Towards More Efficient Meta-heuristic Algorithms for Combinatorial Test Generation

Jinkun Lin
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
China
jkunlin@gmail.com

Shaowei Cai*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
China
caisw@ios.ac.cn

Chuan Luo
Microsoft Research
China
chuan.luo@microsoft.com

Qingwei Lin
Microsoft Research
China
qlin@microsoft.com

Hongyu Zhang
The University of Newcastle
Australia
hongyu.zhang@newcastle.edu.au

ABSTRACT

Combinatorial interaction testing (CIT) is a popular approach to detecting faults in highly configurable software systems. The core task of CIT is to generate a small test suite called a t-way covering array (CA), where t is the covering strength. Many meta-heuristic algorithms have been proposed to solve the constrained covering array generating (CCAG) problem. A major drawback of existing algorithms is that they usually need considerable time to obtain a good-quality solution, which hinders the wider applications of such algorithms. We observe that the high time consumption of existing meta-heuristic algorithms for CCAG is mainly due to the procedure of score computation. In this work, we propose a much more efficient method for score computation. The score computation method is applied to a state-of-the-art algorithm *TCA*, showing significant improvements. The new score computation method opens a way to utilize algorithmic ideas relying on scores which were not affordable previously. We integrate a gradient descent search step to further improve the algorithm, leading to a new algorithm called *FastCA*. Experiments on a broad range of real-world benchmarks and synthetic benchmarks show that, *FastCA* significantly outperforms state-of-the-art algorithms for CCAG algorithms, in terms of both the size of obtained covering array and the run time.

CCS CONCEPTS

• **Computing methodologies → Heuristic function construction**; • **Software and its engineering → Software testing and debugging**; **Search-based software engineering**.

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338914

KEYWORDS

Combinatorial interaction testing, covering array generation, local search

ACM Reference Format:
Jinkun Lin, Shaowei Cai, Chuan Luo, Qingwei Lin, and Hongyu Zhang. 2019. Towards More Efficient Meta-heuristic Algorithms for Combinatorial Test Generation . In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338914

1 INTRODUCTION

With the increasing requirement on customizable software, modern software systems usually have many configurable options. These highly-configurable systems allow users to control the behaviors of the softwares by setting the options to meet their demand. Besides, the cost of development may be reduced significantly by reusing the systematic components among different variants.

These benefits are in the wake of challenges for the validation of highly-configurable softwares, as failures may be caused by some combinations of options [18, 37]. Testing highly-configurable softwares is an intractable task, mainly because the number of configurations grows exponentially with the number of options. Due to the combinatorial explosion, the enumeration methods become futile, and there is an urgent need to develop more effective and practical methods. As a response to this significant requirement, the combinatorial interaction testing (CIT) approach has emerged as a popular paradigm for detecting option-combination faults of configurable software system. By using combinatorial optimization techniques to sample configurations from the configuration space, CIT can significantly reduce the number of required test cases. CIT has proved useful in many domains such as software product line [15], graphical user interfaces [39], concurrent programs [19] among others. Particularly, a recent research suggests that CIT is more desirable when system faults are hard to detect [34].

The core task of CIT is to generate a test suite as small as possible, by which any t-way combination of values of options is covered at least once. Such a test suite is called a t-way covering array (CA), where t is the covering strength. Empirical studies suggest that most of the failures in highly-configurable systems are caused by

RIGHTSLINK

the interaction of a limited number of t options, usually between two and six [11, 18]. These failures can be revealed efficiently and effectively by a t-way CA.

In most real-world systems, there are hard constraints on the permissible combinations of values of the options. For the sake of the accuracy and the efficiency of the testing process, these constraints must be taken into account when generating CA [5]. This gives rise to the constrained covering array generating (CCAG) problem, which aims at generating CA of minimum size while satisfying all constraints. CCAG is NP hard, and popular practical algorithms for solving CCAG can be classified into three main groups: constraint-encoding algorithms [1, 36, 40], greedy algorithms [4, 21, 35] and meta-heuristic algorithms [8–10, 16, 25]. While constraint-encoding algorithms can effectively solve 2-way CCAG, they typically fail to solve 3-way CCAG. Greedy algorithms focus on generating CAs in a short time, and the size of CA is not its major consideration. Meta-heuristic algorithms can generate better solutions than other approaches, but they usually need considerable time to obtain a good-quality solution, which hinders the wider applications of such algorithms.

