TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation

Jinkun Lin*†, Chuan Luo†, Shaowei Cai‡, Kaile Su§¶, Dan Hao*† and Lu Zhang*†
*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
†School of Electronics Engineering and Computer Science, Peking University, Beijing China
‡Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
§Department of Computer Science, Jinan University, Guangzhou, China
¶Institute for Integrated and Intelligent Systems, Griffith University, Brisbane, Australia
{jkunlin, chuanluosaber, shaoweicai.cs}@gmail.com; k.su@griffith.edu.au; {haod, zhanglu}@sei.pku.edu.cn

***Abstract*—Covering arrays (CAs) are often used as test suites for combinatorial interaction testing to discover interaction faults of real-world systems. Most real-world systems involve constraints, so improving algorithms for covering array generation (CAG) with constraints is beneficial. Two popular methods for constrained CAG are greedy construction and meta-heuristic search. Recently, a meta-heuristic framework called two-mode local search has shown great success in solving classic NP-hard problems. We are interested whether this method is also powerful in solving the constrained CAG problem. This work proposes a two-mode meta-heuristic framework for constrained CAG efficiently and presents a new meta-heuristic algorithm called *TCA*. Experiments show that *TCA* significantly outperforms state-of-the-art solvers on 3-way constrained CAG. Further experiments demonstrate that *TCA* also performs much better than its competitors on 2-way constrained CAG.**

I. Introduction

The requirement on more customizable software is increasing both for developers and for users. Comparing with regarding related programs as independent ones, organizing them as a family in which each program shares the same core set of functionaries is easier to design and implement. In these situations, highly-configurable software is increasingly becoming the trend of software development. By implementing systems as an integrated highly-configurable software, significant reuse can be achieved and thus the cost of development can be greatly reduced.

Nevertheless, the validation of highly-configurable software is more challenging than traditional software of comparable scale and complexity [1]. One main reason is that combination of specific sets of features may introduce faults. It is unrealistic to validate every feature combination as that means testing all possible configurations [2], [3]. Supposing there are 9 possible components, each with five settings, then there will be totally $5^9 = 1,953,125$ configurations to test. Therefore, an advisable technique to sample the configurations is called for.

Combinatorial interaction testing (CIT) has proven to be a very effective technique for discovering interaction faults of highly-configurable software. Covering arrays (CAs), by which any combination of the values of any t parameters is covered at least once, is the most widely used representation of test suites for CIT. Covering Array Generation (CAG) is an important problem of CIT [4], whose task is to generate a covering array as small-sized as possible. Most real-world systems have constraints, and CAG algorithms for such systems should generate CAs satisfying the constraints, as ignoring these constraints would cause inaccurate test planning and wasted effort [5]. For constrained CAG problem, 2-way constrained CAG and 3-way constrained CAG are the most commonly studied.

Constrained CAG (as a generalization of the unconstrained version) is an NP-hard problem [4] and remains very difficult. There are two popular classes of practical algorithms for solving this problem: greedy construction methods [6], [7], [8], [9] and meta-heuristic search methods [1], [10]. Greedy construction methods can handle general cases of CAG very fast, but the size of CAs they construct is usually large and cannot be satisfactory in some real-world situations when the cost of testing a configuration is expensive [1]. Meta-heuristic search methods can deal with general cases of CAG, and are also able to generate smaller CAs compared to greedy construction methods. Therefore, meta-heuristic search methods become more favorable as the cost of testing increases [1].

Meta-heuristic search algorithms have shown great success in solving NP-hard problems. In particular, a meta-heuristic framework called two-mode local search has been recently shown to be very effective for classic NP-hard problems such as Boolean Satisfiability [11], [12], [13] and Maximum Satisfiability [14], [15]. A natural question is whether two-mode local search is also efficient for solving the CAG problem. In this paper, we propose a two-mode meta-heuristic framework for solving constrained CAG, which works between the greedy mode and the random mode. Based on this framework, we develop a new two-mode meta-heuristic algorithm called *TCA*. Our *TCA* algorithm employs the influential tabu search [16] in the greedy mode and adapts the random walk heuristic in the random mode. In the greedy mode it prefers to optimize the object function, while in the random mode the algorithm tends to better explore the search space and diversify the search.

We conduct extensive experiments on a broad range of benchmarks to evaluate our *TCA* algorithm against two state-of-the-art solvers, including a meta-heuristic search solver *CASA* [1] and a greedy solver *ACTS* [8], as well as a recently proposed greedy solver *Cascade* [9]. The experimental results show that our *TCA* algorithm obtains much smaller-sized CA

Kaile Su and Dan Hao are the corresponding authors of this paper.

than all its competitors on 3-way constrained CAG. Moreover, *TCA* also performs much better than all its competitors on 2-way constrained CAG. As our *TCA* algorithm employs the random walk heuristic in the random mode and utilizes tabu search in the greedy mode, we conduct more empirical analysis on the effectiveness of the random walk heuristic and the tabu search. The empirical results indicate the effectiveness of both the random walk heuristic and the tabu search heuristic in the algorithm.

Our main contributions in this paper are summarized as follows.

1) A two-mode meta-heuristic framework for constrained CAG solving, which combines the greedy mode and the random mode in an effective way.
2) Empirical evidence that *TCA* has superiority on 2-way constrained and 3-way constrained CAG, over its competitors *CASA*, *ACTS* and *Cascade*.
3) Empirical evidences on the effectiveness of the random walk heuristic and the tabu search heuristic on 3-way constrained CAG.

II. Preliminaries

In this section, we give some necessary definitions and notations about this paper.

Definition 1 *A System under Test (SUT) is $M = \langle P, C\rangle$, where $P = \{p_1, p_2, \cdots, p_k\}$ is the set of parameters of the system and $C = \{c_1, c_2, \cdots, c_m\}$ is the set of constraints. The number of parameters $|P|$ is denoted by k. The value domain of a parameter $p_i \in P$ is denoted as $D(p_i)$.*

Two basic definitions in the CAG problem are *tuple* and *test case*, which are formally defined as follow.

Definition 2 *Given a SUT $M = \langle P, C\rangle$, a tuple $\tau = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \cdots, (p_{i_t}, v_{i_t})\}$ is an assignment to parameters $p_{i_1}, p_{i_2}, \cdots, p_{i_t}$ in P, such that parameter p_{i_j} takes the value $v_{i_j} \in D(p_{i_j})$. A tuple of size t is called t-tuple.*

Definition 3 *Given a SUT $M = \langle P, C\rangle$, a test case $tc = \{(p_1, v_1), (p_2, v_2), \cdots, (p_k, v_k)\}$ is an assignment to all parameters of P, such that parameter p_i takes the value $v_i \in D(p_i)$ (to facilitate our presentations, we allow unassigned parameters in a test case).*

For a SUT $M = \langle P, C\rangle$, a tuple or test case is *valid* if and only if it satisfies all constraints in C. A tuple $\tau = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \cdots, (p_{i_t}, v_{i_t})\}$ is covered by a test case $tc = \{(p_1, v_1), (p_2, v_2), \cdots, (p_k, v_k)\}$ if and only if $\tau \subseteq tc$, that is, the parameters in τ take the identical value as that in tc.

Definition 4 *Given a SUT $M = \langle P, C\rangle$, a covering array $CA(M, t)$ is a matrix, each row of which is a valid test case, such that any valid t-tuple is covered by at least one of the test case of CA, where t is called the covering strength of CA.*

We extend the concept of CA into partial-CA, which is an important concept in our algorithm, and is defined as follows.

