Empir Software Eng (2011) 16:61–102
DOI 10.1007/s10664-010-9135-7

Evaluating improvements to a meta-heuristic search for constrained interaction testing

Brady J. Garvin · Myra B. Cohen · Matthew B. Dwyer

Published online: 13 July 2010

Editors: Massimiliano Di Penta and Simon Poulding

Abstract Combinatorial interaction testing (CIT) is a cost-effective sampling technique for discovering interaction faults in highly-configurable systems. Constrained CIT extends the technique to situations where some features cannot coexist in a configuration, and is therefore more applicable to real-world software. Recent work on greedy algorithms to build CIT samples now efficiently supports these feature constraints. But when testing a single system configuration is expensive, greedy techniques perform worse than meta-heuristic algorithms, because greedy algorithms generally need larger samples to exercise the same set of interactions. On the other hand, current meta-heuristic algorithms have long run times when feature constraints are present. Neither class of algorithm is suitable when both constraints and the cost of testing configurations are important factors. Therefore, we reformulate one meta-heuristic search algorithm for constructing CIT samples, simulated annealing, to more efficiently incorporate constraints. We identify a set of algorithmic changes and experiment with our modifications on 35 realistic constrained problems and on a set of unconstrained problems from the literature to isolate the factors that improve performance. Our evaluation determines that the optimizations reduce run time by a factor of 90 and accomplish the same coverage objectives with even fewer system configurations. Furthermore, the new version compares favorably with greedy algorithms on real-world problems, and, though our modifications were

B. J. Garvin (✉) · M. B. Cohen · M. B. Dwyer
Department of Computer Science and Engineering,
University of Nebraska—Lincoln, Lincoln, NE 68588-0115, USA
e-mail: bgarvin@cse.unl.edu

M. B. Cohen
e-mail: myra@cse.unl.edu

M. B. Dwyer
e-mail: dwyer@cse.unl.edu

aimed at constrained problems, it shows similar advantages when feature constraints are absent.

Keywords Constrained combinatorial interaction testing · Configurable software · Search based software engineering

1 Introduction

Software development is shifting to families of products where commonalities can be shared and variations systematized (Clements and Northrup 2002). At the same time software is also becoming more customizable. Whether the choice of features belongs to the developer or the user, both situations point to the rise of highly-configurable systems, systems in which features can be added to or removed from the core set of program functionality. The changes may be cosmetic, like user interface preferences, platform-based, such as a port to another operating system, or more extensive modifications to functionality, for instance, the selection of software capabilities to match a specific customer base.

Importantly, 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 specific sets of features are combined, and it is generally impractical to validate every feature combination as that means testing all possible configurations (Kuhn et al. 2004; Yilmaz et al. 2006). Instead, testers need a technique for judiciously sampling the possible configurations.

An empirically justified approach is *combinatorial interaction testing (CIT)*, where the sample is guaranteed to contain at least one occurrence of every t-way interaction (Cohen et al. 1997; Yilmaz et al. 2006; Qu et al. 2008); the value t is called the strength of testing. Finding small samples that satisfy this criterion is a combinatorial optimization problem.

Two strategies to find samples that have been used frequently in the literature are greedy sample construction, epitomized by the Automatic Efficient Test Case Generator (AETG) developed by Cohen et al. (1997) and the In Parameter Order (IPO[G]) algorithm (Lei et al. 2007; Tai and Lei 2002), and meta-heuristic search, exemplified in this domain by simulated annealing (Cohen et al. 2003b; Stardom 2001; Stevens 1998). 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 run faster because each decision occurs just once; for the same reason, meta-heuristic strategies should discover better answers when the consequences of a selection are difficult to anticipate. In CIT this impression is accurate: greedy algorithms tend to generate samples more quickly, but meta-heuristic searches usually yield smaller sample sizes (Bryce et al. 2005; Cohen et al. 2003b). Thus, the former are better when building and testing a configuration is inexpensive, but the latter become superior as these costs increase.

There is, however, another difference between these strategies. Highly-configurable systems typically have feature constraints—restrictions on the features

that can coexist in a valid configuration—and while earlier work efficiently handled constraints in a greedy algorithm (Cohen et al. 2008), they remain a roadblock to meta-heuristic search, dramatically increasing the time needed to build a CIT sample (Cohen et al. 2007b). Hence, if testing is expensive and feature constraints are present, neither approach is cost-effective.

In an earlier publication (Garvin et al. 2009) we experimented with a set of modifications to simulated annealing in order to fill this gap. Our results demonstrate that the altered implementation of simulated annealing produces smaller samples than the competing greedy algorithm and that, in most cases, when considering both the time to find the samples and to run the tests, the new formulation improves on the constrained greedy methods. In this work we expand on those results and more formally study the sources of inefficiency in a previously published meta-heuristic algorithm for constrained CIT—a variation on simulated annealing (Cohen et al. 2007b). From our observations, we identify eight algorithmic modifications, two of which show significant promise: modifying the global strategy for selecting a sample size and changing the neighborhood of the search. We empirically validate the benefits of all eight modifications across a range of constrained CIT problems, then compare the best of the modifications on higher strength and unconstrained samples to understand the impact of these changes when the character of our problem differs.

The results show that on constrained problems the updated simulated annealing approach produces CIT samples that have on average 25% fewer configurations than the greedy algorithm, but the run times are longer. Considering both the time taken to construct and to test all configurations, the analysis determines that in most cases if a test suite takes longer than 21 s to run, the reformulated simulated annealing outperforms the greedy algorithm. Additionally, the new search scales to higher values of t and appears to retain competitiveness on a majority of the unconstrained problems we examined.

The contributions of this work include:

1. An improved simulated annealing algorithm for finding constrained CIT samples and an implementation of this algorithm available for download.
2. A discussion of the insights that led to each change and how they can potentially impact search progress.
3. A metric for assessing the total cost of applying CIT that accounts for both the sample generation and the execution of a test suite under each configuration.
4. An empirical evaluation that quantifies the impact of these changes and compares the modified sample generator to a state-of-the-art greedy algorithm.

The rest of this paper is laid out as follows. In the next section we present background on constrained CIT and describe how we have modified an existing simulated annealing CIT algorithm to improve comparability between variants of that algorithm and existing greedy search CIT algorithms; we refer to this modified simulated annealing algorithm as the *base* algorithm. Section 3 presents our observations of the base algorithm and the modifications that they led to. In Section 4, we conduct experiments that measure the effectiveness of our changes and yield insight as to when they are applicable. Under Section 5 we present related work. Our conclusions and future work are summarized in Section 6.

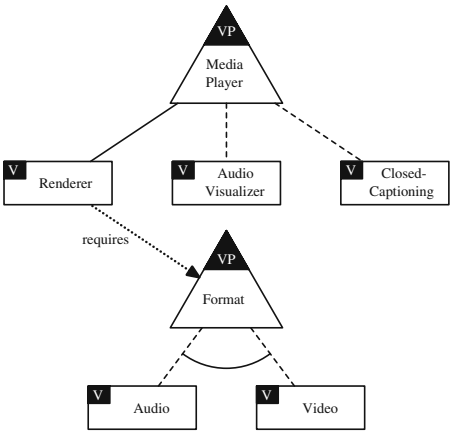
2 Background

To illustrate how constrained CIT problems arise, we present a small software product line (SPL) and consider how we might detect its interaction faults.

2.1 An Example

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 (Clements and Northrup 2002; Pohl et al. 2005). Our example SPL for a media player appears on the left side of Fig. 1 in the Orthogonal Variability Modeling (OVM) language (Pohl et al. 2005). Triangles in OVM indicate variation points, while rectangles represent variants, i.e. features. The uppermost triangle shows that a media player may have three key features: a renderer, an audio visualizer, and closed-captioning. Of these, the renderer is mandatory, as indicated by the solid line, but the audio visualizer and closed-captioning are optional. Furthermore, the mandatory renderer entails a choice of sub-features: we must decide on a format. There are two options, audio and video, which the arc denotes as mutually exclusive. In summary, there are three binary choices to make: which type of media, whether the audio can be visualized, and if closed-captioning is supported.

Were we to test the entire product line for interaction faults, we would construct the eight possible products and then execute each product on a test suite designed to



2-w ay CIT Sample

	Format	*Audio Visualizer*	*Closed-Captioning*
1.	AUDIO	VIS.	NO CC
2.	AUDIO	NO VIS.	CC
3.	VIDEO	NO VIS.	NO CC
4.	VIDEO	VIS.	CC

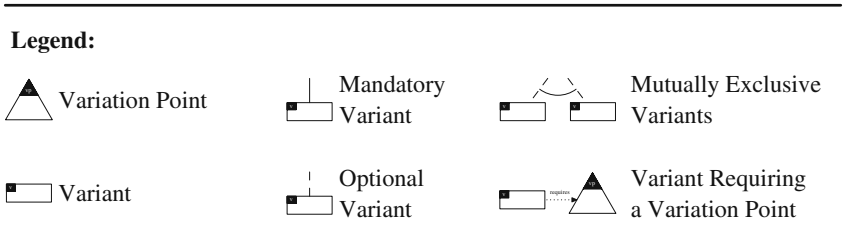


Fig. 1 An SPL and a corresponding covering array

expose interaction faults. On our small example this approach might be reasonable, but the work is exponential in the number of feature choices. For a realistic product line, exhaustive testing might mean building millions of products.

2.2 Covering Arrays

We can reduce the number of products to test by recognizing that software interaction faults usually depend on a small number of interacting features (Kuhn et al. 2004). Exhaustive sampling of the configuration space in the example exercises every three-way interaction. If we settle instead for every two-way interaction we need only test half as many configurations—for instance, the four rows on the right side of Fig. 1. The savings are more pronounced on larger systems, such as the real-world examples used in the evaluation (Section 4): on the order of 10^5–10^{56} configurations saved for each benchmark problem.

Formally, the table in Fig. 1 is an example of a well-studied mathematical structure, a *covering array* (CA) (Colbourn 2004). By convention, a covering array has N rows and k columns (also called factors) where the elements in the i^{th} column are chosen from a set of v_i symbols (also called values). The key property 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. From discrete mathematics, there is currently no general, efficient technique for finding the minimal value of N given the values of k and each v_i (Hartman and Raskin 2004). But there have been many mathematical and algorithmic techniques applied to this optimization problem (Cohen et al. 1997, 2003a, b; Hartman and Raskin 2004; Lei et al. 2007; Tai and Lei 2002, Stardom 2001; Stevens 1998). They approximate the minimum N and can find small samples at a reasonable computational cost.

2.3 Constrained Covering Arrays

Covering array generators alone may not suit testers' purposes though. There is an error in the model of the SPL of Fig. 1: closed-captioning ought to require a visual display, either audio visualization or the video format. Though there are still three binary choices, this constraint invalidates one of the eight original possibilities, the second row in the covering array. In practice, when considering constraints, the number of invalid rows can be quite large, often as much as 90% of the array (Cohen et al. 2008). This means that the sample should take constraints into account. The generalization of the covering array structure to support constraints is called a *constrained covering array* (CCA)[1] (Cohen et al. 2008).

In CCAs, each column is given a variable, and the column's values constitute the domain for its variable. Then each row is treated as an assignment to those variables, which must satisfy a propositional formula that encodes the feature constraints. For instance, the formula $(\mathit{Format} = \text{VIDEO}) \vee (\mathit{Audio\ Visualizer} = \text{VIS.}) \vee \neg(\mathit{Closed\text{-}Captioning} = \text{CC})$ would capture the requirement that closed captioning implies video

[1]Ordinary covering arrays, strictly speaking, are already constrained—they must cover all t-sets. However, in this paper we follow the conventions of the literature and use the terms *constrained covering array* and *unconstrained covering array* to refer to samples with and without feature constraints, respectively.

support. After adding constraints, it is possible that fewer *t*-sets can be covered. For instance, if we strengthened the feature constraints and forbade the combination of *Format* = AUDIO with *Closed-Captioning* = CC, the pair [AUDIO, CC] would be deemed *unattainable* and no longer be required to appear in the array. However, even if there are fewer combinations to cover, the space of valid covering arrays is complicated by feature constraints: they break the space's regular structure and global symmetry. In fact, creating just one row of a constrained covering array means finding a satisfying assignment to an arbitrary boolean formula, an NP-complete problem.

2.4 Search-Based Approaches to CIT

Algorithmic techniques to find covering arrays include both greedy and meta-heuristic algorithms. Prominent examples of the former are the In Parameter Order (IPO) algorithm (Lei et al. 2007; Tai and Lei 2002), the Automatic Efficient Test Case Generator (AETG) developed by Cohen et al. (1997), and the deterministic density algorithm (Colbourn et al. 2004). Some of these algorithms approach the problem by finding one configuration at a time that maximizes the number of newly covered interactions; others add one column at a time. The meta-heuristic category includes simulated annealing (Cohen et al. 2003a, b) and tabu search (Nurmela 2004). In these algorithms the optimization focuses on one value of N at a time, attempting to find a covering array for that size. Depending on the success or failure of the search, the value of N is then refined. There have also been some combined greedy and heuristic search approaches suggested (Bryce and Colbourn 2007) where a single configuration is added to the sample at a time, but a meta-heuristic search is used to find each of those individual configurations. The main drawback to many of these approaches, whether greedy, meta-heuristic, or hybrid, is that they do not handle constrained CIT problems.