This work is dedicated to more efficient meta-heuristic algorithms for CCAG, which can provide good solutions within much shorter time compared to state-of-the-art CCAG algorithms. Meta-heuristic algorithms for CCAG [8–10, 16, 25] are based on a methodology named local search. We discover that the computation of the scores of candidate operations, which are used to guide the search to select the operation to perform, occupies a very high proportion of the time consumption of a local search algorithm for CCAG. According to our investigation on the TCA algorithm [25], this percentage is usually more than 50% and can reach up to 90% for some instances when solving 3-way CCAG. The heavy score computation makes the algorithms slow; and a more serious consequence is that some good algorithmic ideas cannot be used in meta-heuristic algorithms for CCAG, due to the slow score computation.

Thus, to develop efficient local search solvers for CCAG, it is critical to develop an efficient method for score computation. In this work, we propose a method that is much faster than the previous methods for score computation. The proposed method is based on an important observation that only two kinds of value combinations have influence on the scores, and thus only these combinations need to be considered in calculating scores. We apply our novel score computation method to a state-of-the-art meta-heuristic algorithm *TCA*[25]. The experimental results on real-world benchmarks show that the runtime to find CAs of the same size is significant reduced by using our score computation method. In addition, with regard to the size of CAs found within 1 000 seconds, the new algorithm *TCA+* improves *TCA* on 8 out of 26 real-world benchmarks, and performs equally on the rest of the benchmarks.

The new score computation method opens a way to utilize algorithmic ideas relying on scores which were not affordable previously. Based on the new score computation method, we develop a new algorithm named *FastCA*. It improves *TCA* by introducing a gradient descent search step before the two-mode search. Unlike the greedy mode of *TCA* that only considers modifications related to one randomly chosen uncovered combination of values, the gradient descent step considers all uncovered combinations and applies the best modification if it can reduce the number of uncovered t-way combinations. We conduct experiments on a broad range of real-world benchmarks and synthetic benchmarks to compare *FastCA* with the state-of-the-art algorithms for CCAG. Experimental results show that *FastCA* is faster and obtains better solutions than previous heuristics. Specifically, with the time budget of 1 000 seconds, *FastCA* finds CAs of much smaller size than other algorithms on most instances, and worse on none instance. Besides, even the cutoff time for *FastCA* is set to 100 seconds, it still finds better or equal CAs than those found by other algorithms in 1 000 seconds. It means that *FastCA* can generate CA smaller than state-of-the-art meta-heuristic solvers using run time similar to greedy algorithms.

2 PRELIMINARIES

A system under test (SUT) is defined as a pair $M = \langle P, C\rangle$, where P is a set of options and C a set of constraints on the permissible combinations of values of the options in P. For each option $p_i \in P$, the set of feasible values is denoted as V_i. To define the CCAG problem, we need to introduce the notions of *tuple* and *test case*.

Definition 2.1. Given a SUT $M = \langle P, C\rangle$, a tuple $\tau = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \ldots, (p_{i_t}, v_{i_t})\}$ is a set of pairs, which implies that option $p_{i_j} \in P$ takes the value $v_{i_j} \in V_{i_j}$. A tuple of size t is called a t-tuple.

Definition 2.2. Given a SUT $M = \langle P, C\rangle$, a test case tc is a tuple that covers all options in P, that is, it is a complete assignment to P. The value of option p in tc is denoted by $tc[p]$.

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 τ is *covered by a test case tc* if, and only if, $\tau \subseteq tc$, that is, the options in τ take the same values as in tc.

Definition 2.3. Given a SUT $M = \langle P, C\rangle$, a t*-way covering array* $CA(M, t)$ is a set of valid test cases, 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. The size of a CA α is defined as the number of the test cases contained, and it is denoted by $|\alpha|$.

Definition 2.4. Given a SUT $M = \langle P, C\rangle$ and covering strength t, the CCAG problem is to find a t-way CA of minimal size.

2.1 Preliminaries on Local Search for CCAG

As almost all meta-heuristic algorithms for CCAG are based on local search, we also provide the basic concepts used in local search for solving CCAG. Local search CCAG algorithms work with partial covering arries, which are defined as follows.