Definition 5 *Given a SUT $M = \langle P, C\rangle$, a partial covering array (partial-CA) of M is a matrix α of which each row is a valid test case. The cost of a partial-CA α, denoted by $cost(\alpha)$, is the number of uncovered tuples.*

Algorithm 1: Two-Mode Framework for Constrained CAG

```
   Input: SUT M⟨P, C⟩, covering strength t
   Output: CA α*
1  α ← Initialization();
2  α* ← α;
3  while The termination criterion is not met do
4      if α is a CA then
5          α* ← α;
6          remove one row from α;
7          continue;
8      if With probability p then
9          α ← Random_Mode();
10     else
11         α ← Greedy_Mode();
```

A tuple is covered by a CA if and only if there is at least one test case of the CA that covers the tuple. According to the definitions, there may exist some valid t-tuples uncovered by a partial-CA, and CA is a special type of partial-CA that covers all valid t-tuples.

Given a SUT $M = \langle P, C\rangle$, the CAG problem is to find a CA as small-sized as possible. When solving this problem, a main procedure of meta-heuristic algorithms is to iteratively modify a partial-CA until it becomes a CA.

III. The *TCA* Algorithm

In this section, we first describe an iterative-improvement two-mode framework for meta-heuristic algorithms. On the basis of this two-mode framework, we design a new meta-heuristic algorithm called *TCA*. We specify each component of our *TCA* algorithm, and describe them in detail.

A. A two-mode iterative-improvement framework

In the context of a variety of combinatorial problems, such as SAT and MAX-SAT, two-mode heuristic search algorithms usually show effectiveness [11], [12], [13], [14], [15]. These heuristic algorithms usually work between two modes, i.e., the greedy (intensification) mode and the random (diversification) mode. In the greedy mode, they prefer to optimize the object function, in order to obtain good-quality solutions. In the random mode, those algorithms tend to better explore the search space and diversify the search.

In this subsection, we outline a framework of two-mode meta-heuristic for solving the CAG problem in Algorithm 1, as described below.

In the beginning, the framework activates the initialization phase, which generates a CA α. After the initialization phase, the algorithm executes the search steps (lines 3–11) until the termination criterion is met. In each step, the framework first checks whether α is a CA or not. If this is the case (α is a CA), the framework first updates α^* and then removes one row from α. Otherwise (α is not a CA), the framework switches between

Algorithm 2: Initialization

Input: SUT $M\langle P, C\rangle$, covering strength t
Output: CA α

```
1  α ← ∅;
2  S ← {all valid t-tuples};
3  while S ≠ ∅ do
4      Generate a test case tc whose parameters are all
       unassigned;
5      τ ← pick a tuple ∈ S randomly;
6      Assign the values of τ to the corresponding
       parameter of tc;
7      S = S − {τ};
8      while tc has unassigned parameters do
9          p ← the unassigned parameter with the smallest
           index in tc;
10         U(p) ← the set of unvisited values of p which
           are compatible with assigned parameters in tc;
11         if U(p) ≠ ∅ then
12             Assign the value v ∈ U(p) to p such that tc
               covers the most tuples in S;
13             Remove these tuples from S;
14             Mark v as visited;
15         else
16             Backtrack to the parameter which was
               assigned last time;
17             Push back the corresponding removed tuples
               into S;
18     Insert tc into α;
```

the greedy mode and the random mode, devoting to modifying α to become a CA: with a probability p, the framework works in the random mode; otherwise (with a probability $1-p$), the framework works in the greedy mode.

B. Constraints handling

As we mentioned in Section I, a great deal of constraints exist in a broad range of real-world applications. Algorithms ignoring these constraints may generate a CA which contains invalid test cases, and this phenomenon may lead to inaccurate test planning and wasted effort [5].

The method to handle constraints in our *TCA* algorithm is straightforward and simple. Whenever our algorithm assigns values to the parameters of a test case, our algorithm invokes a well-known SAT solver *MiniSAT* [17] to determine whether the combinations of values of parameters in the test are compatible or not. By encoding the test case and the input constraints together as a boolean formula, both the input constraints and the implicit constraints can be detected by *MiniSAT*. We would like to note that this approach of using SAT solvers to handle constraints is first proposed by Cohen *et al.* [5], and then adopted in Garvin *et al.* [1]. Particularly, our *TCA* algorithm uses the same version of *MiniSAT* as *CASA* [1] does.

C. Initialization

The initialization function (Algorithm 2) in the *TCA* algorithm constructs an initialized CA to cover all valid t-tuples.

Algorithm 3: Tabu_Search

Input: partial-CA α, set of tuples S
`/* S is the set of uncovered valid t-tuples. */`
Output: partial-CA α

```
1  τ ← pick a tuple ∈ S randomly;
2  if N(α, τ, T) ≠ ∅ then
3      α ← the partial-CA with the smallest cost from
       N(α, τ, T);
4      Update S;
5  else
6      Select a row r in α, such that cost(α) would be
       smallest after assigning the value of τ to the
       corresponding parameters of r;
7      Assign the value of τ to the corresponding
       parameters of r;
```

It is in essence a simple one-test-at-a-time greedy construction which is similar to *AETG* [6], [5]. We describe it as follows.

In the beginning, the algorithm generates all t-tuples (t is the requiring coverage strength), and uses the SAT solver *MiniSAT* to filter out all invalid tuples. We use S to denote the set of uncovered valid t-tuples.

After that, the algorithm executes a loop (lines 3–18) to construct a CA. Each iteration of the loop generates a row, i.e., test case, using a greedy heuristic. To construct a test case, our algorithm first randomly selects a tuple $\tau \in S$, and assigns the value of the tuple to the corresponding parameters of the test case, making the tuple covered by the test case. For the rest parameters that remain unassigned, the algorithm sequentially assigns to each of them a value greedily, which should be compatible with the previously assigned parameters. To this end, we employ a backtracking routine. Specifically, let $U(p)$ denote the set of unvisited compatible values of parameter p. For each unassigned parameter p under consideration, if $U(p)$ is not empty, p is assigned with a value in $U(p)$ such that the most uncovered tuples would be covered by this assignment (lines 11–14), and the value is marked as visited. Otherwise, the algorithm backtracks (lines 15–16).

D. The greedy mode

As it is mentioned before, the scheme of the algorithm is to modify a partial-CA α iteratively until it becomes a CA and then removes one row from α leaving some valid t-tuples uncovered. The algorithm repeats this procedure until the time budget is reached.

As for search strategies in the greedy mode, we employ the tabu search approach which is originally stated in Nurmela [16]. In this subsection, we strengthen the tabu search by combining it with the constraint handling capability.

Before describing the details of the greedy mode, several definitions and notations (tabu cell, neighbors of a partial-CA, and neighbors of a partial-CA with respect to a tuple) are presented below for better understanding of the algorithm.

Definition 6 *For a partial-CA α, its cell at row r and column c is denoted as $\alpha[r, c]$. A cell is called a T-tabu cell if and*

Algorithm 4: Random_Walk

Input: partial-CA α, set of tuples S
/* S is the set of uncovered valid t-tuples. */
Output: partial-CA α
1 $\tau \leftarrow$ pick a tuple $\in S$ randomly;
2 $tc \leftarrow$ pick a row in α randomly;
3 Insert the tuples only covered by tc to S;
4 Wipe the assignments of parameters in tc;
5 Assign the values of τ to the corresponding parameter of tc;
6 **while** tc *has unassigned parameter* p **do**
7 $U(p) \leftarrow$ the set of unvisited values of p which are compatible with assigned parameters in tc;
8 **if** $U(p) \neq \emptyset$ **then**
9 Assign a random element $e \in U(p)$ to p;
10 Mark e as visited;
11 **else**
12 Backtrack to the parameter which was assigned last time;
13 Remove the tuples covered by tc from S;

only if it has been changed during the last T search steps, where T is called tabu tenure.