In our earlier work (Cohen et al. 2007b, 2008) both a greedy algorithm—a variant of AETG called mAETG—and a meta-heuristic search technique—simulated annealing—were modified to handle constraints during sample generation. The algorithms were coupled with an off-the-shelf satisfiability (SAT) solver. Through a tight integration between the SAT solver and the greedy algorithm, the generator not only can produce small constrained CIT samples efficiently, but runs faster in some cases than the unmodified mAETG does building unconstrained samples of the same problems. In contrast, the constrained simulated annealing algorithm is only effective on small examples; neither its run time nor its sample sizes scale well with problem size. Because this work begins with this implementation of simulated annealing, we describe the basis for its design in the next two sections.

2.5 Simulated Annealing

Meta-heuristic search algorithms work stochastically to solve optimization problems under the guidance of a *fitness* function, a numerical representation of the quality of a candidate solution. Unlike plain heuristic algorithms, which have only procedures for making local improvements in fitness, meta-heuristic searches also employ global strategies, the *meta-heuristics* of their name. This addition is key; at any one moment a local search can only explore the *neighborhood* of its current state—the candidate solutions that are reachable by applying a single *transformation* (or *move*)—and is

subject to local optima. With a meta-heuristic the search is more likely to continue exploring other parts of its *state space*, which is the set of all candidate solutions.

Meta-heuristic searches can operate on either individual states or populations of states. The former are called *local meta-heuristic searches*, whereas the latter are *population-based meta-heuristic searches*. Note that the term "local" has a different meaning here than it does when describing a local search strategy.

Because meta-heuristic search is a paradigm to be adapted to different problems, every search has a set of common elements that must be defined. First, the problem being solved needs a concrete representation for a single state; a *candidate solution*. Second, there must be a *fitness function* that encodes the quality of an individual solution (note that the optimization can be framed as either a minimization or a maximization of the fitness function). Third, meta-heuristics require a method for building the initial or start state(s). Fourth, one must specify a *transformation function* that determines the neighborhood that the local improvement procedure examines. Finally, a *stopping criterion* needs to be defined for the algorithm to terminate.

Simulated annealing is a local meta-heuristic search. Its strategy is that of hill climbing, but altered with a meta-heuristic modeled on the physical annealing process. In each iteration it randomly selects a transformation, applies it, and compares the new state to the old one to see which should be kept. Like the atoms in heated metal or glass, the search acts under two forces: the tendency for crystals to seek a lower energy state is analogous to simulated annealing always applying a transformation if it improves the fitness value; the atoms' random motions at high temperatures are mirrored in simulated annealing probabilistically accepting fitness-worsening moves.

In the case of covering arrays, the fitness function, the number of yet-to-be-covered t-sets, is minimized. Therefore, this probability is given by $e^{-\Delta\text{fitness}/\text{temperature}}$ where the temperature variable plays the role of temperature in physical annealing, measuring the influence of the second force as compared to the first. As in physical annealing, the search gradually reduces the temperature according to a pre-planned *cooling schedule*. Consequently, the algorithm's inclination to explore decreases, and fitness-worsening moves are less likely. The idea is that the early part of the search will escape local optima because of its highly random nature and so explore more of the state space. Once it has settled in the vicinity of a global optimum, the colder temperatures will allow it to focus on making local improvements.

Specifics such as how the initial state is created and how the search responds to fitness-preserving transformations depend on the implementation. It is usual to create the initial state randomly and take neutral moves, i.e., moves to new states of equal fitness.

2.6 The Base Simulated Annealing Algorithm

Our observations and modifications in this work apply to an application of simulated annealing to sample generation for CIT. We refer to the it as the *base algorithm* and describe first the unconstrained and then the constrained version in the following sections.

The global optimization goal in the CIT sample generation problem is to find a minimally sized sample, subject to the constraint that all t-sets be covered. Although

it may be possible to minimize N directly, the simulated annealing algorithm for this problem, starting with the work of Stevens (1998), separated the problem into two parts to simplify the optimization. The part concerned with the minimization of N we call the *outer search*, and the part that ensures full coverage we refer to as the *inner search*.

2.6.1 Outer Search (Binary Search)

The outer search is shown in Fig. 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).

2.6.2 Inner Search (Simulated Annealing)

The inner search, simulated annealing proper, has the more interesting task of finding covering arrays. To do this, it explores the space of $N \times k$ arrays by changing one entry at a time (its transformation), guided by the number of t-sets that are not covered (its fitness function).

For instance, the left side of Fig. 3 shows a random array with incomplete coverage of our SPL example: the two-sets [AUDIO, VIS.], [AUDIO, CC], [VIS., NO CC], and [NO VIS., CC] are missing. Its fitness is 4—the number of absent pairs.

Here the objective is a fitness of zero, so when a randomly selected modification produces the new array on the right side with fitness 3, simulated annealing will judge the move as advantageous. Neutral changes are also accepted, for they help the algorithm investigate more of the state space.

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

```
1  let A ← ∅;
2  let N ← ⌊(lower + 2 · upper)/3⌋;
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 + 2 · upper)/3⌋;
12 end
13 return A;
```

Fig. 2 The base outer search, a binary search

Before-Move CIT Sample (Fitness = 4)

Format	*Audio Visualizer*	*Closed-Captioning*
VIDEO	VIS.	CC
VIDEO	NO VIS.	NO CC
AUDIO	NO VIS.	NO CC
VIDEO	VIS.	CC

After-Move CIT Sample (Fitness = 3)

Format	*Audio Visualizer*	*Closed-Captioning*
VIDEO	VIS.	CC
VIDEO	NO VIS.	NO CC
AUDIO	NO VIS.	NO CC
VIDEO	NO VIS.	CC

Fig. 3 An example of neighboring arrays

Thus far the algorithm described is that used by Stardom (2001) and Stevens (1998) for building unconstrained covering arrays, but generalized for distinct v_i values and amenable to values of t other than two. There are two differences between it and the base algorithm this work began with.

First, unlike a typical implementation of simulated annealing, every invocation of the inner search stores its state in the same array, ignoring the rows at indices N and beyond when the array is too large and adding random rows when the array is too small. Thereby the progress made in one call to simulated annealing is used in the next. This change was incorporated in our 2003 work (Cohen et al. 2003b).

Second, the base algorithm supports constraints. Rather than dealing with variables that have different domain sizes, the base algorithm distills all constraints to a propositional formula over boolean variables. Although a domain of size d can be encoded in $\lceil \log_2 d \rceil$ bits, the base algorithm uses a different truth state to represent each value in the domain. For instance, the example formula from Section 2.3, $(\mathit{Format} = \text{VIDEO}) \vee (\mathit{Audio\ \ Visualizer} = \text{VIS.}) \vee \neg(\mathit{Closed\text{-}Captioning} = \text{CC})$, is converted to a boolean formula by creating two boolean variables for each column (because each v_i is two), replacing each equality predicate with the matching variable, and then adding more clauses that require each of *Format*, *Audio Visualizer*, and *Closed-Captioning* to have exactly one value (Cohen et al. 2007b). After this conversion, the formula is handed to an off-the-shelf SAT solver.

Then, the base algorithm enumerates all t-sets. Any t-sets that are unattainable under the given constraints are tagged during initialization and not counted as part of the fitness function.

To ensure that constraints are not violated during the search, a call to the SAT solver occurs before a new move is accepted. If a move would produce an assignment that does not satisfy constraints, it is rejected.

Figure 4 shows the specifics in pseudocode. At each step, the code randomly chooses a location 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 a simple policy called the death penalty (Coello Coello 2002). Consequently, the search never visits an infeasible state. But if constraints are satisfied, it computes the change in fitness (line 10). When the alteration improves or maintains fitness, the algorithm applies it and continues (lines 11–13), but it only accepts bad moves probabilistically (lines 14–16). The iterations continue until a stabilization criterion is met (line 3);