Definition 2.5. Given a SUT $M = \langle P, C\rangle$, a t*-way partial covering array* (partial-CA) α is a set of valid test cases. A tuple is covered by a partial-CA α iff it is covered by at least one test case of α, and the cost of α, denoted as $cost(\alpha)$, is defined as the number of valid t-tuples not covered by α.

Typically, a local search algorithm for CCAG starts from an initial (partial) CA, and tries to improve the solution by performing small modifications iteratively. Normally, each step of the procedure is to change one option value of a test case, which is formally defined as an operation.

RIGHTSLINK

Algorithm 1: Local search for CCAG

```
   Input: SUT M⟨P, C⟩, covering strength t
   Output: CA α*
1  α ← Initialization();
2  while The termination criterion is not met do
3      if cost(α) = 0 then
4          α* ← α;
5          α ← partial-CA of smaller size;
6      opSet ← candidate operations;
7      forall op ∈ opSet do
8          Calculate score(op);
9      modify α by op ∈ opSet chosen according to socre(op);
10 return α*;
```

Definition 2.6. Given a partial-CA α, an operation, denoted as $op(tc, p_i, v_i)$, modifies the value of one option p_i of a test case tc to be v_i, where $tc \in \alpha$ is the test case to be modified, $p_i \in P$ is the option and $v_i \in V_i$ is the new value for p_i.

Definition 2.7. Given a partial-CA α and an operation op, the score of the operation is defined as $score(op) = cost(\alpha) - cost(\alpha')$, where α' is the resulting partial-CA after applying op to α.

The framework of local search algorithms for CCAG is presented in Algorithm 1. At the beginning, a partial-CA α is generated as starting point for the search procedure (Line 1). During each step of the search procedure (Line 2–10), it first checks whether the cost of α equals to zero. If a CA is found, the algorithm attempts to find CA of smaller size (Line 3–5). Otherwise, it collects candidate operation, and calculates score for each of them (Line 7–9). After that, one of the operation op chosen according to $socre(op)$ is applied to α.

2.2 Benchmarks and Experiment Methodology

In order to study the algorithms, we carry out extensive experiments and report the results in tables. In this subsection, we introduce the benchmarks, the experiment setup and reporting methodology for better understanding the experiment parts.

The experiments are conducted on a broad range of benchmarks including real-world instances and synthetic instances. These benchmarks have been widely used to evaluate the performance of CCAG algorithms [5, 6, 9, 10, 16, 17, 25, 28].

- **Real-world:** This benchmark contains six real-world instances. Five of them are extracted from nontrivial real-world systems, including Apache, Bugzilla, GCC, SPIN-S and SPIN-V. These instances were introduced by Cohen et al. [5, 6][1]. The other instance is TCAS, which was first introduced by Kuhn and Okun [17][2]. It is a traffic collision avoidance system from 'Siemens' suite.
- **IBM:** This benchmark contains 20 real-world instances generated by or for IBM customers [31][3]. These instances cover a broad range of application including banking systems, telecommunications, healthcare, etc.
- **Synthetic:** There are 30 instances in this benchmark. They were generated synthetically from the characteristics found in the five real-world instances. These instances were generated by Garvin et al.[9, 10][1].

All experiments were conducted on a computing cluster consisting of computing nodes equipped with dual 56-core, 2.00GHz Intel Xeon E7-4830 CPUS, 35 MB L3 cache and 256 GB RAM, running Ubuntu (version: 16.04.5 LTS). Because meta-heuristic algorithms are usually randomized, for each algorithm we performed 10 independent runs per instance with a cutoff time (set to 1 000 seconds). All runs were conducted using *runsolver* (version: 3.3.4) [30] to measure CPU time and *GNU Parallel* [32] to manage processes.

For each algorithm on each instance, we report the smallest size ('min') and the averaged size ('avg') found by the respective algorithm over 10 runs. In addition, for each algorithm on each instance, we report the running time ('time') required for finding the optimized CAs averaged over 10 runs, and all running times were measured in CPU seconds. If an algorithm failed to find a CA during all 10 runs, we report size as '–'.

For each instance, the results in bold indicate the best performance in our comparisons. Also, we use boldface to indicate the instances where our proposed algorithms are statistically outperform all its competitors w.r.t. CA size, according to Wilcoxon rank-sum test ($\alpha = 0.05$).

3 SCORE COMPUTATION

In this section, we first show that score computation is the most time-consuming in meta-heuristic algorithms and then propose a novel light-weight method for score computation.