Definition 7 *The neighborhood of a partial-CA α is $N(\alpha) = \{\beta | \beta$ can be obtained from α by modifying just one cell $\}$.*

Definition 8 *For a partial-CA α, the neighborhood of α with respect to a tuple τ is defined as $N(\alpha, \tau) = \{\beta | \beta \in N(\alpha)$ and τ is covered by β but not by $\alpha\}$, and $N(\alpha, \tau, T) = \{\beta | \beta \in N(\alpha, \tau)$ and β is not obtained from α by modifying a T-tabu cell$\}$.*

The detail of the Tabu_Search function is presented in Algorithm 3. In the beginning, the algorithm randomly selects an uncovered valid t-tuple, τ. If $N(\alpha, \tau, T) \neq \emptyset$, then the partial-CA α is transformed to one of its neighbors with the minimum cost in $N(\alpha, T, \tau)$ by modifying one cell (not a T-tabu cell) of α (ties are broken randomly). Otherwise, the algorithm allows changing multiple cells of a row to cover τ, provided that the change does not violate the constraints. Specifically, the algorithm selects one row of α and modifies it to cover τ, ignoring the tabu criterion. In the Tabu_Search function, the algorithm picks the row whose modification would bring the most decrease on the cost of α.

E. The random mode

As we can see from the preceding subsection, in the greedy mode, the algorithm prefers to modify the partial-CA α in a greedy way w.r.t. the cost of α. There is a risk that the search be trapped in a small part of the search space, and thus is likely to miss partial-CAs of smaller sizes. In order to better explore the search space, an advisable solution is to integrate a random mode into the algorithm. The pseudo-code of the random mode utilized in our algorithm is presented in Algorithm 4.

In the random mode, the algorithm first selects an uncovered tuple randomly. Then one row of α is chosen and all the assignment of the parameters of the row is wiped. Next, the algorithm assigns the value of the tuple to the chosen row, making it cover the tuple. For the rest parameters of this row, the algorithm assigns to them a random value in their value domains respectively (lines 6–12 in Algorithm 4). This is accomplished by a backtracking routine, which is similar to that in the Initialization function.

As in the Initialization function, it is possible that there is no compatible value for the parameter assigning. In this case, the algorithm needs to backtrack and choose another value for the parameter lastly assigned.

Algorithm 5: The *TCA* Algorithm

Input: SUT $M\langle P, C\rangle$, covering strength t
Output: CA α^*
1 $\alpha \leftarrow$ Initialization($M\langle P, C\rangle$, t);
2 $S \leftarrow \emptyset$;
/* S is the set of uncovered valid t-tuples. */
3 **while** *The cutoff time is not reached* **do**
4 **if** $S = \emptyset$ **then**
5 $\alpha^* \leftarrow \alpha$;
6 Remove a row from α, such that the number of uncovered tuples is smallest;
7 Add these tuples to S;
8 **continue**;
9 **if** *With probability* p **then**
10 $\alpha \leftarrow$ Random_Walk(α, S);
11 **else**
12 $\alpha \leftarrow$ Tabu_Search(α, S);

F. The description of TCA

In this subsection, we summarise all the components of the algorithm described above, and based on them we propose the *TCA* algorithm. We outline the pseudo-code of the *TCA* algorithm in Algorithm 5 and describe it as follows.

In the beginning, *TCA* calls the Initialization function to construct a CA, which works in a row-by-row fashion and employs greedy strategies.

After that, *TCA* executes the search steps until the time budget is reached. During the search process, *TCA* switches between two mode, that is, the greedy mode and the random mode. With a probability p, *TCA* works in the random mode; otherwise (with a probability $1-p$), *TCA* works in the greedy mode.

In the random mode, *TCA* selects an uncovered valid t-tuple τ and a row of the partial-CA α randomly. Then *TCA* wipes the values of the row and assigns the row according to tuple τ. For each of the unassigned parameter in the row, *TCA* assigns the parameter with a random value from its value domain, as long as the constraints are not violated; if there is not any valid value in the value domain of the parameter, the algorithm backtracks to the last assigned parameter.

In the greedy mode, *TCA* employs a tabu search based heuristic. The algorithm selects an uncovered valid t-tuple τ and then modifies α to its neighbor $\in N(\alpha, \tau, T)$ with lowest cost, breaking ties randomly. If $N(\alpha, \tau, T)$ is empty, *TCA* picks a row and modifies it to cover tuple τ with a greedy strategy (which brings the most decrease on the cost of α).

IV. EXPERIMENTS

In this section, we conduct extensive experiments on a broad range of benchmarks to evaluate the effectiveness of our *TCA* algorithm. We first present the research questions in our experiments. Then we introduce the benchmarks, the state-of-the-art competitors and experimental preliminaries about our experiments. Finally, we present the experimental results and give some discussions about the empirical results.

A. Research Questions

The primary target of this paper is to push forward the state-of-the-art performance on 3-way constrained CAG solving. Therefore, the first research question in our experiments is:

RQ1: How does our *TCA* algorithm compare against the state-of-the-art solvers on 3-way constrained CAG?

Our *TCA* algorithm switches between two modes (i.e., the random mode and the greedy mode). In the random mode, *TCA* employs the random walk heuristic, while in the greedy mode, it utilizes the tabu search heuristic. Thus, two natural questions are to study the effectiveness of the random walk heuristic and the tabu search heuristic in the algorithm.

RQ2: How does our *TCA* algorithm compare with the alternative version which works without the random walk heuristic?

RQ3: How does our *TCA* algorithm compare with the alternative version which works without tabu search?

Although our *TCA* algorithm is developed with the aim of solving 3-way constrained CAG, it can also be used to solve t-way constrained CAG for any t. Thus, it is also interesting to evaluate *TCA* on 2-way constrained CAG, which is the other most commonly studied constrained CAG problem.

RQ4: How does our *TCA* algorithm compare against its state-of-the-art solvers on the 2-way constrained CAG problem?

B. Benchmarks

The benchmarks we use are the same as those used in Garvin *et al.* [1] for evaluating *mAETG* and *CASA*. All the benchmarks are available online[1].

There are 35 benchmarks, five of which are extracted from the nontrivial real-world highly-configurable systems, including:

- SPINV, the verifier-form of SPIN, which is a widely used publicly available model checking tool.
- SPINS, the simulator-form of SPIN.
- GCC, the GNU Compiler Collection including front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj, *etc.*).
- Apache Http Server, a well-known open-source HTTP server for modern operating systems including UNIX and Windows NT.
- Bugzilla, a "Defect Tracking System" or "Bug-Tracking System" to keep track of outstanding bugs in developers' product effectively.

The rest 30 benchmarks were generated synthetically from the characteristics of the abundance, type, and complexity of constraints found in the five real-world systems described above.For the more details about the benchmarks, the readers could refer to Cohen *et al.* [18]. We also note that in our experiments, we put more focus on 3-way benchmarks, as meta-heuristic solvers outperform other kinds of solvers on 2-way benchmarks but their performance on 3-way is still not satisfactory [8].

C. The State-of-the-art Competitors

We compare our *TCA* algorithm against three state-of-the-art constrained CAG solvers, namely *CASA* [1], *ACTS* [8] and *Cascade* [9]. *CASA* is a meta-heuristic algorithm while *ACTS* and *Cascade* are greedy construction algorithms.

CASA [1] is a state-of-the-art constrained CAG algorithm evolved from a simulate-annealing (SA) algorithm [19]. Since its introduction [19], there have been quite a few improvements on the original SA algorithm, and *CASA* stands out as the best-performing algorithm for constrained CAG in terms of the size of CA.