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 Δfitness ← countNoncoverage(A') − countNoncoverage(A);
11 |  |  if Δfitness ≤ 0 then
12 |  |  |  let A ← A';
13 |  |  else
14 |  |  |  with probability e^{−Δfitness/temperature} do
15 |  |  |  |  let A ← A';
16 |  |  |  end
17 |  |  end
18 |  |  let temperature ← cool(temperature);
19 |  end
20 end
21 return A;
```

Fig. 4 The base inner search, element-wise simulated annealing

in other words, the algorithm has found a solution or has spent its iteration budget. Then the array is returned (line 21).

3 Modifications

When the base algorithm is compared to an implementation of the AETG that supports constraints, mAETG, on realistic constrained CIT problems, it does not fare well (Cohen et al. 2007b). For small two-way inputs it builds arrays of similar or smaller size, though with an increase in run time. But on larger problems both the array size and the construction time are worse. The trouble seems to be the constraints; just forbidding constraint-violating moves does not give the search enough guidance.

Consequently, we ran preliminary experiments with a small real-world constrained CIT problem: sampling configurations of MySQL. MySQL was modeled by Fouché et al. (2009), although the version of the model published in their paper removes constraints that were caused by a fault. The version used here has 23 columns with two to five symbols in each column and 15 binary constraints. Typical array sizes are 25, 77, and 277 rows for $t = 2$, $t = 3$, and $t = 4$, respectively. On this model the base algorithm works fairly well building two- and three-way arrays, but is very slow for higher values of t.

During this exploratory phase we implemented two changes that made a notable difference in performance on the MySQL model. Therefore, they are labeled primary modifications. Later attempts to apply the revised algorithm to other CIT problems revealed further opportunities for improvement, born out of motivations akin to those prompting the primary changes—these we call refinements. In our earlier work we called the primary modifications "major," and the refinements "minor" (Garvin et al. 2009). We adopt this new terminology to make it clear that these categories were assigned before the evaluation, not as a consequence of our empirical findings (although our experiments do suggest that the primary modifications have a greater impact on performance).

Both primary modifications and refinements are presented in the following sections. For each change we detail the motivating observations and a resulting criticism of the base algorithm. A discussion of the change itself follows, along with comments on its impact.

3.1 Primary Modifications

The first of the primary modifications, *t*-set replacement, stems from attempts to speed up the generation of two-way samples without compromising array size. However, it causes the algorithm to yield unnecessarily large samples when *t* increases, so the second, one-sided narrowing, aims to achieve small arrays at higher *t*.

3.1.1 t-Set Replacement

Observations We began the investigation by logging data from several runs to get a picture of the implementation's behavior. One of our visualizations was fitness versus number of iterations for each invocation of the inner search. Except in trivial cases, the plots had the same general shape: that of Fig. 5. Though the fitness value eventually reaches zero, its progress is continually slowing.

When most of the *t*-sets are already packed into the array, it is hard for simulated annealing to make changes without disturbing that coverage. But the flattening curve persisted even when simulated annealing had many more rows than are necessary to produce a solution, as in Fig. 5 where it has more than 75 times as many rows as it needs.

The plot shows that the iterations are becoming less useful as time progresses, but it does not show what those iterations are doing. That question was answered by a

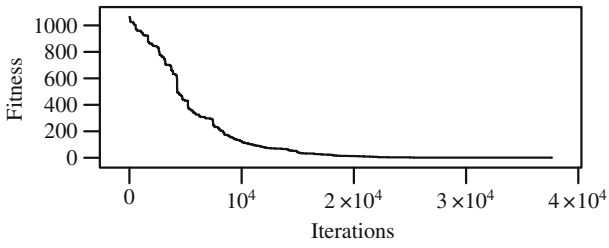


Fig. 5 Example plot of fitness vs. number of iterations of the base algorithm's inner search: 1,653 rows for two-way coverage of MySQL

separate logging process, which revealed these three activities, ranked from rarest to most frequent:

1. Choosing a symbol in a missing *t*-set and a row where that symbol satisfies constraints
2. Choosing a symbol in a missing *t*-set and a row where that symbol violates constraints
3. Choosing a symbol not in the missing *t*-sets

(Note that even in the first case a move could destroy enough other *t*-sets to worsen the fitness and be rejected.)

Clearly, if the first type of iteration is more frequent the search stands a better chance of making progress. Nonetheless, the search cannot always introduce a *t*-set directly, even if that *t*-set can be accommodated without violating constraints. The base algorithm's transformation function modifies just one symbol at a time and might encounter a constraint violation when it has written only some of the symbols in a *t*-set. Moreover, the search cannot predict these cases, which leads to the second behavior. The third activity is not a direct result of constraints, but its role—to vary the constraints that are satisfied by other elements in the same row and thereby change which moves are of the first sort and which are of the second—is only useful because of constraints.

Criticism The shortcoming that these symptoms point to is easier to see viewing simulated annealing as walking a digraph whose vertices are arrays and whose edges are single symbol substitutions. Then, states that violate constraints form obstacles to search progress.

For example, suppose that the array is to cover two-way interactions in the media player presented earlier. To make the search space small enough to illustrate, assume that every pair except [AUDIO, VIS.], [AUDIO, CC], and [VIS., CC] 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 [AUDIO, VIS., CC]. For illustration consider an initial state that is as far away as possible, at [VIDEO, NO VIS., NO CC]. This gives the state space in Fig. 6. Solid lines depict legal transitions; dashed lines indicate transitions that violate constraints. Note that every state also has three self-loops, not shown to improve readability.

Looking at just the lower and right parts of the graph, the path from [VIDEO, NO VIS., CC] through [AUDIO, NO VIS., CC] to [AUDIO, VIS., CC] is valid without constraints. But if simulated annealing attempts this route with constraints, it fails on the first transition and must instead detour via [VIDEO, VIS., CC]. In short, constraints decrease the connectivity of the state space, so the coverage heuristic, unaware of the obstacles, may be uninformative or even misleading. The bottom line is that simulated annealing must depend on its randomness to avoid obstacles where otherwise it could be more goal-oriented.

The Change We solve this problem with a single modification to our strategy: instead of a transformation function that overwrites individual elements of the array with random values, we use one that writes an entire missing *t*-set to a row. The states in the state space are the same, but the neighborhood is different.

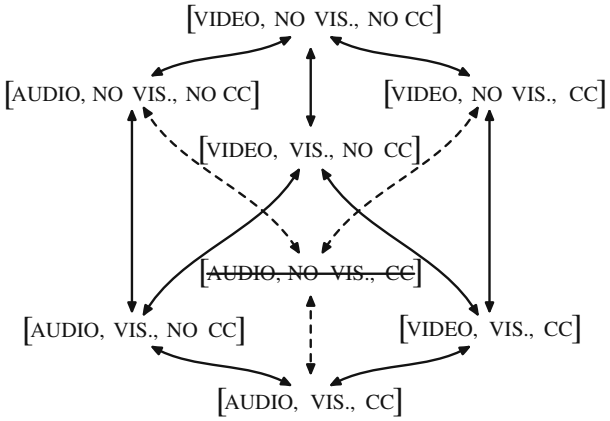


Fig. 6 The original search space

For instance, this effectively means that simulated annealing can take short excursions through infeasible states like [AUDIO, NO VIS., CC]; the constraint check doesn't happen until all of the t-set's symbols have been written to the array. In the example where $t = 2$, [AUDIO, VIS., CC] becomes directly reachable from [VIDEO, NO VIS., CC]. Hence, constraints should block the search less frequently.

With this altered space it is no longer as useful to select symbols outside of the missing t-sets. Thus, already covered t-sets are not candidates for insertion and the search is more goal oriented. For example, Fig. 7 illustrates how [VIDEO, NO VIS., NO CC] becomes AUDIO, VIS., NO CC when the two-set [AUDIO, VIS.] is added. But the reverse move is impossible. Moreover, at that point success is guaranteed; either [AUDIO, CC] or [VIS., CC] must be added, both of which lead to a solution.

Comments Returning to our original observations, Fig. 8 contains an example plot of fitness versus iterations after changing to t-set replacement (note the change of scale). Simulated annealing now has many options when it is far from a solution, but the alternatives dwindle as it nears an answer. The shrinking search neighborhood means the algorithm has less exploration to do, which offsets the increasing difficulty to include t-sets.

The graph's flat tail does shorten; in Fig. 5 roughly the last half of the iterations constitute the tail, whereas in Fig. 8 it appears that the tail is only a fifth of the total

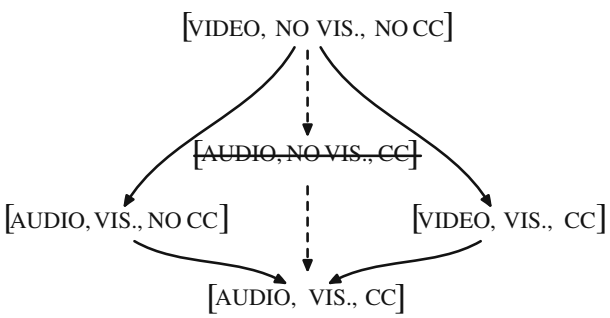


Fig. 7 The revised search space

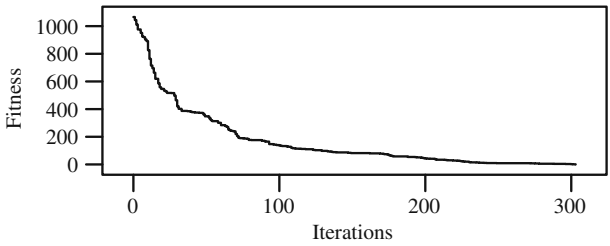


Fig. 8 Example plot of fitness vs. number of iterations of the base algorithm's inner search with *t*-set replacement: 1,653 rows for two-way coverage of MySQL

number of iterations. Moreover, the total number of iterations shrinks by an order of magnitude—much more than we expected and more than would have been achieved by simply eliminating the entire tail from Fig. 5.

3.1.2 One-Sided Narrowing

Observations Because *t*-set replacement makes the inner search faster on MySQL, our next step was to increase *t* from two to four. The program still finished quickly, but it returned arrays almost twice as large as those found by the base algorithm. This time the issue seemed to be the outer search. Every run showed a consistent pattern: early calls to the inner search failed; later ones succeeded. The first few calls at least should have succeeded lest the optimum array size be discarded.

A possible fix would have been to increase the iteration limit for the inner search so that it would not give up so quickly on the first array. But the outer search's two-part behavior hinted at a different issue. In particular, it was striking that an inner search's success seemed to depend more on how the last attempt fared than on the array size.

That led us to consider the inner search's sensitivity to its seeded start state: because the code does not rerandomize the common array between invocations of simulated annealing (Section 2.6.2), later calls have an advantage in accumulated progress. The outer search does not take that difference into account, but judges every result from an inner search as definitive.

Criticism The binary search supposes that the inner search accurately evaluates whether an array of a given size can be built. When it fails, the binary search never revisits this size again. However, the inner search is stochastic and depends on the preceding invocations. It might terminate before finding a solution that does in fact exist, especially when dealing with constraints.

The Change One-sided narrowing keeps the essence of a binary search, but abandons the faulty assumption that a failed inner search precludes a solution of a given size. The central idea in binary search is to narrow the range of candidate sizes as much as possible after each call to simulated annealing. Rather than narrowing from both sides though, one-sided narrowing only improves the upper bound because then 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 Fig. 9. A solid range represents a subrange being considered; a dashed range has been eliminated. Arrowheads indicate the points

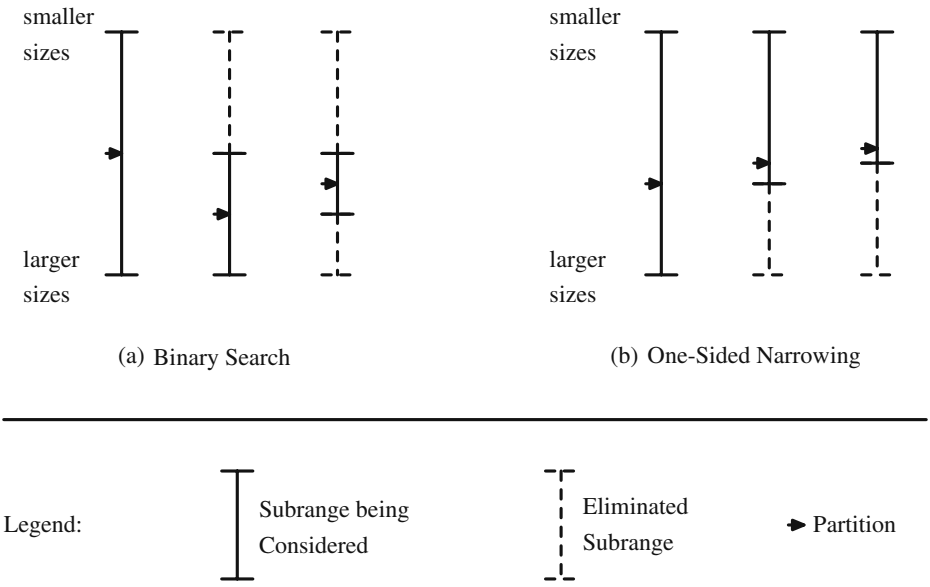


Fig. 9 Binary search versus one-sided narrowing

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 subrange 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.

Though the binary search is not sound 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.

The result is a three-layer search. The outermost search shown in Fig. 10 invokes the old outer search from Fig. 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 can recover the cost because each call to the inner search consumes fewer iterations.

Comments Note that false negatives may still be problematic, but less so than before: they slow the outer search's progress, but do not stop it. When we tried one-sided narrowing on the MySQL problem, each successive binary search was able to reach smaller array sizes, eventually getting within a few rows of the base algorithm's output.

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