3.1 Time Consumption of Previous Score Computation

The state-of-the-art performance for solving *CCAG* is achieved by the meta-heuristic algorithms, such as the two-mode heuristic algorithm *TCA* and the simulated annealing based algorithm *HHSA* and *CASA*. These algorithms are all based on the local search methodology and the search is guided with the scores of operations.

A commonly used method for computing scores of operations is as follows. For an operation $op(tc, p, v)$, as $score(op)$ measures the change on the number of uncovered valid t-tuples, a straightforward way to compute $score(op)$ is to check each tuple τ involving (p, v). If the checked tuple τ becomes covered from uncovered by the operation op, it contributes +1 to $score(op)$; if τ becomes uncovered from covered, it contributes -1 to $score(op)$.

The above method for score computation is widely used in previous local search algorithms for CCAG. However, this process is of high complexity and is very time-consuming in practice. A simple analysis shows that, there are $\binom{k-1}{t-1}$ tuples to be checked for computing the score of only one operation. For instance, consider the real-world instance *apache* with $k = 172$ options, and suppose the required covering strength $t = 3$, then we need to check 14 535 tuples for computing an operation's score. What is worse, in some modes of local search such as the greedy mode of *TCA*, the number

[1] http://cse.unl.edu/~citportal/public/tools/casa/benchmarks.zip
[2] http://cse.unl.edu/~myra/artifacts/HHSA/downloads/ICSE15_HHSA_Benchmarks.tar.gz
[3] https://researcher.watson.ibm.com/researcher/files/il-ITAIS/ctdBenchmarks.zip

RIGHTSLINK

Table 1: The averaged time consumption (in seconds) of score computation on each benchmark, running TCA on each instance with time budget 1000 seconds.

t-way	Real-world	IBM	Synthetic
2	345.38	250.72	380.82
3	770.52	549.12	856.89
4	774.02	548.1	878.27

of candidate operations, for which score computation is required, is proportional to the size of CA.

To show that the above score computation is time-consuming, we carry out experiments to give empirical evidences about the time cost of score computation in *TCA*, a state-of-the-art meta-heuristic algorithm for solving CCAG. We run *TCA* to solve CCAG with covering strength between 2 to 4 on three benchmarks (which will be introduced in Section 3.3), with time budget limited to 1 000 seconds and one run for each instance. The instances that can not be solved within the time budget (appears in 4-way CCAG, see Table 7 for reference) are not considered in the statistics.

The statistics (Table 1) show that the time consumption of score computation occupies a major part of the time budget, and increases with the covering strength. In particular, for the Real-world benchmark, it cost 34.5%, 77.1% and 77.4% of the time budget respectively for 2, 3 and 4 way covering strength. Therefore, improving the efficiency of score computation is critical to the performance of local search algorithms for solving CCAG.

3.2 A Lightweight Method for Score Computation

In this subsection, we propose a lightweight method for score computation. Before describing the method, we first give a definition and a lemma.

Definition 3.1. Given a t-way partial-CA α, a valid t-tuple τ is *δ-covered* if, and only if, there is exactly δ test cases in α cover it. A 0-covered tuple is indeed an uncovered tuple.

An important fact is that, only uncovered and 1-covered tuples are related to the score of an operation. This observation is formally described below.

Lemma 3.2. *For an operation $op(tc, p, v)$, only uncovered and 1-covered t-tuples may have impact to $score(op)$, and any δ-covered t-tuple with $\delta > 1$ has no impact to $score(op)$.*

Proof. For a tuple $\tau = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \ldots, (p_{i_t}, v_{i_t})\}$, there are three possibility to consider.

(1) τ is uncovered tuple. If, and only if, the following equation holds,

$$v_{i_j} = \begin{cases} tc[p_{i_j}] & p_{i_j} \neq p \\ v & p_{i_j} = p \end{cases} \quad (1)$$

τ will be covered by tc after apply op, contributing 1 to $score(op)$.

(2) τ is 1-covered tuple. If it is not covered by tc, then the coverage of it is unchanged after the applying op. Otherwise, if $(p, v') \in \tau$ and $v' \neq v$, then applying op will make τ uncovered, contributing -1 to $score(op)$.