ACTS [8] is improved from an efficient greedy construction algorithm named *IPOG* [20]. It introduces three techniques to improve the performance of constraint handling, and the experimental results in the literature [8] show that the runtime has been improved by one or two orders of magnitude. Moreover, *ACTS* is able to perform significantly better for systems with more complex constraints. It is worth noting that the time consuming cost by *ACTS* for generating CA for constrained CAG is quite small compared to other algorithms.

Cascade [9] is a recently proposed constrained CAG solver and is a one-test-at-a-time solver which uses pseudo-Boolean optimization to generate each new test case. It utilizes a self-adaptive stopping mechanism to stop the optimization process when the solution is good enough and finding better solutions becomes hard. The authors of *Cascade* conducted extensive experiments in the literature [9], and they claimed that *Cascade* works fine on existing benchmarks and outperforms existing approaches when handling massive constraints.

D. Experimental Settings

Our *TCA* algorithm is implemented in C++, and compiled by g++ with the option '-O2'. There is a parameter p in our *TCA* algorithm. In our experiments, p is set to 0.001 (in Algorithm 5) and the tabu tenure T is set to 4 (in Algorithm 3) according to our preliminary experiments. The source code of *TCA* is available online[2].

CASA[3] is also implemented in C++, but is compiled by g++ without any optimization options according to the makefile in the source code online. In order to make the empirical comparisons fair, we modify the makefile of *CASA* to make it

[1] http://cse.unl.edu/~citportal/public/tools/casa/benchmarks.zip

[2] https://github.com/leiatpku/TCA

[3] http://cse.unl.edu/~citportal/

TABLE I. EXPERIMENTAL RESULTS OF *TCA*, *CASA* AND *ACTS* FOR 3-WAY CONSTRAINED CAG ON ALL TESTING BENCHMARKS. WE DO NOT REPORT THE EXPERIMENTAL RESULTS OF *Cascade* BECAUSE IT BECOMES VERY SLOW ON OUR MACHINE DUE TO ITS MEMORY PROBLEM.

Benchmark	*TCA*			*CASA*			*ACTS*	
	avg (min)	avg_time	#suc.	avg (min)	avg_time	#suc.	size	time
benchmark_apache	**155.6** (**153**)	923.8	10	242.9 (240)	921.3	10	173	9.7
benchmark_bugzilla	**48.0** (**48**)	9.8	10	64.6 (61)	26.7	10	68	0.4
benchmark_gcc	**83.0** (**81**)	883.1	10	127.3 (121)	898.4	10	108	11.1
benchmark_spins	**80.0** (**80**)	5.6	10	100.5 (94)	4.9	10	98	0.3
benchmark_spinv	**202.9** (**201**)	245.8	10	231.6 (224)	645.5	10	286	1.2
benchmark_1	**255.2** (**253**)	856.7	10	359.7 (352)	952.8	10	293	1.8
benchmark_2	**143.3** (**141**)	499.4	10	177.3 (166)	795.7	10	174	1.3
benchmark_3	**50.9** (**50**)	99.6	10	61.1 (59)	2.4	10	71	0.4
benchmark_4	**80.0** (**80**)	78.3	10	103.6 (96)	68.7	10	102	0.7
benchmark_5	408.6 (393)	996.0	10	1030.7 (784)	950.7	9	**386**	13.4
benchmark_6	**99.8** (**98**)	244.8	10	122.2 (118)	564.9	10	119	1.0
benchmark_7	**27.0** (**27**)	292.9	10	27.8 (**27**)	3.6	10	35	0.5
benchmark_8	**268.6** (**265**)	847.5	10	394.6 (386)	924.3	10	326	3.4
benchmark_9	**60.0** (**60**)	20.8	10	76.4 (70)	478.1	10	84	0.9
benchmark_10	**321.7** (**311**)	991.4	10	775.2 (599)	966.7	10	329	7.1
benchmark_11	**285.5** (**282**)	878.1	10	401.5 (395)	920.6	10	318	1.9
benchmark_12	**237.1** (**230**)	960.1	10	370.6 (356)	936.8	10	263	5.4
benchmark_13	**181.9** (**180**)	791.3	10	277.6 (271)	948.2	10	200	3.4
benchmark_14	**216.0** (**216**)	155.1	10	261.1 (255)	953.0	10	244	1.3
benchmark_15	**150.0** (**150**)	149.8	10	168.8 (164)	468.8	10	173	0.9
benchmark_16	**96.0** (**96**)	224.0	10	122.6 (119)	652.6	10	117	1.0
benchmark_17	**229.5** (**225**)	896.0	10	343.6 (337)	940.6	10	265	4.3
benchmark_18	**303.6** (**295**)	940.3	10	446.3 (437)	895.2	10	344	5.4
benchmark_19	491.5 (478)	990.2	10	N/A (N/A)	N/A	0	**373**	22.4
benchmark_20	508.3 (487)	982.5	10	1039.0 (1011)	906.9	10	**463**	11.8
benchmark_21	**216.0** (**216**)	78.3	10	241.6 (235)	973.8	10	235	1.4
benchmark_22	**144.0** (**144**)	45.9	10	170.7 (162)	410.6	10	164	1.0
benchmark_23	**36.0** (**36**)	4.2	10	39.2 (37)	2.0	10	48	0.4
benchmark_24	**301.8** (**298**)	890.8	10	449.4 (438)	948.5	10	341	3.1
benchmark_25	**394.5** (**380**)	984.5	10	573.7 (558)	963.7	10	404	5.1
benchmark_26	**169.6** (**167**)	697.4	10	214.2 (206)	896.3	10	207	1.5
benchmark_27	**180.0** (**180**)	51.5	10	201.2 (193)	528.1	10	204	0.9
benchmark_28	504.0 (494)	985.0	10	1053.0 (1037)	940.4	7	**420**	22.1
benchmark_29	**125.2** (**125**)	723.7	10	185.2 (176)	900.7	10	154	4.2
benchmark_30	**73.3** (**73**)	375.1	10	88.8 (82)	364.9	10	100	1.1
#cnt.	31			0			4	

compiled with the same optimization level as our *TCA* algorithm does, i.e., the '-O2' option. For *ACTS*[4] and *Cascade*[5], we use the default settings for them.

All experiments are carried out on a machine with Intel Core i7 2.7 GHz CPU and 15.6 GB memory under Linux. To evaluate the performance of *TCA*, we compare it against the three state-of-the-art solvers on the benchmarks described in Section IV-B for generating 2-way and 3-way CAs.

[4]http://barbie.uta.edu/~fduan/ACTS/release_v2.92/

[5]The source code is kindly provided by its authors.

For both 2-way constrained CAG and 3-way constrained CAG, we run *ACTS* one time and other solvers 10 times on each of the 35 benchmarks. The reason why we run *ACTS* only one time is that *ACTS* is a deterministic algorithm [8], i.e., its empirical behaviors are always identical on the same benchmark. For each run of each solver, the cutoff time is set to 1000 CPU seconds, which is utilized by testing heuristics for classic NP-hard problems, such as Boolean satisfiability, maximum satisfiability as well as minimum vertex cover.

For each solver on each benchmark, we report the smallest size ('min') and averaged size ('avg') of CA found by the

TABLE II. EXPERIMENTAL RESULTS OF *TCA* AND ITS TWO ALTERNATIVE VERSIONS FOR 3-WAY CONSTRAINED CAG ON ALL TESTING BENCHMARKS.