```
let A ← ∅;
while upper ≥ lower do
    let A′ ← binarySearch(t, k, v, C, lower, upper);
    if A′ = ∅ then
        break;
    else
        let A ← A′;
        let upper ← rows(A′) − 1;
    end
end
return A;
```

Fig. 10 The new outer search, one-sided narrowing

3.2 Refinements Related to *t*-set Replacement

In Section 3.1.1 we modified the inner search to focus on just the symbols from missing *t*-sets, pointing to the diminished effect of constraints as the reason why the other symbols were no longer needed in the search transformations. Unfortunately, the truth of this claim depends on how severely the problem is constrained. In about a third of the problems we considered, just substituting *t*-sets produces a search whose new search space can cause it to get stuck before finding a solution. When substituting absent *t*-sets is not enough, the following three refinements offer a remedy without returning to the original state space.

3.2.1 Row Replacement

Observations Some problems with feature constraints, despite having a solution, are unsolvable for simulated annealing, under both the old and new search space. As a small example of how this can happen, consider Fig. 11. The two legal rows are [A, C, E] and [B, D, F], so the only constraint-satisfying changes are those that affect three symbols at a time. Except with *t*-set replacement when $t = 3$, the inner search will be stuck.

Beyond that, the directed search space of *t*-set replacement can also make solutions unreachable. The array in Fig. 12 shows a non-trivial example of this where $t = 2$. First note that the last two rows are identical, so one of them can be replaced

	α (A/B)	β (C/D)	γ (E/F)
1.	A	C	E
2.	A	C	E

subject to $[(\alpha = \text{A}) \wedge (\beta = \text{C}) \wedge (\gamma = \text{E})] \vee [(\alpha = \text{B}) \wedge (\beta = \text{D}) \wedge (\gamma = \text{F})]$

Fig. 11 A constrained array that neither element replacement nor *t*-set replacement can modify

	α (A/B)	β (C/D)	γ (E/F)	δ (G/H)	ε (I/J)	ζ (K/L)
1.	A	C	E	G	I	K
2.	A	C	E	H	J	K
3.	A	C	F	G	J	K
4.	A	C	F	H	I	L
5.	A	D	E	G	J	L
6.	A	D	F	H	I	K
7.	A	D	F	H	I	K

$$\text{subject to } [\neg(\alpha = \text{B}) \vee (\beta = \text{D})] \wedge [\neg(\alpha = \text{B}) \vee (\gamma = \text{F})] \wedge [\neg(\alpha = \text{B}) \vee (\delta = \text{H})] \wedge [\neg(\alpha = \text{B}) \vee (\varepsilon = \text{J})] \wedge [\neg(\alpha = \text{B}) \vee (\zeta = \text{L})]$$

Fig. 12 A constrained array that *t*-set replacement cannot modify

without any loss of coverage. Next, the only pairs missing are those that involve B, and feature B requires features D, F, etc., so that it can only exist in the row [B, D, F, H, J, L]. Because all of the attainable pairs involving B fit into one row, there is certainly room to construct a constrained covering array. But none of the absent pairs can be inserted without violating constraints. On the other hand, if the search were substituting arbitrary *t*-sets instead of just missing ones it could make the last row [A, D, F, H, J, L] and then insert the B.

It seemed that this problem would be difficult to encounter in practice. However, when we tried our algorithm on problems besides MySQL, we found several instances where *t*-set replacement became trapped or nearly so.

Criticism Again we think of the search as a graph walk. In the extreme cases, constraints disconnect the search space or form sinks, trapping the search and rendering the problem unsolvable.

The Change Row replacement adds a meta-heuristic that detects when the search gets stuck and offers simulated annealing an additional way to escape. Because it is prudent to release the search from states with few outgoing edges as well as those with none, the detection code does not check every possible move for infeasibility. Instead, the program counts consecutive failures to satisfy the constraints and switches strategies when the count gets abnormally high.

As the first example in the this sub-section shows, the algorithm might have to change as much as a row to fit the target *t*-set. So it writes the *t*-set to the row and then randomly chooses the row's other symbols subject to constraints. It returns to the normal search space immediately after.

Comments With row replacement it is impossible for any state to be disconnected from a solution, provided one exists. We can prove this by choosing a reference solution and inducting on the number of *t*-sets that occupy a row in the current state that they also occupy in the reference array. Let us call this number *n*.

For the base case, let *n* be at its maximum—the number of attainable *t*-sets. Any *t*-sets counted by *n* are covered, so the state is a solution. It is obviously connected to itself.

For the inductive hypothesis, let us assume that a solution can be reached from all states having n larger than at the current state. If the current state is a covering array, it is certainly connected to a solution. Otherwise we can choose a missing t-set and determine a row in the known solution that contains it. If it can be placed in the same row of the current array there is a t-set replacement move that connects the current state to one with larger n; by transitivity the state is connected to a solution. If not, a sufficient number of repeated attempts will trigger row replacement. One of the possible moves under row replacement is to copy the row from the reference solution, in which case n must increase. Again by transitivity and the inductive hypothesis, at least one path to a solution exists.

Using the principle of complete induction, the proof is concluded. Row replacement guarantees that, unless given an unsolvable problem, simulated annealing always has a chance to build a covering array.

3.2.2 SAT History

Observations Row replacement randomly generates a new row in much the same way that rows in the initial solution are randomly generated. The base algorithm's implementation starts with either an empty row (for the initial state generator) or one populated with just the target t-set (for row replacement). It adds random symbols one column at a time, asserting the corresponding literals in the SAT solver. If a symbol causes a contradiction, the algorithm undoes its placement and retries the column. In the worst case this process could continue forever.

Criticism We note that 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.

The Change The algorithm remembers symbols that constraints reject and does not try them again.

3.2.3 Row Sorting

Observations When we ran the program on problems other than MySQL, constraints caused some t-sets to be far more difficult to cover than others. During the early iterations of the inner search, the program will try to cover these t-sets and fail, but subsequent attempts might choose a less troublesome t-set, and row replacement will not be triggered. It is not until later, when everything that is missing from the array's coverage is hard to fit, that constraint-violating moves are tried frequently enough for row replacement to take over. At that point, it often cannot help but dislodge the search's earlier work.

It makes more sense to put the disruptive row replacements early in the search and save the easier t-set substitutions for last. Or better yet, to include the difficult t-sets in the start state.

Thus, row replacement highlights another opportunity for the algorithm to learn from previous inner searches: if they have managed to include troublesome t-sets, it should keep the relevant rows for the next start state.

Criticism 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 omitted while redundant rows are preserved.

The Change In the base algorithm the rows at smaller indices are used in later inner searches while those at higher indices are ignored (Section 2.6.2). Hence, the algorithm should move valuable rows—those that cover difficult t-sets—to earlier locations. Based on the idea that difficult t-sets will appear infrequently, the code labels all t-sets covered just once as valuable. It then sorts the rows by how many valuable t-sets each covers.

Comments The motivation for this change is specific to constrained problems, but as a side-effect row sorting shifts redundant rows to the end of the array where the next search iteration will drop them. This behavior can be helpful even when problems do not involve constraints. The advantage diminishes as the array gets smaller though, for then even easily-covered t-sets may only occur once.

3.3 Refinements Related to One-Sided Narrowing

We also identified other opportunities to take advantage of the fact that the inner search is stateful. These three changes are all designed to arrange calls to the inner search for better performance.

3.3.1 Iteration Bounding

Observations As pointed out in Section 3.1.2, the extra cost of one-sided narrowing must be offset by a smaller iteration limit in the inner search. Trial runs to see how far the iteration limit could be reduced showed that reductions at large array sizes make little difference in the algorithm's final outcome. In that case, it is still possible to find smaller arrays in each binary search, and one-sided narrowing continues. On the other hand, a smaller iteration budget at low N is detrimental; the false negatives stifle all progress, and the algorithm terminates.

Criticism Lower iteration limits are possible at large values of N under one-sided narrowing, but not when N is smaller. The algorithm treats all calls to the inner search equally, so it cannot capture this difference.

The Change Because it is difficult to predict what constitutes large and small array sizes, an adaptive approach to selecting the limits makes more sense than a fixed policy. Therefore, the algorithm sets the stopping criterion on the fly. Specifically, it limits early invocations to a small number of iterations, I, and delays increasing this quantity as long as possible—until the search can no longer find a solution. Then it doubles the limit and retries. The number of consecutive retries is also bounded so that the algorithm eventually terminates.

Comments On most problems this policy reserves the majority of search iterations for small arrays where they are most needed, as we intended. But on large inputs it will sometimes take this trend too far, dedicating iterations to sizes where it cannot find an array. In Section 6, we discuss possible fixes.

3.3.2 Informed Partitions

Observations We noticed that the number of iterations needed by simulated annealing was not just correlated with the success or failure of the previous invocation, but also with the change in N. As we wanted to avoid both the bookkeeping overhead of short-running calls and the time lost waiting for failing searches to reach their iteration limit, the algorithm ought to always choose N at the smallest value where success can still be expected. However, the binary search picks its partitions blindly.

Criticism The outer search has information besides the success or failure of the last inner search: in the case of success the number of iterations taken to reach an answer indicates the difficulty the inner search is experiencing; for failures difficulty is reflected in the number of t-sets that could not be covered. With this knowledge it could more accurately predict the smallest N at which a subsequent success is likely.

The Change Based on these two metrics the algorithm estimates the best choice for the next N. If the estimate is a valid search partition, it is kept. Otherwise the algorithm uses a weighted average of the lower and upper bounds, much like the original search.

At present, it is unclear how these estimates should be calculated. By trial-and-error we developed ad-hoc heuristics that correspond to two general rules:

- After a failure, increase N by an amount proportional to the number of t-sets not covered. That is, $\Delta N \propto \text{countNoncoverage}(A)$.
- After a success, decrease N by an amount proportional to the logarithm of the ratio of iterations used. Put another way, if i out of I iterations were taken, $\Delta N \propto \log(i/I)$.

As informed partitions did not affect performance as significantly as the primary modifications in preliminary experiments, we leave a thorough evaluation of alternative heuristics to future work.

Comments While informed partitions certainly increased the inner search's rate of success, it seemed on further observation that failures also play an important role: they pack more t-sets in the low-index rows so that high-index rows can be changed without as much disturbance to coverage. Again, further study is needed to characterize the trade-offs entailed by different partitioning policies.

3.3.3 Bounds Revision

Observations When generalizing from MySQL to other problems, we found an example where the algorithm gave up too early because it had reached the theoretic lower bound for a covering array without feature constraints. But because of the constraints, the minimum was actually smaller. Similarly, we encountered large problems where simulated annealing never had enough rows to find its first array.

Table 1 Parameters used in the experiments

Parameter	Old set	New set
Initial lower bound	$\max_{S\subseteq\{1\ldots k\},\lvert S\rvert=t} \prod_{i\in S} v_i$	$\max_{S\subseteq\{1\ldots k\},\lvert S\rvert=t} \prod_{i\in S} v_i$
Initial upper bound	$5(\max_{i=1\ldots k} v_i)^t$	$5(\max_{i=1\ldots k} v_i)^t$
New lower bound when *lower* is met	N/A	$lower - 5$ if $lower > 5$, $lower/2$ otherwise
New upper bound when *upper* cannot be met	N/A	$2 \cdot upper$
Number of consecutive failed t-set replacements before row replacement	N/A	32
Partition adjustment after success	N/A	$-\log_2(i/I)$
Partition adjustment after failure	N/A	countNoncoverage$(A)/2$
Weight of *upper* in fallback partition	2/3	2/3
Initial temperature	0.2	0.5
Cooling	−0.0001% every 10 iterations	−0.0001% every iteration
Inner search iteration limit	100,000	4,096
New inner search iteration limit when the binary search restarts	N/A	$2 \cdot I$
Number of failed restarts to trigger termination	N/A	2

i is the number of iterations used in the last inner search; I is the number of iterations the last inner search was allowed

Criticism 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.

The Change We add checks to the outer search for two cases: the annealing needs more space to work because it can't find a solution when N is the initial *upper*, or it built one with $N = lower$ and needs to check the minimality of N. The resulting bounds adjustment is a parameter to the algorithm; see Table 1 in Section 4 for our choices.

4 Evaluation

In our previous work we performed a preliminary set of experiments to understand the effectiveness of some of these changes (Garvin et al. 2009). In this section, we expand on that work and more systematically and thoroughly evaluate our modifications on programs other than MySQL. Our goals are to understand how well our modifications work on realistic problems, how they compare with other state-of-the-art algorithms for constrained CIT, and how they scale to higher strength. We also want to understand if we have retained the effectiveness of the original algorithm on unconstrained CIT problems. Our evaluation is structured as follows: We begin by presenting four research questions in Section 4.1. Then we outline our experimental design in Section 4.2 while the results and analysis appear in Section 4.3. Finally, we discuss threats to validity.