(3) τ is δ-covered tuple with $\delta > 1$. Suppose the tuple τ is covered by a set of test cases T, and $|T| = \delta > 1$. If $tc \notin T$, obviously, the operation op has no influence on the coverage of τ; if $tc \in T$, then τ either remains δ-covered or becomes $(\delta-1)$-covered tuple (the proof is similar to (2)). Since $\delta > 1$, the tuple τ remains covered before and after the operation, and thus has no impact to $score(op)$. □

Table 2: The averaged number of uncovered tuples and 1-covered tuples over the procedure of *TCA* when solving 3-way CCAG in 1 000 seconds.

Instance	#Prev_check	Uncovered		1-covered	
		#num	%	#num	%
apache	14535	1.35	0.01	16.39	0.11
bugzilla	1275	2.42	0.19	27.44	2.15
gcc	19503	1.59	0.01	17.15	0.09
spins	136	1.34	0.99	8.81	6.48
spinv	1431	1.36	0.1	8.23	0.58
tcas	55	1	1.82	2.9	5.27
Total	36935	9.06	0.02	80.92	0.22
Banking1	6	1.25	20.83	2.63	43.83
Banking2	91	2.17	2.38	9.11	10.01
CommProtocol	45	1.57	3.49	2.99	6.64
Concurrency	6	1.8	30	2.4	40
Healthcare1	36	1.22	3.39	3.38	9.39
Healthcare2	55	1.59	2.89	6.26	11.38
Healthcare3	378	1.48	0.39	9.85	2.61
Healthcare4	561	1.27	0.23	11.55	2.06
Insurance	78	1	1.28	2.35	3.01
NetworkMgmt	28	1	3.57	2.1	7.5
ProcessorComm1	91	1.4	1.54	7.06	7.76
ProcessorComm2	276	1.89	0.68	12.01	4.35
Services	66	1.1	1.67	3.84	5.82
Storage1	3	1	33.33	1.54	51.33
Storage2	6	1.69	28.17	4.35	72.5
Storage3	91	1.08	1.19	2.82	3.1
Storage4	171	1.01	0.59	4.62	2.7
Storage5	231	1.01	0.44	3.94	1.71
SystemMgmt	36	2.16	6	6.58	18.28
Telecom	36	1.25	3.47	3.32	9.22
Total	2291	27.94	1.22	102.7	4.48

Now we describe the lightweight method for score computation. For an operation $op(tc, p, v)$, its score is computed by the following procedure.

Lightweight_score_computation(operation *op*)

(1) calculate $make(op)$, which is the number of uncovered tuples that will become 1-covered after applying op.
(2) calculate $break(op)$, which is the number of 1-covered tuples that will become uncovered after applying op.
(3) $score(op) = make(op) - break(op)$.

By this method, the time complex of calculating score is reduced from $O(\binom{k-1}{t-1})$ to $O(|C_0| + |C_1|)$, where C_0 and C_1 is the set of uncovered and 1-covered tuples respectively. $|C_0|$ and $|C_1|$ are typically much smaller than $\binom{k-1}{t-1}$.

We run *TCA* on real-world instances with time budget 1 000 seconds to count the averaged number of uncovered and 1-covered tuples in one procedure for score computation. The results on 3-way CCAG are presented in Table 2, where "#Prev_check" denotes the number of tuples need to be checked for computing an operation's score in the previous score computation method. When compared to the number of tuples need to be checked in the previous score computation method (i.e., $\binom{k-1}{t-1}$), the numbers of uncovered and 1-covered tuples are several orders smaller. On average, $\frac{|C_0|}{\binom{k-1}{t-1}} = 0.02\%$