Benchmark	*TCA*			*TCA_alt1*			*TCA_alt2*		
	avg (min)	avg_time	#suc.	avg (min)	avg_time	#suc.	avg (min)	avg_time	#suc.
benchmark_apache	**155.6** (153)	923.8	10	156.3 (**152**)	962.0	10	185.9 (182)	824.9	10
benchmark_bugzilla	**48.0** (**48**)	**9.8**	10	**48.0** (**48**)	10.9	10	**48.0** (**48**)	410.0	10
benchmark_gcc	**83.0** (81)	883.1	10	83.5 (**80**)	875.9	10	99.9 (95)	769.0	10
benchmark_spins	**80.0** (**80**)	**5.6**	10	90.2 (**80**)	5.5	10	**80.0** (**80**)	189.7	10
benchmark_spinv	**202.9** (201)	245.8	10	232.7 (**195**)	482.7	10	221.2 (218)	786.1	10
benchmark_1	255.2 (253)	856.7	10	**251.8** (**250**)	926.6	10	281.0 (279)	890.2	10
benchmark_2	**143.3** (141)	499.4	10	144.9 (**135**)	593.0	10	165.9 (164)	747.2	10
benchmark_3	**50.9** (**50**)	99.6	10	51.0 (51)	154.7	10	51.7 (51)	320.7	10
benchmark_4	**80.0** (**80**)	78.3	10	**80.0** (**80**)	**40.9**	10	88.8 (87)	649.6	10
benchmark_5	**408.6** (**393**)	996.0	10	420.7 (**393**)	991.0	10	422.3 (400)	989.0	10
benchmark_6	99.8 (98)	244.8	10	**96.8** (**96**)	453.0	10	118.1 (117)	564.2	10
benchmark_7	**27.0** (**27**)	292.9	10	27.3 (**27**)	329.2	10	**27.0** (**27**)	**170.0**	10
benchmark_8	268.6 (265)	847.5	10	**267.0** (**264**)	920.2	10	303.3 (301)	904.0	10
benchmark_9	**60.0** (**60**)	**20.8**	10	**60.0** (**60**)	22.4	10	60.7 (**60**)	609.6	10
benchmark_10	**321.7** (311)	991.4	10	332.7 (**310**)	992.1	10	361.9 (352)	953.6	10
benchmark_11	285.5 (282)	878.1	10	**281.6** (**278**)	928.3	10	321.1 (316)	906.6	10
benchmark_12	**237.1** (**230**)	960.1	10	238.7 (**230**)	974.2	10	287.6 (277)	926.5	10
benchmark_13	**181.9** (**180**)	791.3	10	184.0 (**180**)	793.8	10	218.1 (211)	896.6	10
benchmark_14	**216.0** (**216**)	155.1	10	**216.0** (**216**)	**154.1**	10	236.5 (230)	837.5	10
benchmark_15	**150.0** (**150**)	149.8	10	151.6 (**150**)	55.1	10	159.7 (157)	681.6	10
benchmark_16	**96.0** (**96**)	224.0	10	**96.0** (**96**)	**161.7**	10	109.1 (106)	714.7	10
benchmark_17	229.5 (225)	896.0	10	**229.4** (**223**)	960.0	10	274.8 (269)	888.5	10
benchmark_18	**303.6** (**295**)	940.3	10	304.1 (**295**)	980.0	10	337.3 (329)	877.3	10
benchmark_19	**491.5** (**478**)	990.2	10	496.0 (**478**)	983.4	10	496.4 (479)	979.4	10
benchmark_20	**508.3** (487)	982.5	10	516.0 (**480**)	990.5	10	517.9 (493)	993.3	10
benchmark_21	**216.0** (**216**)	**78.3**	10	**216.0** (**216**)	84.2	10	218.1 (**216**)	916.1	10
benchmark_22	**144.0** (**144**)	**45.9**	10	**144.0** (**144**)	48.5	10	149.7 (145)	812.4	10
benchmark_23	**36.0** (**36**)	**4.2**	10	**36.0** (**36**)	31.8	10	**36.0** (**36**)	288.3	10
benchmark_24	**301.8** (298)	890.8	10	302.2 (**297**)	948.7	10	335.9 (327)	885.6	10
benchmark_25	**394.5** (380)	984.5	10	404.2 (**378**)	984.9	10	424.6 (411)	952.2	10
benchmark_26	169.6 (167)	697.4	10	**166.7** (**165**)	817.1	10	190.8 (188)	739.7	10
benchmark_27	**180.0** (**180**)	**51.5**	10	**180.0** (**180**)	53.4	10	183.3 (182)	757.8	10
benchmark_28	**504.0** (**494**)	985.0	10	511.5 (495)	984.9	10	510.7 (496)	985.7	10
benchmark_29	**125.2** (**125**)	723.7	10	128.8 (**125**)	746.7	10	164.3 (155)	813.4	10
benchmark_30	73.3 (73)	375.1	10	**70.1** (**69**)	426.9	10	83.2 (82)	759.8	10
#cnt.	28			16			4		

solver over the 10 runs. In addition, for each solver on each benchmark, we report the averaged time ('avg_time') used for finding the resulted CAs. If a solver fails to find a CA within the cutoff time, we regard this run as failed and ignore this run when calculating the 'avg_time' and 'avg_size'. We also report the number of successful runs (out of 10 runs) for each solver on each benchmark as '#suc.'. If a solver fails in all 10 runs, we mark 'min', 'avg' and 'avg_time' as 'N/A'. For each solver, we report the number of benchmarks (out of 35 benchmarks) as '#cnt.' where it obtains the smallest average size. For each benchmark, the results in the **bold** font indicate the best performance.

E. Experimental Results

In this subsection, we present experimental results and conduct some analyses to answer the research questions mentioned in Section IV-A.

Experiments on comparing *TCA* against its state-of-the-art competitors for 3-way constrained CAG (RQ1): Table I presents the experimental results of our *TCA* algorithm and its state-of-the-art competitors (*CASA* and *ACTS*) for 3-way constrained CAG on all testing benchmarks. We do not report the experimental results of *Cascade* because it becomes very slow on our machine due to its memory problem. As

TABLE III. EXPERIMENTAL RESULTS OF *TCA*, *CASA*, *ACTS* AND *Cascade* FOR 2-WAY CONSTRAINED CAG ON ALL TESTING BENCHMARKS.