In addition to the content here, the source code and evaluation artifacts are available for download at http://cse.unl.edu/citportal/tools/casa/, along with some explanatory documentation.

4.1 Research Questions

Ultimately we are interested in whether or not our suggested changes are useful in real constrained CIT. Our first objective is therefore to measure the impact of those changes:

RQ1 (Effect of Modifications) Do the modifications reduce the total computation time to generate and use a constrained CIT sample?

Once we understand the effects of our changes, we then need to determine how they affect simulated annealing's standing with respect to other algorithms. The second question is then:

RQ2 (Comparison to a Greedy Approach) Does the revised simulated annealing save total computation time versus a greedy approach?

Then, some CIT problems are unconstrained and a complete evaluation must ask whether the trends discovered in RQ1 and RQ2 still hold in those cases:

RQ3 (Unconstrained Problems) Do the changes alter simulated annealing's performance on unconstrained problems as they do with constrained inputs?

Finally, we ask whether the modifications result in an implementation that is sufficiently consistent for use in the field:

RQ4 (Robustness) Does simulated annealing with the proposed modifications yield consistent sample sizes and run times?

4.2 Experimental Design

For our experiments we ran each implementation of the algorithm being tested five times (50 times for RQ2) per benchmark problem and averaged the run times and array sizes. Our data were collected under Linux on 2.4GHz Opteron 250s. Specifics on the benchmarks and experimental variables follow.

4.2.1 Benchmarks

Our benchmark problems for constrained CIT are the 35 system models presented in our 2008 work (Cohen et al. 2008). The models are meant to represent a realistic set of highly-configurable software systems. Five of these CIT problems are taken from case studies on five real-world highly-configurable systems: the SPIN verifier and SPIN simulator, GCC, Apache, and Bugzilla. The other thirty were generated synthetically from the characteristics of these five. More details about the case studies, modeling process and simulated samples can be found in that paper.

For the unconstrained problems we also selected problems from the literature where simulated annealing was more effective at finding CIT samples than its greedy counterparts. Most of these problems are very regular, with all of the v_i values being the same, but we also chose nine problems with a variety of domain sizes for

the columns. The smallest known arrays for these benchmarks range from nine to 3,654 rows.

4.2.2 Independent Variables

The experiments' independent variable is the implementation used to generate the constrained covering arrays. Ideally we would consider each of the eight suggested changes as an independent variable and examine all combinations, but many combinations have too long a run time for such an evaluation to be practical, so our experiments concentrate on the primary modifications and their refinements separately.

First, we examine five different implementations of simulated annealing where the refinements are grouped together as one change:

- Control is a version of the base algorithm refactored (and validated against the original base algorithm) to support all of the suggested alterations. It also has the zChaff (Malik 2004) SAT solver replaced with MiniSAT (Eén and Sörrenson 2007) for comparability with the greedy implementations, which also use MiniSAT.
- Grouped is the same as Control with code for the grouped refinements enabled.
- TSR builds on Grouped by adding t-set replacement.
- OSN extends Grouped with one-sided narrowing.
- All combines all of our proposed changes.

Second, we consider 64 versions with every possible combination of refinements, but for all of these versions we keep both t-set replacement and one-sided narrowing enabled.

Note that unlike our preliminary work on this problem (Garvin et al. 2009) we kept a common code base between versions to minimize the effects of extraneous implementation details. In particular, the equivalent to Control in our earlier experimentation chose the first partition of its binary search randomly, which put it at an unfair disadvantage.

Aside from the algorithmic changes, only one other factor differs between the implementations used: the base algorithm's parameters for simulated annealing are ill-chosen when the primary modifications are enabled.

The parameters for Control came from our prior work (Cohen et al. 2003b, 2007a). There, for the version without feature constraint support, we chose settings that fared well on a variety of inputs with experimentation across a wide range of values, tuning for generality instead of performance on a specific subject (Cohen et al. 2003b). When feature constraint support was added, we increased the cooling rate and allocated fewer iterations to the inner search; both of these changes helped the search complete within a reasonable amount of time despite the more complicated search space (Cohen et al. 2007a). This second tuning was done informally. The resulting values are summarized in the middle column of Table 1.

We re-tuned the parameters for the primary modifications with the MySQL benchmark at $t = 4$. These values are in the right column of Table 1.

Starting with Control's parameter set we first varied the iteration count (with possible values being the powers of two from 256 to 131,072) and the number of retries (considering the integers from 1 to 4). The combination of 4,096 iterations and two retries yielded the smallest run times among the settings that give the best sizes,

so we fixed those two parameters. Next we varied the initial temperature (examining from 0.1 to 0.9 in steps of 0.1) and the cooling rate (with possible per-iteration decreases of −0.01%, −0.001%, −0.0001%, and −0.00001%), choosing 0.5 and −0.0001% for similar reasons. After that we varied the weight of the upper bound used in the partitions when the informed partitioning process has no suggestions; though we considered values 0.5–0.9 in increments of 0.1, no change outperformed the former weight of 2/3, so we retained it.

The remaining parameters were set more informally. The initial bounds on N we kept the same because, in our experience, constraints rarely drive the minimum N outside this range. In the cases we did know of, the minimum was never more than two or three rows less than the initial lower bound, so we made the adjustment to the lower bound decrease its value by a slightly larger amount, five rows. The special case of halving the bound was added as a safeguard against pathologically small problems where a fixed decrease could result in an attempt to create negative array sizes. For the upper bound increase we needed more than a 50% increase so that 2/3 of the new value would always yield a size larger than any that had formerly been tried. A 100% increase was usually sufficient for the anomalous examples we had at hand. The number of failed t-set replacements before switching to row replacement was obtained by observing typical values of this count when the search was not stuck. We usually saw no more than eight consecutive failures, so conservatively we set the limit four times as high. To determine the partition adjustments we tried several functions on the quantity i/I and a range of integers dividing countNoncoverage(A). The doubling of iterations when the binary search restarts was part of the original design of one-sided narrowing, and we did not experiment with different rates of growth.

Finally, for each of Grouped's parameters we used the old value whenever it was available and the new value otherwise.

As the competing greedy implementations we used the four constraint-supporting variations on AETG that were presented by Cohen et al. (2008), always choosing the variation that fared the best against our algorithms. See Section 4.3.2 for more detail on the selection criterion.

4.2.3 Dependent Variables

The evaluation's dependent variables are the size of the covering array produced and the time taken to create it, as well as a function of these two: total computation time to build and use the covering array. The total computation time is modeled as the array generation time incurred up-front plus the average per-configuration testing time incurred once for each row of the array:

$$\begin{aligned}\text{total computation time} = {} & (\text{array generation time}) \\ & + (\text{mean per-configuration test time}) \\ & \cdot (\text{number of rows}) \end{aligned} \tag{1}$$

Because the average time to test each configuration is unknown for these problems, we leave that variable unspecified and qualify our conclusions with the range of values for which they are valid. Specifically, we compute the *break-even point* of one implementation with respect to another as the minimum per-configuration testing

time at which the first implementation saves computation time over the second. As a formula,

$$\begin{aligned}&\text{break-even point of X with respect to Y}\\&\quad=(\text{array generation time of X}-\text{array generation time of Y})\\&\qquad/\,(\text{number of rows created by Y}-\text{number of rows created by X})\end{aligned}\tag{2}$$

Note that two samples may have different mean per-configuration test times, but we have assumed equality. Intuitively we would not expect large deviations in the means, because the differences in the time to test a configuration are rooted in the differences in chosen features, and both samples must cover a similar diversity of feature combinations. Nonetheless, we also quantify the impact of distinct means in our analysis.

4.3 Results and Analysis

For each research question we present our results and a comparison based on total computation time to build and use the covering arrays.

4.3.1 Effect of Modifications (RQ1)

Results The results for the first experiment are listed in Table 2. We show data for five of the simulated annealing implementations and report first array size and then time in seconds. All of the results are for strength $t = 2$. In the left columns 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 for each i there are y_i columns with x_i symbols to choose from. A constraint term, written in the same format, means that y_i constraints involve x_i symbols. After the benchmark names and these descriptions, the table gives the sample sizes (N) and run times obtained with each implementation. The smallest values are shown in italic. The last row contains coefficients of variation, which are discussed under RQ4.

Looking first at the influence of the grouped refinements, we see smaller array sizes and run times. Note however that both CONTROL and GROUPED give similar sizes on every input except Benchmark Problem 15, so these changes cannot be said to have had much effect in general. Their consequences for time are more consistent, but not universal.

Next, we compare the columns for GROUPED with those for TSR and OSN. t-Set replacement dramatically reduces run time at the expense of a few rows per benchmark problem. One-sided narrowing is more difficult to characterize, because it does not behave consistently from problem to problem. It seems to be a generally bad choice from an array size perspective. On run time it can sometimes help, for its updated parameters permit shorter inner searches, but more often it takes longer. The cost can be extreme; witness Benchmark Problems 20 and 28 which together averaged a month per run and accounted for more than half of OSN's total.

But when the two primary changes are combined in the ALL implementation we obtain sizes that resemble those produced by GROUPED and run times on the order of TSR's. So it appears that one-sided narrowing is only useful in combination with t-set replacement.

Table 2 Average sizes and times over 5 runs for constrained two-way problems

Name	Model	Constraints	Size					Run time (s)				
			CONTROL	GROUPED	TSR	OSN	ALL	CONTROL	GROUPED	TSR	OSN	ALL
SPIN-S	$2^{13}4^{5}$	2^{13}	20.2	19.6	20.0	20.8	*19.4*	414.94	407.27	*2.08*	159.15	7.23
SPIN-V	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	33.2	*32.6*	43.2	49.4	36.8	4,010.73	4,710.78	*27.58*	1,080.60	60.03
GCC	$2^{189}3^{10}$	$2^{37}3^{3}$	19.8	19.6	26.8	*19.2*	21.8	36,801.39	41,181.63	*717.54*	366,713.76	1,708.92
Apache	$2^{158}3^{8}4^{4}5^{1}6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	32.6	*31.2*	34.8	38.8	33.0	23,589.09	18,326.97	59.90	27,274.91	*59.35*
Bugzilla	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	*16.0*	*16.0*	18.0	17.0	16.4	58.04	45.90	*4.14*	322.11	6.08
1.	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	39.6	39.4	43.0	55.6	*39.0*	10,596.51	13,190.71	*26.37*	12,581.48	80.82
2.	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	*30.0*	*30.0*	32.6	33.0	31.4	1,888.95	1,163.55	*20.16*	6,085.74	22.59
3.	$2^{27}4^{2}$	$2^{9}3^{1}$	*18.0*	*18.0*	*18.0*	*18.0*	*18.0*	506.12	629.50	*2.80*	60.58	5.43
4.	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	25.6	*20.2*	22.0	20.8	21.6	2,145.45	690.53	*7.27*	3,282.58	143.89
5.	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	48.4	49.0	52.0	80.6	*47.0*	61,775.54	48,312.01	*122.17*	405,339.65	816.86
6.	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	*24.0*	*24.0*	*24.0*	27.6	24.2	603.51	414.45	*8.88*	3,241.36	20.98
7.	$2^{29}3^{1}$	$2^{13}3^{2}$	*9.0*	*9.0*	*9.0*	9.2	*9.0*	378.11	655.98	*1.51*	25.66	2.12
8.	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	41.4	42.2	44.2	98.6	*40.4*	18,046.28	12,355.77	*39.04*	149,367.29	121.22
9.	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	30.0	30.0	30.0	*20.0*	*20.0*	11.59	*3.23*	3.33	418.29	26.37
10.	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	45.0	45.6	48.8	73.4	*44.4*	32,034.07	24,105.66	*59.69*	16,511.05	322.95
11.	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	43.2	43.8	45.8	57.6	*41.4*	15,605.26	11,467.99	*28.10*	13,408.84	380.83
12.	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	40.2	*40.0*	43.4	50.0	41.2	23,708.98	26,328.56	*56.71*	549,157.34	109.31

13.	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	*36.0*	36.2	37.4	38.6	36.6	4,721.45	5,513.18	*40.88*	24,168.64	56.44
14.	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	37.4	36.8	39.6	43.4	*36.4*	4,962.51	6,915.28	*25.43*	27,614.18	76.75
15.	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	67.0	*30.0*	31.8	39.8	31.0	6,032.95	1,231.50	*9.75*	1,594.78	14.26
16.	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$	*24.0*	*24.0*	*24.0*	25.8	24.2	1,501.38	254.08	*10.16*	3,649.28	18.98
