.9	29.4
7.	$2^{29}3^1$	$2^{13}3^2$	**9**	**9**	**9**	**9**	**9**	121.9	2902.2	**1.5**	192.2	2.1
8.	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	49	43	45	66	**39**	243499.0	11621.1	**30.5**	19858.2	105.9
9.	$2^{57}3^14^15^16^1$	$2^{30}3^7$	N/A	30	30	**20**	**20**	N/A	10.8	**3.3**	2103.3	25.6
10.	$2^{130}3^64^55^26^4$	$2^{40}3^7$	58	44	48	78	**43**	257061.2	40577.3	**60.4**	9575.0	258.8
11.	$2^{84}3^44^25^26^4$	$2^{28}3^4$	46	43	47	43	**41**	170596.4	63153.2	**25.0**	143728.6	139.7
12.	$2^{136}3^44^35^16^3$	$2^{23}3^4$	49	41	43	62	**40**	359336.2	14544.8	**66.6**	16873.7	116.3
13.	$2^{124}3^44^15^26^2$	$2^{22}3^4$	40	**36**	**36**	54	**36**	304404.0	12641.5	**31.5**	6237.2	56.1
14.	$2^{81}3^54^36^3$	$2^{13}3^2$	37	37	41	**36**	37	81464.6	63353.5	**19.6**	159571.8	104.0
15.	$2^{50}3^44^15^26^1$	$2^{20}3^2$	N/A	**30**	**30**	**30**	**30**	N/A	6467.3	**9.6**	61320.4	20.0
16.	$2^{81}3^34^26^1$	$2^{30}3^4$	**19**	24	25	24	24	1001.9	2208.4	**15.2**	27995.8	60.9
17.	$2^{128}3^34^25^16^3$	$2^{25}3^4$	42	**39**	41	64	**39**	47396.6	36955.7	**51.7**	10406.1	69.7
18.	$2^{127}3^24^45^66^2$	$2^{23}3^44^1$	177	44	45	72	**43**	15515.0	18433.3	**61.2**	2511.9	72.7
19.	$2^{172}3^94^95^36^4$	$2^{38}3^5$	180	49	51	103	**47**	132833.5	68558.4	**118.3**	8436.6	1550.7
20.	$2^{138}3^44^55^46^7$	$2^{42}3^6$	67	56	57	105	**53**	340518.9	32677.3	**86.4**	25030.8	915.3
21.	$2^{76}3^34^25^16^3$	$2^{40}3^6$	37	36	37	53	**36**	81000.3	16736.3	**20.3**	6816.0	37.6
22.	$2^{72}3^44^16^2$	**36**	36	36	37	40	**36**	64098.0	2699.5	12.5	2882.2	**12.4**
23.	$2^{25}3^16^1$	$2^{13}3^2$	13	18	18	**12**	13	48.0	0.7	**0.5**	906.0	5.0
24.	$2^{110}3^25^36^4$	$2^{25}3^4$	51	**44**	47	71	**44**	241610.7	113832.0	**32.8**	11674.2	80.2
25.	$2^{118}3^64^25^26^6$	$2^{23}3^34^1$	59	**49**	54	80	**49**	26043.5	143208.0	**45.2**	23952.7	285.1
26.	$2^{87}3^14^35^4$	$2^{28}3^4$	**30**	**30**	33	48	31	151137.6	43067.7	**23.1**	5821.4	25.4
27.	$2^{55}3^24^25^16^2$	$2^{17}3^3$	**36**	**36**	**36**	47	**36**	5105.4	1634.9	**5.7**	2402.5	7.8
28.	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	164	51	55	84	**50**	255115.0	115759.9	**126.3**	71001.6	787.3
29.	$2^{134}3^75^3$	$2^{19}3^3$	31	29	32	40	**27**	36434.7	17289.0	**38.7**	9752.7	283.8
30.	$2^{73}3^34^3$	$2^{20}3^2$	20	**17**	20	20	19	33245.9	32270.0	**16.1**	9378.8	36.7
Sum			1580+	1171	1257	1654	**1147**	7905417.1+	1058315.0	**2387.5**	932614.0	6696.9

only decline to 11.06, still a 25% cut. However, the overall cost for running our algorithm on these samples is more than six times that of mAETG.

In the last column we show the break-even points between mAETG and simulated annealing as the minimum time that each configuration must take before simulated annealing outperforms mAETG. Most of these times are quite small, less than 30 seconds. There are two notable exceptions: the break-even point for the GCC optimizer is just short of eight minutes, and our algorithm never does better on benchmark 22. As demonstrated by [16] though, eight minutes is not much time to test a configuration, so we conclude that the new simulated annealing is competitive in every case but one.