Benchmark	*TCA*			*CASA*			*Cascade*			*ACTS*	
	avg (min)	avg_time	#suc.	avg (min)	avg_time	#suc.	avg (min)	avg_time	#suc.	size	time
benchmark_apache	**30.0** (**30**)	3.9	10	34.6 (32)	3.4	10	N/A (N/A)	N/A	0	33	0.4
benchmark_bugzilla	**16.0** (**16**)	0.1	10	16.4 (**16**)	0.1	10	20.2 (20)	13.9	10	19	0.3
benchmark_gcc	**16.4** (**16**)	247.9	10	22.1 (19)	62.8	10	N/A (N/A)	N/A	0	23	0.6
benchmark_spins	**19.0** (**19**)	2.8	10	19.8 (**19**)	0.1	10	27.0 (27)	2.5	10	26	0.3
benchmark_spinv	**32.0** (**31**)	353.6	10	40.2 (36)	2.5	10	41.4 (41)	49.2	10	45	0.6
benchmark_1	**36.0** (**36**)	136.0	10	40.1 (38)	9.4	10	N/A (N/A)	N/A	0	48	0.6
benchmark_2	**30.0** (**30**)	2.9	10	31.8 (**30**)	1.5	10	N/A (N/A)	N/A	0	32	0.5
benchmark_3	**18.0** (**18**)	0.0	10	18.6 (**18**)	0.0	10	20.3 (19)	3.7	10	19	0.3
benchmark_4	**20.0** (**20**)	0.3	10	21.9 (**20**)	0.4	10	26.2 (24)	20.7	10	22	0.4
benchmark_5	**43.3** (**43**)	235.9	10	50.1 (45)	49.4	10	N/A (N/A)	N/A	0	54	0.7
benchmark_6	**24.0** (**24**)	0.3	10	24.2 (**24**)	0.6	10	32.4 (30)	63.3	10	25	0.6
benchmark_7	**9.0** (**9**)	**0.0**	10	**9.0** (**9**)	**0.0**	10	12.0 (12)	3.9	10	12	0.3
benchmark_8	**37.4** (**37**)	227.6	10	41.5 (38)	15.8	10	N/A (N/A)	N/A	0	47	0.6
benchmark_9	**20.0** (**20**)	0.1	10	20.2 (**20**)	0.2	10	23.6 (23)	28.2	10	22	0.6
benchmark_10	**40.0** (**40**)	164.8	10	44.2 (42)	21.8	10	N/A (N/A)	N/A	0	47	0.9
benchmark_11	**39.1** (**39**)	172.9	10	43.3 (41)	12.6	10	N/A (N/A)	N/A	0	47	0.6
benchmark_12	**36.0** (**36**)	12.7	10	41.7 (39)	13.3	10	N/A (N/A)	N/A	0	43	0.5
benchmark_13	**36.0** (**36**)	2.6	10	37.6 (**36**)	3.7	10	N/A (N/A)	N/A	0	40	0.5
benchmark_14	**36.0** (**36**)	0.9	10	38.2 (37)	3.0	10	N/A (N/A)	N/A	0	39	0.4
benchmark_15	**30.0** (**30**)	1.8	10	31.9 (**30**)	0.4	10	39.6 (37)	35.6	10	32	0.7
benchmark_16	**24.0** (**24**)	0.7	10	24.8 (**24**)	0.6	10	N/A (N/A)	N/A	0	25	0.6
benchmark_17	**36.0** (**36**)	32.2	10	40.5 (38)	8.0	10	N/A (N/A)	N/A	0	41	0.6
benchmark_18	**39.9** (**39**)	209.4	10	42.4 (41)	19.2	10	N/A (N/A)	N/A	0	52	0.5
benchmark_19	**44.0** (**43**)	192.9	10	49.4 (47)	28.7	10	N/A (N/A)	N/A	0	51	1.1
benchmark_20	**49.8** (**49**)	216.3	10	53.4 (52)	82.5	10	N/A (N/A)	N/A	0	60	0.8
benchmark_21	**36.0** (**36**)	1.1	10	36.6 (**36**)	2.0	10	N/A (N/A)	N/A	0	39	0.6
benchmark_22	**36.0** (**36**)	**0.5**	10	**36.0** (**36**)	**0.5**	10	N/A (N/A)	N/A	0	37	0.4
benchmark_23	**12.0** (**12**)	0.0	10	12.7 (**12**)	0.0	10	14.8 (14)	3.1	10	14	0.3
benchmark_24	**40.3** (**40**)	185.8	10	43.1 (42)	22.9	10	N/A (N/A)	N/A	0	48	0.6
benchmark_25	**45.7** (**45**)	98.4	10	48.0 (47)	80.9	10	N/A (N/A)	N/A	0	52	0.6
benchmark_26	**27.1** (**27**)	263.4	10	32.9 (30)	2.1	10	N/A (N/A)	N/A	0	34	0.7
benchmark_27	**36.0** (**36**)	0.5	10	36.6 (**36**)	0.4	10	45.8 (45)	32.0	10	37	0.5
benchmark_28	**47.0** (**47**)	387.1	10	51.4 (50)	54.1	10	N/A (N/A)	N/A	0	57	0.9
benchmark_29	**25.0** (**25**)	19.3	10	30.7 (29)	2.7	10	N/A (N/A)	N/A	0	29	0.5
benchmark_30	**16.0** (**16**)	176.5	10	19.7 (19)	0.8	10	N/A (N/A)	N/A	0	22	0.6
#cnt.	35			2			0			0	

can be clearly seen from Table I, our *TCA* algorithm stands out as the best solver on generating small-sized CA. We first focus on the comparison between *TCA* and the state-of-the-art meta-heuristic algorithm *CASA*: on all 35 benchmarks, the averaged size of CAs generated by our *TCA* algorithm is much smaller than that by *CASA*, indicating that the performance of *TCA* exceeds the current state-of-the-art performance of meta-heuristic algorithms for 3-way constrained CAG. When it comes to the comparison between *TCA* and the state-of-the-art greedy construction algorithm *ACTS*, on all 35 benchmarks, our *TCA* algorithm is able to generate smaller-sized CA on 31 benchmarks. It is not surprising that the run time of *TCA* is longer than that of *ACTS*, because it is well acknowledged that greedy construction algorithms are much faster than meta-heuristic algorithms [1], [21]. We consider that our *TCA* algorithm could cooperate well with *ACTS* by employing *ACTS* as the initialization function, which may yield further improvements. Particularly, on those five real-world benchmarks (benchmark_apache, benchmark_bugzilla, benchmark_gcc, benchmark_spins and benchmark_spinv), our *TCA* algorithm generates much smaller-sized 3-way CA compared to *CASA* and *ACTS*, which indicates that *TCA* might be beneficial in practice. We also observe that, even we filter out the redundant rows of CAs generated by *CASA*, the results

of *CASA* can be enhanced on only 4 benchmarks, and those results are still much worse than that of *TCA*.

Empirical analyses of the effectiveness of the random walk and tabu search in *TCA* (RQ2, RQ3): To illustrate the effectiveness of the random walk heuristic and the tabu search heuristic in our *TCA* algorithm, we modify *TCA* to work without the random walk heuristic (i.e., by setting the probability p to 0), resulting in an alternative version named *TCA_alt1*, and to work without tabu search, (i.e., by setting the tabu tenure T to 0), resulting in an alternative version called *TCA_alt2*. We conduct experiments to compare *TCA* with *TCA_alt1* and *TCA_alt2* for 3-way constrained CAG on all testing benchmarks, and the related experimental results are summarized in Table II. In regard of the comparison between *TCA* and *TCA_alt1*, *TCA* achieves the best performance on 28 benchmarks, while the number is only 16 for *TCA_alt1*, indicating that the random walk heuristic is able to improve the robustness of the algorithm. When it comes to the comparison between *TCA* and *TCA_alt2*, *TCA* is able to generate smaller-sized or equal-sized 3-way CAs compared to *TCA_alt2*, which demonstrates the effectiveness of the tabu search heuristic. These indicate that the random walk heuristic and the tabu search heuristic are able to improve the robustness of the algorithm.

Experiments on comparing *TCA* with its state-of-the-art competitors for 2-way constrained CAG (RQ4): Because our *TCA* algorithm exceeds the current state-of-the-art performance on 3-way constrained CAG solving, it is very interesting to investigate the performance of *TCA* for 2-way constrained CAG solving. For this purpose, we conduct additional experiments to evaluate our *TCA* algorithm with its state-of-the-art competitors (i.e., *CASA*, *Cascade* and *ACTS*) for 2-way constrained CAG on all testing benchmarks. The experimental results are reported in Table III. As can be seen from Table III, it is clear that our *TCA* algorithm stands out as the best solver in this comparison. It achieves the best performance in terms of both average size and minimum size of generated CAs on all the 35 benchmarks, pushing the state of the art in solving 2-way constrained CAG. Also, on those five real-world benchmarks, our *TCA* algorithm decreases the average size on all of them, which demonstrates that *TCA* might be beneficial in practice.