17.	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	*38.8*	39.4	43.0	53.8	40.0	22,742.49	22,096.80	*46.15*	8,306.83	99.34
18.	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	42.6	43.6	46.6	63.4	*42.0*	36,907.75	18,813.12	*50.77*	15,663.36	245.32
19.	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	49.6	49.6	51.8	77.0	*47.6*	89,240.98	62,418.58	*124.46*	169,470.59	747.64
20.	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	55.4	54.8	57.8	67.2	*54.2*	41,527.58	38,089.49	*86.11*	1,177,089.69	425.07
21.	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	36.6	36.6	38.6	52.2	*36.0*	4,064.05	3,261.35	*18.40*	517.35	72.37
22.	$2^{72}3^{4}4^{1}6^{2}$	$2^{31}3^{4}$	38.0	*36.0*	*36.0*	39.0	*36.0*	848.69	191.24	*8.91*	2,168.70	9.87
23.	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	18.0	18.0	18.0	*12.0*	13.0	0.70	*0.25*	0.50	109.24	10.45
24.	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	43.4	43.8	47.4	60.8	*42.6*	23,949.69	22,376.39	*37.15*	78,771.41	251.14
25.	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$	49.8	49.8	53.4	76.8	*48.4*	33,822.78	25,175.35	*49.28*	24,897.32	363.73
26.	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	*29.8*	30.4	35.8	39.8	30.8	12,286.50	9,893.79	*22.66*	43,046.04	82.16
27.	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	*36.0*	*36.0*	37.0	*36.0*	36.2	368.33	300.34	*8.46*	3,142.71	13.41
28.	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	51.8	53.4	56.0	59.6	*51.2*	93,493.65	59,797.45	*113.67*	1,314,793.81	578.47
29.	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	29.0	*28.8*	31.4	34.0	29.2	20,088.01	23,313.04	*41.77*	182,819.54	68.52
30.	$2^{73}3^{3}4^{3}$	$2^{20}3^{2}$	*18.2*	*18.2*	20.2	19.0	18.8	4,285.60	5,481.25	25.74	10,136.84	*24.68*
Sum			1,217.6	1,175.6	1,261.4	1,527.8	*1,159.2*	633,019.65	509,113.01	*1,907.52*	4,642,990.67	7,053.57
Coefficient of variation (%)			2.9	0.4	0.7	5.8	0.6	4.9	8.6	9.5	64.5	25.8

The refinements are analyzed in Table 3. We originally conducted a full factorial experiment on those six changes, but 42.3% of the runs with versions lacking row replacement timed out because they repeatedly entered inescapable, non-solution states (see Section 3.2.1). Thus, the table only presents data from the 32 versions with row replacement. For each refinement we divide these versions into two groups of 16, the first group having the refinement and the second group lacking it. Then, within each group we average the versions' performance on the 35 benchmark problems. The improvements afforded by each modification are also listed.

Italicized figures in Table 3 show where we detected a significant difference. Because our data exhibits neither normal distributions nor homogeneous variances, we use the two-tailed Mann-Whitney-Wilcoxon test, which is a nonparametric test for estimating the likelihood that one distribution is stochastically greater than the other. In our context, if the test detects a significant difference, then there is a high probability that a run with the refinement enabled will compare with a run lacking the refinement in the same way that the means in Table 3 compare. To obtain a 95% confidence level for the entire determination of significant effects we reject the test's null hypothesis at the threshold $p < 0.01$, which includes a Bonferroni correction for the fact that we apply the test repeatedly.

Only two refinements are significant: row sorting significantly affects the distribution of array sizes ($p = 2.27 \cdot 10^{-3}$), and iteration bounding significantly influences the distribution of both array sizes ($p = 3.67 \cdot 10^{-3}$) and run times ($p = 3.33 \cdot 10^{-9}$). For completeness, we note that the same test applied to Table 2 identifies both *t*-set replacement as one-sided narrowing are significant to sample size and sample generation time.

Analysis As described in Section 4.2.3, practitioners are not interested in size or run time per se, but in their impact on the total computation time to make and use a constrained covering array. Naturally, if an implementation yields a smaller array size in less time than a competitor, it is the better alternative regardless of the per-configuration testing time. By that reasoning, ALL outperforms GROUPED, which beats CONTROL, an algorithm clearly better than OSN. It is only the ranking of TSR that depends on how quickly the test suite can be executed. Using (2), TSR is the best choice if on average a test suite can finish in under 50 s, it is second to ALL if the test suite needs less than 5,911 s, in third place after CONTROL until 14,409 s, and fourth otherwise.

Table 3 Average sizes and times over 16 versions for the 35 constrained two-way problems

Refinement	Size			Run time (s)		
	With refinement	Without refinement	Improvement	With refinement	Without refinement	Improvement
SAT history	1,201.4	1,200.6	−0.8	20,335.21	22,167.41	1,832.20
Row sorting	*1,180.8*	*1,221.1*	*40.3*	17,509.06	24,993.57	7,484.51
Iteration bounding	*1,175.4*	*1,226.5*	*51.1*	*38,193.27*	*4,309.36*	*−33,883.91*
Informed partitions	1,207.2	1,194.7	−12.5	18,898.81	23,603.81	4,705.00
Bounds revision	1,193.8	1,208.1	14.3	20,841.05	21,661.57	820.51

To answer RQ1 with regards to the primary changes, *t*-set replacement does reduce total computation time while one-sided narrowing lessens it if configurations cannot be tested quickly—less than 50 s on a 2.4GHz Opteron 250s. Few test suites for configurable systems run this quickly (Memon et al. 2004; Qu et al. 2008), so the common case is best treated by incorporating all of the changes.

However, it must be noted that the difference in mean per-configuration testing time is an important factor in these observations: if the sample produced by ALL has a mean that is more than 5% higher than the mean for CONTROL, then the latter becomes preferable for long-running test suites. For future work we should determine how this compares to the variance among CIT samples in practice.

Based on the results for the refinements we can add further observations. Row replacement is almost essential: without it the search gets stuck far too often to be useful in practice. Row sorting is the next most important. By freeing up iterations from the recovery of difficult-to-cover *t*-sets, row sorting helps simulated annealing reach smaller samples, possibly in less time. Iteration bounding is an even more effective way to reduce the final sample size, but it comes at the cost of run time. Therefore it only makes sense when the per-configuration testing time is large enough to warrant the extra time spent up-front.

The other refinements were less effective. SAT history is guaranteed to improve the run times except in pathological cases, but the experimental results show that those gains constitute only a small portion of the total execution time. As we guessed in Section 3.3.2, the speed-up from informed partitions comes at a cost in array size, though neither the speed-up nor the expense were significant. Bounds revision would perhaps be a significant factor on a set of benchmarks where constraints lowered the theoretical minimum size, but it made little difference on the benchmark problems we derived from real-world systems.

Follow-up Experimentation We also checked whether the reductions in total computation time were enough for a higher-strength run of ALL to fare favorably against a two-way run of CONTROL. Table 4 shows the higher-strength data collected with one run of ALL on the five real-world problems; figures from Table 2 that report average run times for $t = 2$ are repeated for reference. At $t = 4$ some runs timed out after 27 days (2.3 million s); N/A is listed.

For SPIN-S, SPIN-V, and GCC at $t = 3$ the ALL implementation managed shorter run times than CONTROL with $t = 2$, so the changes were significant enough to buy higher-strength coverage in the same amount of array generation time. Of course, because the three-way samples are larger, the modifications can only save total

Table 4 Average size and times with all changes on constrained *t*-way problems

Name	Size			Run time (s)		
	$t = 2$	$t = 3$	$t = 4$	$t = 2$	$t = 3$	$t = 4$
SPIN-S	19.4	98	345	7.23	100.46	8,637.04
SPIN-V	36.8	232	N/A	60.03	2,380.33	N/A
GCC	21.8	94	N/A	1,708.92	31,597.03	N/A
Apache	33.0	177	N/A	59.35	55,406.70	N/A
Bugzilla	16.4	61	201	6.08	174.52	45,680.22
Sum	127.4	662	N/A	1,841.61	89,659.04	N/A

computation time for quickly running test suites. Again using (2), the break-even point for SPIN-S is 4.04 s, it is 8.20 s for SPIN-V, and on the largest problem, GCC, it is as high as 70.14 s.

4.3.2 Comparisons to a Greedy Approach (RQ2)

For the second experiment we discard all but the leading variations on simulated annealing—TSR and ALL. Consequently, the programs run quickly enough that we can afford 50 runs rather than just five. Because we have already published figures for several variants of mAETG under these same experimental conditions (Cohen et al. 2008), we drew data from that work. Rather than compare to all of the greedy implementations, for each problem we choose the mAETG variant that saves computation time versus ALL for the broadest range of configuration testing times. This amounts to maximizing the break-even point of ALL with respect to mAETG, for mAETG always finished faster than ALL. The lone tie on benchmark 22 was broken by choosing the smaller array size.

Results Table 5 gives the results of the second experiment in the same format as described in Section 4.3.1. The middle group of columns gives the break-even points for each benchmark. For instance, the 5.31 in this group's upper right-hand corner indicates that ALL is preferable to TSR on SPIN-S if the per-configuration testing time is more than 5.31 s. In the same column the N/A indicates that ALL never saves total computation time over TSR on benchmark 7. Also, in the last row the break-even points are computed from the sums over all 35 problems. As with Table 2, the coefficients of variation will be covered in our discussion of RQ4.

Simulated annealing almost exclusively finds arrays of smaller size, especially if we contrast mAETG with ALL. Over the set of 35 samples ALL needs 25% fewer configurations to achieve the same CIT goals, even if we had chosen mAETG variants according to array size. For run time the opposite trend holds: mAETG always finishes before ALL, on average more than six times faster. TSR is sometimes competitive with mAETG though. This trade-off is consistent with the characterization of meta-heuristic versus greedy algorithms on unconstrained problems.

Analysis The break-even columns captures how the trade-off of run time for array size affects total computation time. Despite its longer time to generate covering arrays, TSR outperforms mAETG on most problems whenever a configuration takes more than a fraction of a second to execute. ALL shows the same pattern, but configurations must generally take 20.40 s before it is a worthwhile alternative to mAETG. That only one N/A appears in the "ALL vs. TSR" column of Table 5 confirms our expectations from RQ1: ALL also betters TSR if configurations cannot be tested quickly.

Importantly, an appropriately chosen version of simulated annealing rarely demands extra computation time unless the per-configuration testing time is extremely short—less than 1.31 s. In contrast, according to the data in Table 2 the CONTROL implementation needs configurations to require 32 mins per test suite before it saves computation time over mAETG. So the modifications have significantly improved simulated annealing's standing with respect to mAETG.

We further note that all of the break even points that are defined for TSR versus mAETG continue to exist even if the configurations in TSR's sample need

Table 5 Average size and times over 50 runs for constrained two-way problems

Name	Size			Run time (s)			Break-even (s/cfg.)			Coefficient of variation in size (%)			Coefficient of variation in run time (%)		
	mAETG best	TSR	ALL	mAETG best	TSR	ALL	TSR vs. mAETG best	ALL vs. mAETG best	ALL vs. TSR	mAETG best	TSR	ALL	mAETG best	TSR	ALL
SPIN-S	27.0	20.6	*19.4*	*0.2*	2.2	8.6	0.32	1.11	5.31	3.3	5.7	3.4	4.2	40.4	75.6
SPIN-V	42.5	42.9	*36.8*	*11.3*	32.5	102.1	N/A	15.93	11.44	3.0	6.4	6.3	3.0	59.5	121.6
GCC	24.7	25.8	*21.1*	*204.0*	661.7	1,902.0	N/A	471.67	266.15	4.4	8.5	11.7	4.4	39.4	88.1
Apache	42.6	34.6	*32.3*	76.4	*61.8*	109.1	0.00	3.17	20.55	3.6	4.0	5.0	3.6	11.5	137.9
Bugzilla	21.8	16.9	*16.2*	*1.9*	3.4	9.1	0.31	1.29	8.32	5.2	6.4	3.1	5.3	42.6	74.8
1.	53.3	42.5	*38.6*	*24.4*	27.9	179.5	0.32	10.55	38.68	3.1	2.9	2.8	3.9	21.6	138.3
2.	40.5	32.9	*31.0*	*14.7*	23.5	25.4	1.16	1.13	1.00	4.1	7.0	3.6	4.0	73.7	86.9
3.	20.8	18.3	*18.0*	*0.2*	2.5	3.4	0.90	1.14	3.33	6.9	3.3	1.1	6.8	32.5	61.5
4.	28.6	21.8	*21.0*	*3.1*	7.3	29.3	0.61	3.45	29.01	4.0	5.1	5.5	4.3	23.4	323.0
5.	63.8	54.8	*47.7*	*134.8*	165.6	655.9	3.43	32.37	69.05	1.9	8.8	3.2	4.2	95.7	133.3
6.	34.0	*24.2*	*24.2*	*7.2*	9.1	18.2	0.19	1.12	456.71	4.5	2.7	1.5	4.6	26.6	54.9
7.	12.5	*9.0*	*9.0*	*0.3*	1.5	2.1	0.33	0.51	N/A	4.9	0.0	0.0	8.7	2.4	4.0
8.	55.6	44.0	*40.5*	45.1	*41.2*	249.9	0.00	13.56	60.32	3.3	3.6	3.3	3.3	23.2	220.4
9.	26.0	30.0	*20.0*	*3.0*	3.4	29.8	N/A	4.47	2.64	5.3	0.0	1.0	5.3	2.7	54.0
10.	60.4	48.2	*44.0*	74.3	*60.8*	357.3	0.00	17.26	70.26	2.6	2.4	2.1	2.6	15.1	89.2
11.	58.3	46.1	*41.9*	*23.8*	27.0	240.7	0.26	13.23	50.65	2.5	2.2	1.9	2.8	20.0	94.5
12.	54.5	43.2	*40.4*	68.0	*53.7*	221.1	0.00	10.86	58.95	3.5	2.5	4.1	3.6	15.3	151.6
13.	48.6	37.7	*36.5*	45.5	*40.1*	60.5	0.00	1.24	17.28	5.4	3.1	2.2	5.4	12.6	94.0

Table 5 (continued)

Name	Size			Run time (s)			Break-even (s/cfg.)			Coefficient of variation in size (%)			Coefficient of variation in run time (%)		
	mAETG best	TSR	ALL	mAETG best	TSR	ALL	TSR vs. mAETG best	ALL vs. mAETG best	ALL vs. TSR	mAETG best	TSR	ALL	mAETG best	TSR	ALL
14.	51.8	39.6	*36.9*	*20.2*	23.9	58.1	0.30	2.54	12.68	4.0	3.1	2.0	4.1	18.4	71.6
15.	40.4	31.8	*30.7*	*4.9*	9.8	19.3	0.57	1.48	8.65	3.4	3.9	2.7	3.3	18.5	76.4
16.	33.4	24.3	*24.1*	*9.7*	11.2	19.7	0.16	1.08	38.81	5.3	3.6	1.0	5.5	21.2	35.1
17.	53.4	42.6	*39.2*	46.9	*45.8*	335.5	0.00	20.32	84.21	3.3	2.9	3.9	3.4	14.7	171.8
18.	57.3	45.9	*42.3*	60.1	*52.8*	303.5	0.00	16.23	70.43	2.1	2.7	3.6	2.2	13.6	108.3
19.	64.7	52.3	*47.8*	168.4	*124.2*	823.6	0.00	38.77	154.72	2.4	4.5	2.4	2.5	47.7	86.1
20.	71.5	58.5	*53.2*	121.1	*86.3*	1,133.3	0.00	55.31	199.05	1.7	4.8	1.7	1.8	63.4	171.4
21.	51.7	38.1	*36.3*	*14.3*	18.7	46.2	0.32	2.07	15.29	3.9	3.4	1.3	3.9	17.1	89.1
22.	*26.2*	36.0	36.0	*6.7*	8.9	12.3	N/A	N/A	86.24	4.6	0.5	0.0	4.5	14.8	39.4
23.	15.7	18.0	*12.6*	0.6	*0.5*	10.1	0.04	3.06	1.78	5.4	0.0	7.5	6.0	2.8	52.2
24.	58.3	47.0	*42.8*	40.6	*39.6*	304.5	0.00	17.03	63.38	2.7	2.6	2.3	2.6	22.8	128.1
25.	65.7	53.1	*48.3*	61.7	*53.1*	507.8	0.00	25.64	93.94	2.0	2.1	1.7	7.7	21.0	104.0
26.	42.0	32.9	*30.7*	*15.5*	22.0	71.2	0.71	4.93	22.18	2.8	5.9	2.5	3.3	26.6	75.8
27.	45.8	36.5	*36.0*	*4.6*	8.2	10.8	0.39	0.63	5.15	4.6	2.0	0.4	4.6	24.5	60.5
28.	68.5	55.4	*50.7*	170.4	*121.6*	1,522.3	0.00	75.95	296.75	2.1	2.9	2.4	2.1	27.4	119.9
29.	38.4	31.6	*29.7*	*38.9*	41.0	89.3	0.31	5.79	25.41	4.1	3.1	3.3	4.1	17.7	117.0
30.	45.8	20.4	*19.3*	*10.3*	17.5	30.5	0.28	0.76	12.23	6.9	4.3	4.9	6.8	113.2	129.0