RIGHTSLINK

Table 3: *TCA+* vs *TCA* on the Real-world and IBM benchmarks

Instance	*TCA+*		*TCA*		$time_r$
	min(avg)	time	min(avg)	time	
apache	**144(145.4)**	786.99	154(156.1)	871.68	212.57
bugzilla	48(48)	**2.45**	48(48)	9.96	2.45
gcc	**77(79.4)**	522.88	82(83.6)	802.79	310.79
spins	80(80)	**1.31**	80(80)	3.55	1.31
spinv	198(200.2)	**23.38**	198(200.2)	152.27	23.38
tcas	400(400)	**0.03**	400(400)	0.1	0.03
Banking1	45(45)	0.28	45(45)	0.28	0.28
Banking2	30(30)	< 0.01	30(30)	< 0.01	< 0.01
CommProto.	41(41)	**1.25**	41(41)	1.4	1.25
Concurrency	8(8)	< 0.01	8(8)	< 0.01	< 0.01
Healthcare1	96(96)	**0.03**	96(96)	0.04	0.03
Healthcare2	**51(51.9)**	153.7	52(52)	129.18	80.29
Healthcare3	**153(154.4)**	236.42	154(154.8)	283.19	80.53
Healthcare4	**239(239.7)**	505.01	240(241.2)	651.75	145.7
Insurance	6851(6851)	**2.3**	6851(6851)	10.07	2.3
NetworkMg.	1100(1100)	**0.28**	1100(1100)	0.55	0.28
Proc.Comm1	**106(108)**	336.58	108(108.5)	273.87	151.97
Proc.Comm2	126(**126.5**)	285.46	126(126.6)	516.91	218.1
Services	842(848.5)	**174.59**	842(848.5)	218.69	174.58
Storage1	25(25)	< 0.01	25(25)	< 0.01	< 0.01
Storage2	54(54)	< 0.01	54(54)	< 0.01	< 0.01
Storage3	222(222)	**3.53**	222(222)	7.68	3.53
Storage4	910(910)	**10.58**	910(910)	34.68	10.58
Storage5	**1708(1709.7)**	658.86	1710(1712.3)	796.41	327.39
SystemMg.	45(45)	**0.64**	45(45)	0.88	0.64
Telecom	120(120)	**0.07**	120(120)	0.12	0.07
Avg_time				183.31	67.23

and $\frac{|C_1|}{\binom{k-1}{t-1}} = 0.22\%$ for the Real-world benchmark, and these figures are 1.22% and 4.48% for the IBM benchmark.

3.3 Application to Existing Algorithms

The lightweight score computation method is a generic method that can be used in developing any local search algorithm for CCAG that employs scores. Since it can accelerate the existing algorithms, a natural question is whether this acceleration leads to performance improvement. Here, we present the first research question.

RQ1: Can the lightweight score computation method improve existing meta-heuristic algorithms for CCAG?

As the state of the art for solving CCAG is acquired by *TCA*, we apply the lightweight score computation method to re-implement *TCA*, and the obtained solver is named *TCA+*.

We first measure the speedup of *TCA+* over *TCA*, by comparing the number of steps executed with the cutoff time of 1 000 seconds for all instances (Figure 1). On 31 out of 56 instances, *TCA+* executes 5 times more steps than *TCA* with the same cutoff time. Dramatic speedups are observed on a considerable portion of instances: 10× for 22 instances and 100× for 6 of them.

To study whether the acceleration leads to performance improvement, we conduct experiments to compare the two solvers’ performance in terms of solution quality and run time on all benchmarks. The experimental results on 3-way CCAG are presented in Table 3 and 4. After using the score calculation method proposed, *TCA+* outperforms the original *TCA* on all instances. On the metrics of the size of CAs, *TCA+* is better than *TCA* on 8 out of 26 real-world instances and 19 out of 30 synthetic instances, while worse than *TCA* on none instance. For the instance where *TCA+* and *TCA* find

Table 4: *TCA+* vs *TCA* on the Synthetic benchmark

Instance	*TCA+*		*TCA*		$time_r$
	min(avg)	time	min(avg)	time	
Syn_1	**249(251.6)**	487.89	254(255.6)	847.41	78.52
Syn_2	**139(140.4)**	414.98	141(143.7)	469.11	56.9
Syn_3	51(51)	**9.38**	51(51)	37.39	9.38
Syn_4	80(80)	**7.13**	80(80)	40.02	7.13
Syn_5	**335(337.2)**	790.35	410(413.9)	991.75	301.11
Syn_6	96(96)	**17.63**	96(96)	137.35	17.63
Syn_7	25(**25**)	230.22	25(25.3)	353.54	81.21
Syn_8	**262(263.1)**	643.76	268(270.9)	873.29	95.64
Syn_9	60(60)	**2.69**	60(60)	8.09	2.69
Syn_10	**288(292)**	546.12	323(329)	992.59	152.94
Syn_11	**279(280.2)**	410.68	284(285.5)	909.77	82.81
Syn_12	**217(218.8)**	645.2	237(238.8)	980.72	145.76
Syn_13	180(**180**)	151.61	180(181.7)	732.81	106
Syn_14	216(216)	**27.5**	216(216)	159.48	27.5
Syn_15	150(150)	**18.37**	150(150)	117.57	18.37
Syn_16	96(96)	**16.29**	96(96)	98.77	16.29
Syn_17	**218(219)**	580.21	228(230.7)	923.09	124.63
Syn_18	**289(291.6)**	459.05	301(303.8)	971.85	143.27
Syn_19	**332(335.3)**	743.75	486(492.4)	989.92	498.84
Syn_20	**415(418.4)**	752.5	501(511.6)	991.69	232.06
Syn_21	216(216)	**15.96**	216(216)	88.39	15.96
Syn_22	144(144)	**10.77**	144(144)	45.96	10.77
Syn_23	36(36)	**1.16**	36(36)	4.49	1.16
Syn_24	**293(295.4)**	489.29	299(303.1)	919.7	92.75
Syn_25	**360(362.8)**	596.54	393(395)	993.33	129.26
Syn_26	**164(165.9)**	538.64	167(169.5)	773.54	74.21
Syn_27	180(180)	**7.32**	180(180)	45.08	7.32
Syn_28	**380(383.7)**	856.02	503(506.5)	989.24	479.19
Syn_29	125(**125**)	146.94	125(125.4)	708.87	126.15
Syn_30	**68(68.7)**	353.31	69(69.7)	432.07	52.66
Avg_time				554.23	106.27