Summary on experiments: In this section, we conduct extensive experiments to evaluate our *TCA* algorithm against a state-of-the-art meta-heuristic algorithm *CASA* and a state-of-the-art greedy construction algorithm *ACTS* as well as a recently proposed greedy construction algorithm *Cascade* for 3-way constrained CAG and 2-way constrained CAG on a broad range of testing benchmarks. The experimental results indicate that our *TCA* algorithm exceeds the current state-of-the-art performance on both 3-way constrained CAG solving and 2-way constrained CAG solving. Also, our *TCA* algorithm is able to decrease the average size of generated CA on real-world benchmarks compared to its competitors, which confirms that our *TCA* algorithm might be beneficial in practice. Note that we also tested *TCA* on 4-way CAG benchmarks and found that such benchmarks remain challenging for *TCA*. Nevertheless, since our solver does not exploit any structure of 2-way and 3-way benchmarks, it is possible to improve it for higher way benchmarks.

V. Related Work

Combinatorial interaction testing (CIT) has been extensively studied for the last 20 years and is one of the most important research domains of software engineering. Many classic works on CIT can be found in the survey by Nie and Leung [4]. Many theoretical and practical aspects of CIT are covered by Kuhn *et al.* [22]. The book by Zhang *et al.* [21] reviews the state-of-the-art CAG methods for combinatorial testing.

There are plenty of methods for solving CAG (including unconstrained and constrained versions). Each of them can be roughly classified into one of the four main groups: greedy algorithms, heuristic search algorithms, mathematic method, and using constraint solvers.

Greedy algorithms are the most widely used method for solving CAG. There are two main classes of greedy algorithms, that is, one-test-at-a-time algorithm and in-parameter-order (IPO) algorithm. The former one generates test cases one-by-one until all the valid tuples are covered. The principle for generating each new test case is usually to cover as many as possible uncovered valid t-tuples. The well-known algorithm *AETG* (*Automatic Efficient Test Generator*) is the first that uses the one-test-at-a-time strategy [6]. A generic framework of the *AETG*-like algorithm was introduce in Bryce *et al.* [23]. A number of variations of *AETG* were proposed later on [24], [25], [26], [27], [28], [9]. It is worth noting that the initialization phase of our algorithm *TCA* is actually a simple one-test-at-a-time procedure. On the other hand, the *IPO* algorithm begins by generating a small CA which involves the first t parameters only, and then extends the CA by considering one more parameter in each iteration. The extension process consists of two stages which extends the small CA horizontally and vertically, respectively. The *IPO* strategy was first propose by Lei and Tai [7] for solving 2-way CAG, and was later generalized as an in-parameter-order-general (*IPOG*) framework [20] for solving t-way CAG. There are also a number of variations of the *IPO* algorithm [29], [30], [31], [32], [33], [34].

Heuristic search algorithms usually begin with a partial-CA and then apply a series of transformations to it until all valid t-tuples have been covered. Usually, once a CA has been constructed, the algorithm will go on to find another CA of smaller size. This group of algorithms include hill climbing, greet flood, tabu search [16], [35], [36], simulated annealing (SA) [37], [38], [19], genetic algorithm [39], [40], etc.

Tabu search approach has also been utilized by Hernandez *et al.* [36]. Their algorithm uses a mixture of three neighborhood functions. Each of these functions has been assigned a probability to be selected by the algorithm. This algorithm is designed to solve CAG without constraints, that is, it can not be use to generate CA when constraints exist.

To cover as many t-tuples as possible in the earliest tests, Bryce *et al.* [35] use a one-test-at-a-time greedy algorithm to initialize test cases, and then apply heuristic search to increase the number of t-tuples covered in each test. Tabu search is one of the heuristic search approach they study. However, the goal of their algorithm differs from ours in that they aim at rapid t-tuple coverage instead of smallest CA.

Our *TCA* algorithm is also a meta-heuristic search algorithm. In fact, the greedy mode of our algorithm is based on the tabu search proposed by Nurmela [16]. We would like to note that there is no publicly available implementation of Nurmela's approach [16] and the original tabu search algorithm can not be used to solve CAG with constraints. Our algorithm enhances it with the capability of handling constraints. Moreover, to avoid being stuck in local optima and to better explore the search space, *TCA* works in the random walk with a probability p. Different from the algorithms beginning with a partial-CA whose size is ordinarily estimated, our *TCA* algorithm uses a construction procedure to generate an initialized CA, usually of smaller size.

The third group is mathematic method which has two kinds of flavors. The first one is based on the mathematical function, and it is generally an extension of the mathematical methods for constructing orthogonal arrays [4], [21]. The other one is based on a recursive construction process that builds a larger CA from smaller ones [41], [42]. Compared to the former two groups, the mathematic method can construct small-sized CAs for some specific benchmarks within extremely short time. However, it can only apply to those benchmarks which comply with certain restrictions.

In addition to the first three groups of algorithms which solve CAG directly, there is another way to generate CAs. Banbara *et al.* [43] and Yamada *et al.* [44] encode the CAG problem into the SAT problem and using constraint solver to solve it. These algorithms can benefit from the powerful capacity of current constraint solvers, while the main challenge is to design an effective way that transforms CAG to CSP. It is worth noting that the *Calot* solver proposed by Yamada *et al.* achieves very good results for 2-way CAG, particularly, for more than half of the 35 benchmarks it obtains the smallest CAs and proves the optimality within reasonable time [44]. Nevertheless, 3-way CAG is still a challenge of *Calot* [44]. On the contrary, our *TCA* algorithm solves CAG without encoding procedure. Therefore, it can make use of the structural information of the problem directly. The size of generated CAs of *TCA* is comparable to that of *Calot* for 2-way benchmarks. Moreover, *TCA* can solve 3-way CAG efficiently.

Recently, Jia *et al.* propose a CAG algorithm *HHSA* using hyperheuristics search [45]. The algorithm consists of two layers to generate CAs. The outer layer is a Simulated Annealing algorithm, while the inner layer uses a reinforcement learning agent to search for the best candidate operators for the outer layer heuristics in the current problem state.

With regards to constraints handling, Cohen *et al.* utilized a SAT solver to enhance an *AETG*-like CAG algorithm for dealing with constraints [18]. The algorithm exploits the history of the SAT solver to improve its performance, and hence is tightly coupled with the SAT solver. Garvin *et al.* [1], [10] integrated a SAT solver into their SA algorithm to check whether each modification of partial-CA violates the constraints. It was found that the original SA algorithm does not couple well with the constraints handling capability in terms of effectiveness. Therefore, several improvements have been made to the original SA algorithm, resulting in the well known algorithm *CASA*. In this work, the integration of a SAT solver to the algorithm is straightforward and simple, and the experimental results reveal that the algorithm couples well with the constraint handling capability. Unlike the *AETG*-like algorithms mentioned above, *CASA* and *TCA* are independent from the SAT solver and thus can use another kind of constraint solver to check the validation of each step. For the more details about constraint handling, the readers could refer to Grindal *et al.*[46], which investigates four different kinds of constraint handling methods and provides a good guideline on when to use each method.

Two-mode framework has shown its success in solving NP-hard problems such as the Boolean satisfiability (SAT) problem [11], [12], [13] and the maximum satisfiability (MAX-SAT) problem [14], [15], resulting in a number of state-of-the-art heuristic algorithms. This inspires us to utilize the two-mode framework for constrained CAG. Extensive experimental results reveal that such a framework is effective in the constrained CAG problem.

Search based software engineering [47] uses meta-heuristic search techniques, such as genetic algorithms, simulated annealing and tabu search, to solve software engineering problems. These problems, such as test case generation [48], program refactoring [49], prioritization for regression testing [50], and module clustering [51], can usually be modeled as optimization problems. Then meta-heuristic search techniques can be used to find near-optimal solutions. In a broader sense, our work belongs to this kind of research.