Sum	1,546.1	1,257.4	*1,155.4*	*1,533.1*	1,910.4	9,501.8	1.31	20.40	74.40	0.6	0.8	0.6	0.9	16.8	37.6

on average 14.0% more time for testing. Similarly, ALL's break-even points against mAETG are defined until its configurations take an average of 15.5% more time. Thus, simulated annealing can save total computation time, even if it chances to pick feature combination that take somewhat longer to test.

4.3.3 Unconstrained Problems (RQ3)

Results RQ3 considers the five real-world benchmark problems without their constraints. The results appear in Table 6 following the same format as in Tables 2 and 5.

The difference in array size is quite small; two rows in 25 runs. Even so, ALL is still consistently faster than CONTROL, by a factor of more than 300. On the same problems with constraints, the difference in size was larger—a 4.60% increase instead of a 0.37% increase—and the difference in speed less striking—only a factor of 35. So without constraints it appears that ALL performs better than the trends from constrained problems would suggest.

Analysis The size differences in Table 6 are so small compared to the standard deviations (e.g., 1.9 rows for CONTROL's total and 1.2 rows for ALL's) that we cannot report break-even points with any reasonable measure of confidence. To decide between CONTROL and ALL on an unconstrained problem with long-running test suites, practitioners would have to understand more about the character of their problem and how it affects either implementation. Such a detailed understanding belongs to future work. If, on the other hand, configurations can be tested quickly, it is unlikely that the differences in array size will counteract ALL's savings in sample generation time.

Follow-up Experimentation Because earlier work has claimed better array sizes for simulated annealing on unconstrained problems (Cohen et al. 2003a, b), and our modifications seemed unhelpful for array size, we returned to those findings.

In Table 7 we have identified 29 unconstrained problems from the literature on covering arrays that have been solved by simulated annealing. The results of one run of the variant ALL are placed beside the best published sizes by any algorithm or direct constructions, and the best sizes by simulated annealing. Note that some best sizes were obtained by simulated annealing and appear in both the "Best" and "Best by SA" columns. Run times are omitted, first because our focus is on array size, and second because for many of these benchmarks run times are not available,

Table 6 Average size and times over 5 runs for unconstrained two-way problems

Name	Size		Run time (s)		Coefficient of variation in size (%)		Coefficient of variation in run time (%)	
	CONTROL	ALL	CONTROL	ALL	CONTROL	ALL	CONTROL	ALL
SPIN-S	17.2	*16.4*	187.81	*4.02*	6.4	3.3	44.4	62.6
SPIN-V	*26.4*	*26.4*	11,934.89	*45.04*	2.1	2.1	152.3	97.9
GCC	*17.0*	17.2	39,878.71	*148.08*	0.0	2.6	2.5	28.3
Apache	*32.0*	32.4	28,132.43	*55.82*	4.4	1.7	10.9	11.9
Bugzilla	*16.0*	16.6	50.44	*8.43*	0.0	3.3	46.5	85.8
Sum	*108.6*	109.0	80,184.28	*261.39*	1.7	1.1	23.0	23.6

Table 7 Sizes for unconstrained t-way problems

t	Model	Best		Best by SA		ALL	Diff.
2	3^4	9	Lei and Tai (1998)	9	Cohen et al. (2003b)	9	0
2	$5^1 3^8 2^2$	15	Cohen et al. (2003b)	15	Cohen et al. (2003b)	15	0
2	3^{13}	15	Lei and Tai (1998)	16	Cohen et al. (2003b)	15	1
2	$4^1 3^{39} 2^{35}$	21	Cohen et al. (2003b)	21	Cohen et al. (2003b)	22	−1
2	$5^1 4^4 3^{11} 2^5$	21	Cohen et al. (2003b)	21	Cohen et al. (2003b)	23	−2
2	$4^{15} 3^{17} 2^{29}$	30	Cohen et al. (2003b)	30	Cohen et al. (2003b)	30	0
2	$6^1 5^1 4^6 3^8 2^3$	30	Cohen et al. (2003b)	30	Cohen et al. (2003b)	30	0
2	$7^1 6^1 5^1 4^5 3^8 2^3$	42	Cohen et al. (2003b)	42	Cohen et al. (2003b)	42	0
2	4^{100}	45	Cohen et al. (2003b)	45	Cohen et al. (2003b)	46	−1
2	6^{16}	62	Cohen et al. (2003b)	62	Cohen et al. (2003b)	64	−2
2	7^{16}	84	Colbourn (2009)	87	Cohen et al. (2003b)	86	1
2	8^{16}	110	Colbourn (2009)	112	Cohen et al. (2003b)	112	0
2	8^{17}	111	Colbourn (2009)	114	Cohen et al. (2003b)	114	0
2	10^{20}	162	Colbourn (2009)	183	Cohen et al. (2003b)	185	−2
3	3^6	33	Chateauneuf and Kreher (2002)	33	Cohen et al. (2003b)	33	0
3	4^6	64	Chateauneuf and Kreher (2002)	64	Cohen et al. (2003b)	96	−32
3	$5^2 4^2 3^2$	100	Cohen et al. (2003b)	100	Cohen et al. (2003b)	100	0
3	5^6	125	Chateauneuf and Kreher (2002)	152	Cohen et al. (2003b)	185	−33
3	5^7	180	Colbourn (2009)	201	Cohen et al. (2003b)	213	−12
3	6^6	258	Colbourn (2009)	300	Cohen et al. (2003b)	318	−18
3	$6^6 4^2 2^2$	272	Cohen et al. (2003a)	317	Cohen et al. (2003a)	383	−66
3	$10^1 6^2 4^3 3^1$	360	Cohen et al. (2003b)	360	Cohen et al. (2003b)	360	0
3	8^8	512	Chateauneuf and Kreher (2002)	918	Cohen et al. (2003a)	942	−24
3	7^7	545	Cohen et al. (2003a)	552	Cohen et al. (2003a)	573	−21
3	9^9	729	Chateauneuf and Kreher (2002)	1,490	Cohen et al. (2003a)	1,422	68
3	10^6	1,100	Colbourn (2009)	1,426	Cohen et al. (2003b)	1,462	−36
3	10^{10}	1,219	Chateauneuf and Kreher (2002)	2,163	Cohen et al. (2003a)	2,175	−12
3	12^{12}	2,190	Chateauneuf and Kreher (2002)	4,422	Cohen et al. (2003a)	4,262	160
3	14^{14}	3,654	Chateauneuf and Kreher (2002)	8,092	Cohen et al. (2003a)	8,103	−11
Sum		12,098		21,377		21,420	−43

nor would they have come from comparable hardware. The last column presents the difference between ALL and the best reported by simulated annealing in the literature. A positive value means that ALL produced a smaller size array, a negative value means we have created a larger array.

The aggregate figures are promising; ALL's total array size is only 0.2% higher than the best sizes formerly listed for simulated annealing. In several cases it produced smaller arrays than reported in the earlier simulated annealing work. However, most of these cases are large problems where search-based techniques show more variance from run to run.

The two-way benchmark problems give much the same impression as the figures in Table 6: no clear difference in array size. On the three-way problems, however, less is apparent. We believe that a sound comparison at $t = 3$ demands further study. For one point, we are comparing minimums obtained with potentially specialized parameter settings to a single run using generic parameters. We looked at one instance, $6^6 4^2 2^2$, and found that parameter variation could account for the difference in array size; it may be that this is true on all of the problems. Furthermore, we have

not measured the variance in array size for the competing algorithms, so we yet lack the data to test for a significant difference. We leave this investigation to future work.

4.3.4 Robustness (RQ4)

Results To evaluate the robustness of our modifications, we considered the factors suggested by Garousi (2008):

1. Are the results repeatable across multiple runs?
2. How quickly does the search reach a stable fitness plateau?
3. Is the search resilient to changes in its inputs?

Together with the results we presented earlier, we use estimated coefficients of variation—the estimated standard deviation divided by the estimated mean—to answer the first and second questions about robustness. We provide these coefficients in Tables 2, 5, and 6. For Table 2 we only discuss coefficients for the totals; standard deviations and coefficients of variation for each cell in the table are available on the evaluation's website. Tables 5 and 6, on the other hand, include coefficients for each benchmark problem.

We partially answer the third robustness question by turning to the follow-up experimentation in RQ1, presented in Table 4, and the results from RQ3 in Tables 6 and 7. Our benchmark problems are designed to follow the characteristics of constrained highly-configurable software (Cohen et al. 2008), and so do not represent the complete diversity of possible inputs. Nonetheless, these tables highlight the influence of two important dimensions. Table 4 shows the effects of larger t on performance, and Table 6 additionally explores problems that lack feature constraints. This latter set in particular does not follow our characterization of highly-configurable software, not only because the problems are unconstrained, but also because the k and v_i values are distributed differently.

Analysis We first consider Table 2, where we note that, in general, our modifications cause the search to counteract poor starting points and poor random decisions with increased run time, shifting the variation in the final array size to variation in the sample generation time. Thus, our array size results with Grouped, TSR, and All are more repeatable than our results with Control. Not only have our alterations made the average sizes and times more suitable for our target scenario, but as differences between runs have less impact when the per-configuration testing time is large, they have also adjusted the variances appropriately.

Table 5 shows that simulated annealing with this shift is as consistent as the greedy algorithm in terms of size: its coefficient of variation is only 0.6% for All. And TSR comes close with a coefficient at 0.8%.

On the other hand, the meta-heuristic search is far more variable with regards to run time: a coefficient of 37.6% when both primary changes are included, as compared to 0.9% for mAETG. Nevertheless, the standard deviation of run time is less than 5 mins on most (23) of the benchmarks with All, and all of the benchmarks with TSR. In the worst case, Benchmark Problem 20 under All, the standard deviation reaches a half hour, so our experiments estimate the cost of a six-sigma event to be at most three hours. Together with the already small means, this evidence suggests that simulated annealing reaches its fitness plateaus quickly, though, according to Table 2, not as consistently as Control.

The coefficients of variation are less telling in Table 6. They are similar for the sums across all five problems, but sometimes quite different on individual benchmark problems. Understanding this discrepancy will require a more extensive investigation.

Finally, we revisit Tables 4, 6, and 7 to understand the impact of differences in the inputs. Table 4 shows that run time grows rapidly with t, which may indicate excessive persistence on these harder problems. Table 6 demonstrates that ALL does just as well as the base algorithm on unconstrained problems, and Table 7 indicates that ALL is competitive with the best published simulated annealing results. So, apart from changes to the strength of testing, simulated annealing with our modifications tolerates a variety of inputs.

In summary, we can conclude that our changes yield a robust search with respect to all three criteria: the search results are as repeatable as the greedy algorithm's, fitness plateaus are typically encountered within 5 mins and almost certainly within three hours, and the search behaves well under a variety of inputs, except when t is large.

4.4 Threats to Validity

All four research questions have threats to their validity. We classify these threats as affecting external, internal, or construct validity.

4.4.1 External Validity

The major external threat is our choice of benchmarks. We cannot guarantee that the real-world CIT problems, the synthetic inputs, or the unconstrained problems accurately represent configurable software. Though diverse, the first two groups are all derived from open-source command-line or web-based software; the last group is artificial.

4.4.2 Internal Validity

There are three main threats to internal validity. First, the experiments for RQ1 only use five runs of the stochastic algorithms because the cost of each run is high: 0.2 machine-years in the case of Table 2. Though more repetitions are unlikely to change the qualitative answer to the research question, they may affect the measured magnitude of the algorithmic differences. Second, the changes forced a recalibration of the simulated annealing parameters, and the full parameter space was not explored, again because of the prohibitive cost. It may be that we biased the results by ignoring some parameters. Third, although we have verified that the results of every run cover every attainable t-set and satisfy constraints, it is impossible to be completely confident that the implementations are correct translations from pseudocode, or that they are fault-free.

4.4.3 Construct Validity

The studies have only compared simulated annealing to one implementation of a greedy algorithm. To determine if the modified versions of simulated annealing are

indeed best, one would have to compare against other leading competitors. We do not have access to the source code of those competitors, so it would be very difficult to control for implementation details that are not associated with algorithmic differences. Because the goal is to compare algorithms, the chosen implementations are those that have produced many of the best results in the literature and whose source code was available to us.

We also restricted our experiments to performance of this algorithm. Other measures such as usability may also be important.

Another threat to construct validity is that performance may not be reflected solely in total computation time. For instance, if configurations can be tested in parallel at little expense then array size may not have as great of an importance as if test suites are run sequentially. Other factors might also matter. Suppose that interaction testing is not expected to finish. In that case, diversity of *t*-sets in the early rows is desirable, whereas otherwise ordering is of no importance.

5 Related Work

Early efforts to build CIT samples focused on unconstrained covering arrays (Cohen et al. 1997, 2003b; Colbourn et al. 2004; Lei et al. 2007; Tai and Lei 2002; Nurmela 2004; Stardom 2001; Stevens 1998). In many of these papers however, constraints are discussed as secondary to the sample constructions or listed as future work.

The challenges of constrained CIT are first discussed in the work of D. Cohen et al. They propose remodeling the feature models to encode constraints (Cohen et al. 1997). Hartman and Raskin as well as Bryce and Colbourn present a discussion and formalize some of the issues that arise in constrained CIT, even though neither provides a general algorithmic solution (Hartman and Raskin 2004; Bryce and Colbourn 2006). Hnich et al. used a SAT solver to construct CIT samples, so their algorithm could support constraints if the adjustments for unattainable *t*-sets are made manually, but they have not implemented this (Hnich et al. 2006).

In 2006 Czerwonka provided constraint support in his greedy algorithm by tracking all of the excluded combinations and then checking for each of them after adding to the array (Czerwonka 2006). He, however, does not say how the checking is implemented. Our own work revisited and formalized constrained CIT and created a framework for incorporating this capability into an AETG-like algorithm and a meta-heuristic search algorithm, through the use of a SAT solver (Cohen et al. 2007b). In follow-on work we modified this approach to share information between the greedy search and the SAT solver search, further increasing performance (Cohen et al. 2007a, 2008).