Further Analysis. To confirm that we have still retained the quality of the original simulated annealing algorithm on unconstrained problems we selected 29 benchmarks from a variety of publications. For each we ran version All one time and compared this with the best size reported for simulated annealing. The aggregate figures are promising; our implementation's total array size is only 0.2% higher than the best figures.

5.5. Threats to Validity

We classify our threats as affecting external, internal, or construct validity.

External Validity. The major external threat is our choice of benchmarks. We cannot guarantee that either the real CIT problems or the synthetic inputs accurately represent configurable software. Though diverse, they are all derived from open-source command-line or web-based software.

Internal Validity. There are four main threats to internal validity. First, in RQ1 we only use one run of stochastic algorithms. Our continuing experimentation suggests that this threat is minor; we see consistency as in RQ2 where the standard deviation in size is no more than three rows on any benchmark and less than a row

Table 2. Average Size and Times Over 50 Runs for Constrained Two-Way Problems

Name	Size mAETG Best [7]	Size All	Run Time (s) mAETG Best [7]	Run Time (s) All	Break-even (s/cfg.)
SPIN-S	27.0	**19.4**	**0.2**	8.6	1.11
SPIN-V	42.5	**36.8**	**11.3**	102.1	15.93
GCC	24.7	**21.1**	**204.0**	1902.0	471.67
Apache	42.6	**32.3**	**76.4**	109.1	3.17
Bugzilla	21.8	**16.2**	**1.9**	9.1	1.29
1.	53.3	**38.6**	**24.4**	179.5	10.55
2.	40.5	**31.0**	**14.7**	25.4	1.13
3.	20.8	**18.0**	**0.2**	3.4	1.14
4.	28.6	**21.0**	**3.1**	29.3	3.45
5.	63.8	**47.7**	**134.8**	655.9	32.37
6.	34.0	**24.2**	**7.2**	18.2	1.12
7.	12.5	**9.0**	**0.3**	2.1	0.51
8.	55.6	**40.5**	**45.1**	249.9	13.56
9.	26.0	**20.0**	**3.0**	29.8	4.47
10.	60.4	**44.0**	**74.3**	357.3	17.26
11.	58.3	**41.9**	**23.8**	240.7	13.23
12.	54.5	**40.4**	**68.0**	221.1	10.86
13.	48.6	**36.5**	**45.5**	60.5	1.24
14.	51.8	**36.9**	**20.2**	58.1	2.54
15.	40.4	**30.7**	**4.9**	19.3	1.48
16.	33.4	**24.1**	**9.7**	19.7	1.08
17.	53.4	**39.2**	**46.9**	335.5	20.32
18.	57.3	**42.3**	**60.1**	303.5	16.23
19.	64.7	**47.8**	**168.4**	823.6	38.77
20.	71.5	**53.2**	**121.1**	1133.3	55.31
21.	51.7	**36.3**	**14.3**	46.2	2.07
22.	**26.2**	36.0	**6.7**	12.3	N/A
23.	15.7	**12.6**	**0.6**	10.1	3.06
24.	58.3	**42.8**	**40.6**	304.5	17.03
25.	65.7	**48.3**	**61.7**	507.8	25.64
26.	42.0	**30.7**	**15.5**	71.2	4.93
27.	45.8	**36.0**	**4.6**	10.8	0.63
28.	68.5	**50.7**	**170.4**	1522.3	75.95
29.	38.4	**29.7**	**38.9**	89.3	5.79
30.	45.8	**19.3**	**10.3**	30.5	0.76
Sum	1546.1	**1155.4**	**1533.1**	9501.8	20.40

on average. Second, it may be that we bias the results by making poor parameter choices for some algorithms. For instance, perhaps we should have chosen parameters for the versions with major modifications on an individual basis, rather than as a group. Third, the refactoring that accompanies the minor changes may have affected the run time; however, this threat only affects comparisons to the base version, not the effects of the major changes. Fourth, although we have verified the results of every run, we cannot be completely confident that the implementations are correct translations from pseudocode, nor that they are bug-free.

Construct Validity. We have considered both run time and the size of the output, but we have ignored other metrics that are important in specific scenarios. For example, if we do not expect to finish the interaction testing, diversity of t-sets in the early rows is desirable.