VI. Conclusions and Future Work

In this work, we propose a two-mode meta-heuristic framework for 3-way constrained CAG solving. Based on this framework, we develop an efficient meta-heuristic algorithm called *TCA*. Experimental results on a broad range of benchmarks show that our algorithm *TCA* can generate CAs with obviously smaller size than state-of-the-art algorithms including a meta-heuristic algorithm *CASA* and a greedy construction algorithm *ACTS*, as well as a recently proposed greedy construction algorithm *Cascade*.

As the generated CAs by our algorithm are used for combinatorial interaction testing, we would like to learn the fault-detection capability of the test cases generated based on our *TCA*. It is worth noting that our *TCA* algorithm can be used to solve t-way constrained CAG for any t, so it would be interesting to follow the work proposed by Petke *et al.* [52] to investigate the efficiency of our *TCA* algorithm in generating higher-strength CAs. On the other hand, since two-mode framework has shown its effectiveness for this problem, we wonder whether it can also work effectively in solving other search based software engineering problems (program refactoring, prioritization for regression testing, *etc*).

Acknowledgement

This work is supported in by part by the National Key Basic Research Program (973 Program) of China under Grant No. 2014CB340701, the High-Tech Research and Development Program of China under Grant No. 2015CB352201, the National Natural Science Foundation of China under Grant Nos. 61472369, 61225007 and 61432001, and the ARC project No. DP150101618.

REFERENCES

[1] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Search Based Software Engineering, 2009 1st International Symposium on*, 2009, pp. 13–22.

[2] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 418–421, 2004.

[3] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Trans. Software Eng.*, vol. 32, no. 1, pp. 20–34, 2006.

[4] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, p. 11, 2011.

[5] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, 2007, pp. 129–139.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatiorial design," *IEEE Trans. Software Eng.*, vol. 23, no. 7, pp. 437–444, 1997.

[7] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98), 13-14 November 1998, Washington, D.C, USA, Proceedings*, 1998, pp. 254–261.

[8] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 18-22, 2013*, 2013, pp. 242–251.

[9] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *Journal of Systems and Software*, vol. 98, pp. 191–207, 2014.

[10] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

[11] C. M. Li and W. Huang, "Diversification and determinism in local search for satisfiability," in *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, 2005, pp. 158–172.

[12] A. Balint and A. Fröhlich, "Improving stochastic local search for SAT with a new probability distribution," in *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, 2010, pp. 10–15.

[13] S. Cai and K. Su, "Local search for boolean satisfiability with configuration checking and subscore," *Artif. Intell.*, vol. 204, pp. 75–98, 2013.

[14] S. Cai, C. Luo, J. Thornton, and K. Su, "Tailoring local search for partial maxsat," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, pp. 2623–2629.

[15] C. Luo, S. Cai, W. Wu, Z. Jie, and K. Su, "CCLS: an efficient local search algorithm for weighted maximum satisfiability," *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 1830–1843, 2015.

[16] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

[17] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, 2003, pp. 502–518.

[18] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Software Eng.*, vol. 34, no. 5, pp. 633–650, 2008.

[19] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, 2003, pp. 38–48.

[20] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *14th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2007), 26-29 March 2007, Tucson, Arizona, USA*, 2007, pp. 549–556.

[21] J. Zhang, Z. Zhang, and F. Ma, *Automatic Generation of Combinatorial Test Data*, ser. Springer Briefs in Computer Science. Springer, 2014.

[22] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. CRC press, 2013.

[23] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A framework of greedy methods for constructing interaction test suites," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 146–155.

[24] G. Sherwood, "Effective testing of factor combinations," in *Proc. of the Third Int. Conf. on Softw. Test., Anal., and Rev.(STAR94), Washington, DC, Software Quality Eng*, 1994.

[25] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Softw. Test., Verif. Reliab.*, vol. 17, no. 3, pp. 159–182, 2007.

[26] ——, "A density-based greedy algorithm for higher strength covering arrays," *Softw. Test., Verif. Reliab.*, vol. 19, no. 1, pp. 37–53, 2009.

[27] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of the 24th Pacific Northwest Software Quality Conference*, 2006.

[28] Y.-W. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Aerospace Conference Proceedings, 2000 IEEE*, vol. 1, 2000, pp. 431–437.

[29] C. Nie, B. Xu, L. Shi, and Z. Wang, "A new heuristic for test suite generation for pair-wise testing," in *Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2006), San Francisco, CA, USA, July 5-7, 2006*, 2006, pp. 517–521.

[30] C. Nie, B. Xu, L. Shi, and G. Dong, "Automatic test generation for n-way combinatorial testing," in *Quality of Software Architectures and Software Quality, First International Conference on the Quality of Software Architectures, QoSA 2005 and Second International Workshop on Software Quality, SOQUA 2005, Erfurt, Germany, September 20-22, 2005, Proceedings*, 2005, pp. 203–211.

[31] C. Nie, B. Xu, Z. Wang, and L. Shi, "Generating optimal test set for neighbor factors combinatorial testing," in *Sixth International Conference on Quality Software (QSIC 2006), 26-28 October 2006, Beijing, China*, 2006, pp. 259–265.

[32] Z. Wang, C. Nie, and B. Xu, "Generating combinatorial test suite for interaction relationship," in *Fourth International Workshop on Software Quality Assurance, SOQUA 2007, in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3-4, 2007*, 2007, pp. 55–61.

[33] Z. Wang, B. Xu, and C. Nie, "Greedy heuristic algorithms to generate variable strength combinatorial test suite," in *Proceedings of the Eighth International Conference on Quality Software, QSIC 2008, 12-13 August 2008, Oxford, UK*, 2008, pp. 155–160.

[34] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Softw. Test., Verif. Reliab.*, vol. 18, no. 3, pp. 125–148, 2008.

[35] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, 2007, pp. 1082–1089.

[36] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *Combinatorial Optimization and Applications - 4th International Conference, COCOA 2010, Kailua-Kona, HI, USA, December 18-20, 2010, Proceedings, Part I*, 2010, pp. 51–64.

[37] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collofello, "Variable strength interaction testing of components," in *27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*, 2003, p. 413.

[38] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *14th International*

Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA, 2003, pp. 394–405.

[39] S. A. Ghazi and M. A. Ahmed, "Pair-wise test coverage using genetic algorithms," in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 2, 2003, pp. 1420–1424.

[40] J. D. McCaffrey, "Generation of pairwise test sets using a genetic algorithm," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009, Seattle, Washington, USA, July 20-24, 2009. Volume 1*, 2009, pp. 626–631.

[41] A. W. Williams, "Determination of test configurations for pair-wise interaction coverage," in *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), August 29 - September 1, 2000, Ottawa, Canada*, 2000, pp. 59–74.

[42] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "A new method for constructing pair-wise covering designs for software testing," *Inf. Process. Lett.*, vol. 81, no. 2, pp. 85–91, 2002.

[43] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers," in *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, 2010, pp. 112–126.

[44] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental SAT solving," in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–10.

[45] Y. Jia, M. Cohen, and M. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *37th International Conference on Software Engineering (ICSE 2015), 16-24 May 2015, Firenze, USA*, 2015, pp. 540–550.

[46] M. Grindal, J. Offutt, and J. Mellin, "Handling constraints in the input space when using combination strategies for software testing," 2006.

[47] M. Harman, "The current state and future of search based software engineering," in *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, 2007, pp. 342–357.

[48] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test., Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[49] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, 2007, pp. 1106–1113.

[50] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Software Eng.*, vol. 33, no. 4, pp. 225–237, 2007.

[51] M. Harman, S. Swift, and K. Mahdavi, "An empirical study of the robustness of two module clustering fitness functions," in *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, 2005, pp. 1029–1036.

[52] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 26–36.