The recent work of Calvagna and Gargantini (2009) and Grieskamp et al. (2009) incorporates constraint handling directly into the construction of CIT samples through the use of SAT modulo theory (SMT) solvers. The former work selects *t*-sets to be packed into a row greedily, but relies on the solver to check which new *t*-sets can be added. The resulting arrays are competitive, but almost always larger than those produced by mAETG. However, the focus of that work is the investigation of *t*-set prioritization heuristics in the presence of constraints, for which their tool, ATGT, was specially built (Calvagna and Gargantini 2009). The latter work leverages the

greater capabilities of SMT solvers to support non-boolean constraints. It also broadens the notion of a CIT sample by considering which factors interact and at which strengths they interact to offer variable strength testing (Grieskamp et al. 2009). But the focus of their algorithm is on cases where the time to generate samples is of the greatest importance; array size is not the paramount consideration as in the problem that we target.

On the meta-heuristic side, in our 2007 work, we added constraint handling to a simulated annealing-based algorithm (Cohen et al. 2007b). We used a similar approach as we did with the greedy algorithm and integrated an off-the-shelf SAT solver with the search, but saw unsatisfactory performance when scaling to large problems and higher strength t. Having identified the need for a generator that could create small samples subject to constraints, we modified the simulated annealing algorithm from our preliminary work to improve its performance (Garvin et al. 2009). We are unaware of other attempts to incorporate constraint handling into a meta-heuristic search algorithm for CIT.

There has, however, been general work on constraint handling in meta-heuristics; Coello Coello (2002) gives a survey of this challenge. The principal techniques include integration of constraints into the fitness function and switching to a multi-objective search. Another alternative, employed in this work, is to reorganize the search space to mitigate the effects of forbidding constraint violations outright.

Ideas similar to t-set replacement also appear in earlier work to build unconstrained CIT samples. Nurmela for instance used tabu search and moves that replaced t-sets to find CIT samples with $t = 2$ and all v_i set to 3 (Nurmela 2004). Likewise, the algorithms by Czerwonka and Grieskamp et al. both operated from the perspective of t-sets, though in a greedy, not a meta-heuristic, algorithm (Czerwonka 2006; Grieskamp et al. 2009).

As for our work in the broader sense, the area of using meta-heuristic search to solve software engineering problems is commonly called search based software engineering (Harman 2007). In this sub-area of software engineering problems such as test case generation (McMinn 2004), program refactoring (Harman and Tratt 2007), prioritization for regression testing (Li et al. 2007), and module clustering (Harman et al. 2005) are reformulated as optimization problems. Then meta-heuristic search algorithms such as simulated annealing, genetic algorithms, or tabu search are used to optimize the problems. Our work falls under this category of research: we are using a search to optimize configuration sample generation for software testing.

6 Conclusions and Future Work

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. However, the meta-heuristic search simulated annealing did not retain its ability to produce small samples and scaled poorly when solving constrained CIT problems.

Therefore we optimize 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 further refinements these changes provide smaller sample sizes for a fraction of the original cost.

We have run experiments on 67 versions of the constrained simulated annealing algorithm and compared the best versions with a greedy implementation on a set of 35 realistic samples. The experimental results show that, relative to the greedy implementation, simulated annealing needs on average 25% fewer configurations, but its run time is longer. Combining the cost of sample generation and the time needed to test each configuration in a sample provides a comprehensive measure of the total cost of testing using CIT methods. Based on that measure, we calculate a break-even point of less than 21 s for the simulated annealing algorithm relative to the greedy algorithm. Thus, if a test suite takes longer than 21 s, on average, to execute, simulated annealing leads to lower overall testing time. Since for most real-world systems test suite execution takes considerably longer than 21 s, simulated annealing may be preferable in practice to greedy CIT algorithms.

We found that simulated annealing can save time without compromising array size on two-way unconstrained problems. Our preliminary results were inconclusive on higher strength samples, largely owing to the variance between runs of the stochastic algorithm, which highlights an opportunity for future work: we should consider how can this variance might be mitigated. From our experience one-sided narrowing's stopping criterion seems to be the culprit, so we should search for a more predictable way to terminate that search.

Based on our objective to minimize total computation time, the ideal stopping criterion would halt when further compressing the CIT sample would incur more computation time than it would save. Given the per-configuration testing time such a termination condition would adapt the algorithm to the relative importance of speed and array size. The principal challenge is the noisiness in local estimates of the search's rate of progress; we aim to address this challenge in future work.

We also intend to investigate the trade-off between coverage and total computation time to see if it would be more practical to build samples that cover most rather than all of the *t*-sets.

Acknowledgements We would like to thank Jiangfan Shi for the use of his constrained mAETG tool and for supplying the CIT models for evaluation. 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 Air Force Office of Scientific Research through award FA9550-09-1-0129, the Army Research Office through DURIP award W91NF-04-1-0104, the Defense Advanced Research Projects Agency through award HR0011-09-1-0031 and through the National Aeronautics and Space Administration under grant number NNX08AV20A. 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, AFOSR, ARO, DARPA or NASA.

References

Bryce RC, Colbourn CJ (2006) Prioritized interaction testing for pair-wise coverage with seeding and constraints. Inform Software Tech 48(10):960–970

Bryce R, Colbourn CJ (2007) One-test-at-a-time heuristic search for interaction test suites. In: 9th conference on genetic and evolutionary computation, pp 1082–1089

Bryce R, Colbourn CJ, Cohen MB (2005) A framework of greedy methods for constructing interaction tests. In: 27th international conference on software engineering, pp 146–155

Calvagna A, Gargantini A (2009) Combining satisfiability solving and heuristics to constrained combinatorial interaction testing. In: 3rd international conference on tests and proofs. Springer, pp 27–42
Chateauneuf M, Kreher DL (2002) On the state of strength-three covering arrays. J Comb Des 10(4):217–238
Clements P, Northrup L (2002) Software product lines: practices and patterns. Addison-Wesley
Coello Coello C (2002) Theoretical and numerical constraint handling techniques used with evolutionary algorithms: a survey of the state of the art. Comput Methods Appl Mech Eng 191(11–12):1245–1287
Cohen DM, Dalal SR, Fredman ML, Patton GC (1997) The AETG system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng 23(7):437–444
Cohen MB, Colbourn CJ, Ling ACH (2003a) Augmenting simulated annealing to build interaction test suites. In: 14th international symposium on software reliability engineering, pp 394–405
Cohen MB, Gibbons PB, Mugridge WB, Colbourn CJ (2003b) Constructing test suites for interaction testing. In: 25th international conference on software engineering, pp 38–48
Cohen MB, Dwyer MB, Shi J (2007a) Exploiting constraint solving history to construct interaction test suites. In: 2nd testing: academic and industrial conference—practice and research techniques, pp 121–130
Cohen MB, Dwyer MB, Shi J (2007b) Interaction testing of highly-configurable systems in the presence of constraints. In: 5th international symposium on software testing and analysis, pp 129–139
Cohen MB, Dwyer MB, Shi J (2008) Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. IEEE Trans Softw Eng 34(5):633–650
Colbourn C (2009) Covering array tables. Available at http://www.public.asu.edu/~ccolbou/src/tabby/catable.html. Accessed 28 September 2009
Colbourn CJ (2004) Combinatorial aspects of covering arrays. Le Matematiche (Catania) 58:121–167
Colbourn CJ, Cohen MB, Turban RC (2004) A deterministic density algorithm for pairwise interaction coverage. In: 1st IASTED international conference on software engineering, pp 345–352
Czerwonka J (2006) Pairwise testing in real world. In: 10th Pacific northwest software quality conference, pp 419–430
Eén N, Sörrenson N (2007) MiniSAT-C v1.14.1. Available at http://minisat.se/. Accessed 29 June 2010
Fouché S, Cohen MB, Porter A (2009) Incremental covering array failure characterization in large configuration spaces. In: 7th international symposium on software testing and analysis, pp 177–187
Garousi V (2008) Empirical analysis of a genetic algorithm-based stress test technique. In: Proceedings of annual conference on genetic and evolutionary computation, pp 1743–1750
Garvin BJ, Cohen MB, Dwyer MB (2009) An improved meta-heuristic search for constrained interaction testing. In: 1st international symposium on search based software engineering, pp 13–22
Grieskamp W, Qu X, Wei X, Kicillof N, Cohen MB (2009) Interaction coverage meets path coverage by SMT constraint solving. In: Testing of communicating systems and international workshop on formal approaches to testing of software
Harman M (2007) The current state and future of search based software engineering. In: Future of software engineering, pp 342–357
Harman M, Tratt L (2007) Pareto optimal search based refactoring at the design level. In: 9th conference on genetic and evolutionary computation, pp 1106–1113
Harman M, Swift S, Mahdavi K (2005) An empirical study of the robustness of two module clustering fitness functions. In: 7th conference on genetic and evolutionary computation, pp 1029–1036
Hartman A, Raskin L (2004) Problems and algorithms for covering arrays. Discrete Math 284:149–156
Hnich B, Prestwich S, Selensky E, Smith BM (2006) Constraint models for the covering test problem. Constraints 11:199–219
Kuhn DR, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans Softw Eng 30(6):418–421
Lei Y, Tai KC (1998) In-parameter-order: a test generation strategy for pairwise testing. In: 3rd international symposium on high-assurance systems engineering, pp 254–261

Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J (2007) IPOG: a general strategy for t-way software testing. In: 14th international conference on the engineering of computer-based systems, pp 549–556
Li Z, Harman M, Hierons RM (2007) Search algorithms for regression test case prioritization. IEEE Trans Softw Eng 33(4):225–237
Malik S (2004) zChaff. Available at http://www.princeton.edu/~chaff/zchaff.html. Accessed 29 June 2010
McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156
Memon A, Porter A, Yilmaz C, Nagarajan A, Schmidt DC, Natarajan B (2004) Skoll: distributed continuous quality assurance. In: 26th international conference on software engineering
Nurmela KJ (2004) Upper bounds for covering arrays by tabu search. Discrete Appl Math 138(1–2):143–152
Pohl K, Böckle G, van der Linden F (2005) Software product line engineering. Springer, Berlin
Qu X, Cohen MB, Rothermel G (2008) Configuration-aware regression testing: an empirical study of sampling and prioritization. In: 6th international symposium on software testing and analysis, pp 75–85
Stardom J (2001) Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University
Stevens B (1998) Transversal covers and packings. PhD thesis, University of Toronto
Tai KC, Lei Y (2002) A test generation strategy for pairwise testing. IEEE Trans Softw Eng 28(1):109–111
Yilmaz C, Cohen MB, Porter A (2006) Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Trans Softw Eng 31(1):20–34

Brady J. Garvin is a Ph.D. student and research assistant in the Department of Computer Science and Engineering at the University of Nebraska – Lincoln. He received the M.S. in computer science and the B.S. in mathematics and computer science, also from the University of Nebraska – Lincoln. He was awarded the Graduate Assistantship in Areas of National Need from the U.S. Department of Education and the National Science Foundation Graduate Research Fellowship. His research interests include software testing and analysis, especially as they apply to preventing, finding, mitigating, and repairing interaction faults.

Myra B. Cohen is an associate professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln where she is a member of the Laboratory for Empirically based Software Quality Research and Development (ESQuaReD). She received the PhD degree in computer science from the University of Auckland, New Zealand and the MS degree in computer science from the University of Vermont. She received the BS degree from the School of Agriculture and Life Sciences, Cornell University. She is a recipient of a National Science Foundation Faculty Early CAREER Development Award and an Air Force Office of Scientific Research Young Investigator Program Award. Her research interests include testing of configurable software systems and software product lines, combinatorial interaction testing, and search based software engineering. She is a member of the IEEE and ACM.

Matthew B. Dwyer is the Henson Professor of Software Engineering in the Department of Computer Science and Engineering at the University of Nebraska - Lincoln. He received the BS in Electrical Engineering in 1985 from the University of Rochester and then worked for six years as a Senior Engineer with Intermetrics Inc. developing compilers and software for safety-critical embedded systems. He received the MS in Computer Science from the University of Massachusetts at Boston in 1990 and the PhD from the University of Massachusetts at Amherst in 1995. His research interests are in software analysis, verification and testing. Dr. Dwyer has served as program chair for the SPIN Workshop on Model Checking of Software (2001), the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2002), the ACM SIGSOFT Symposium on Foundations of Software Engineering (2004), the ETAPS conference on Fundamental Approaches to Software Engineering (2007) and the International Conference on Software Engineering (2008). He is a member of the IEEE Computer Society and an ACM Distinguished Scientist.