6. Related Work

There has been a large body of work on constructing unconstrained interaction test samples [4, 8, 14, 21–23]. Two primary algorithmic techniques are greedy [4, 9, 14, 23] and meta-heuristic search. The latter includes implementations of simulated annealing, tabu search and genetic algorithms [8, 18, 21, 22].

The work of D. Cohen et al. [4] describes the problem of constrained CIT although the primary means to handle this is through a re-modeling of the CIT parameters. Czerwonka [9] provides a more general constraint handling technique and uses the t-set as the granularity of the search. However, few details of the exact constraint handling techniques are provided. Both of the above algorithms are greedy. Our own work [6, 7] provides a more general solution for greedy construction of constrained CIT samples.

Hnich et al. [12] use a SAT solver to construct CIT samples, but they do not provide a direct way to encode constraints into the problem. Recent work by [1] incorporates constraints directly into their ATGT tool which utilizes a model checker to search for a CIT sample. The only work we are aware of that incorporates constraints into a meta-heuristic CIT algorithm is our own [6] but this solution does not scale.

In this work we focus on one meta-heuristic search algorithm, simulated annealing [8], and reformulate the search to work well on both constrained and unconstrained CIT problems.

7. Conclusions

Testers need CIT tools that perform well in the presence of constraints. Earlier work focused on greedy algorithms, which tend to build larger samples than meta-heuristic searches. But the meta-heuristic search, simulated annealing, did not retain its quality and scaled poorly when finding constrained samples.

In this paper, we adapted a simulated annealing algorithm for constrained CIT. It finds the CIT sample size more efficiently and employs a coarser-grained search neighborhood. Together with some minor modifications these changes provide small sample sizes for a fraction of the original cost, less than one thousandth of the run time.

We have run experiments on four versions of the constrained simulated annealing algorithm and compared the best of these with a greedy implementation on a set of 35 realistic samples. Our experimental results show that we need on average 25% fewer configurations, but the run time for simulated annealing is longer. When we calculate the break-even point based on the

sample construction time and the time to run the test suite on each configuration, we find that if a test suite takes more than 30 seconds to run, simulated annealing typically outperforms the greedy algorithm.

In future work we will plan to add more runs to our experiments, experiment with higher strength CIT samples, and to optimize the simulated annealing parameters for both constrained and unconstrained problems.

8. Acknowledgments

We would like to thank Jiangfan Shi for the use of his mAETG tool and for supplying the CIT models for evaluation of RQ1 and RQ2. Brady Garvin is supported in part by CFDA#84.200A: Graduate Assistance in Areas of National Need (GAANN). This work is supported in part by the National Science Foundation through awards CNS-0454203, CCF-0541263, CNS-0720654, and CCF-0747009, by the National Aeronautics and Space Administration under grant number NNX08AV20A, by the Army Research Office through DURIP award W91NF-04-1-0104, and by the Air Force Office of Scientific Research through award FA9550-09-1-0129. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF, NASA, ARO or AFOSR.

References

[1] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Tests and Proofs, Lecture Notes in Computer Science, 4966*, pages 66–83, 2008.

[2] P. Clements and L. Northrup. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[3] C. Coello Coello. Theoretical and numerical constraint handling techniques used with evolutionary algorithms: A survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, Jan. 2002.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[5] M. B. Cohen, M. B. Dwyer, and J.Shi. Coverage and adequacy in software product line testing. In *Proceedings of the Workshop on the Role of Architecture for Testing and Analysis*, pages 53–63, July 2006.

[6] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis*, pages 129–139, July 2007.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[8] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 38–48, May 2003.

[9] J. Czerwonka. Pairwise testing in real world. In *Pacific Northwest Software Quality Conference*, pages 419–430, October 2006.

[10] N. Eén and N. Sörrenson. MiniSAT-C v1.14.1. http://minisat.se/, 2007.

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

[12] B. Hnich, S. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006.

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

[14] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A general strategy for t-way software testing. *Engineering of Computer-Based Systems, IEEE International Conference on the*, pages 549–556, 2007.

[15] S. Malik. zChaff. http://www.princeton.edu/~chaff/zchaff.html, 2004.

[16] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *Proceedings of the 26th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2004)*, 2004.

[17] H. Muccini and A. van der Hoek. Towards testing product line architectures. In *Proceedings of the International Workshop on Test and Analysis of Component-Based Systems*, pages 111–121, 2003.

[18] K. J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.

[19] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[20] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, Berlin, 2005.

[21] J. Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, 2001.

[22] B. Stevens. *Transversal Covers and Packings*. PhD thesis, University of Toronto, 1998.

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

[24] 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.