CAs of the same size, *TCA+* is always faster than *TCA*. To further show the efficiency of the score computation method, we report the running time (‘$time_r$’) averaged over 10 runs when *TCA+* find CAs of the same size with *TCA* on each instances. As shown in the last column in Table 3 and Table 4, *TCA+* uses much less time to find CAs found by *TCA* in 1 000 seconds. For example, on the apache instance, the running time needed for *TCA+* is only 212.57 seconds while *TCA* needs 871.68 seconds. More significant improvements can be seen on the synthetic benchmarks.

It is also worthy to remark that, since most meta-heuristic algorithms for CCAG use scores to guide the search, our method for score computation is also useful to improve other algorithms.

4 THE *FASTCA* ALGORITHM

Thanks to the lightweight score computation method, some algorithmic strategies that are not affordable previously now can be implemented with affordable time consumption. In this section, we further improve the *TCA* algorithm by integrating a gradient descent mode, which relies heavily on score computation, leading to a new algorithm called *FastCA*.

TCA works between the random mode and greedy mode. While the random mode is for better exploring the search space, the greedy mode focuses on finding good operations to reduce uncovered tuples of the partial-CA. However, *TCA* searches for good operations (w.r.t. scores) within a limited space. It only considers operations involving one selected uncovered tuple. Therefore, there is a considerable probability that the score of the chosen operation is negative,

RIGHTSLINK

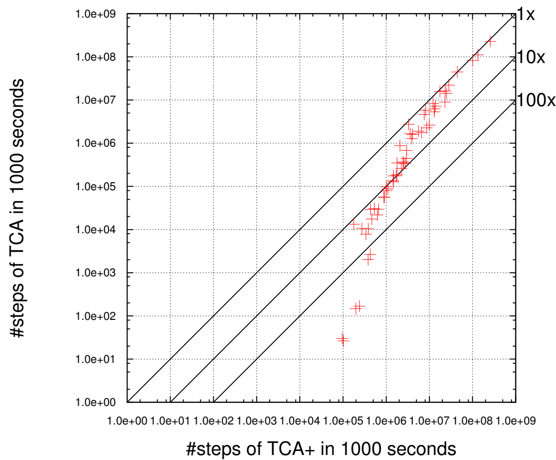

Figure 1: The number of steps of *TCA+* and *TCA* for solving 3-way CCAG on real-world and synthetic benchmarks.

Algorithm 2: The *FastCA* algorithm

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

```
1  α ← Initialization();
2  α* ← α;
3  while The termination criterion is not met do
4      if there is no uncovered tuple then
5          α* ← α;
6          Remove one row from α;
7      α ← Gradient_descent();
8      if Gradient_Descent is not successful then
9          if With probability p then
10             α ← Random_Mode();
11         else
12             α ← Greedy_Mode();
13 return α*;
```

indicating an increment on the number of uncovered tuples, even when there are other operations of positive score.

The considerations above suggest that the exploitation (corresponding to operations with positive scores) of the search in *TCA* should be enhanced. According to research on local search, whenever there is an operation leading to a better objective function, it should be taken [23, 26]. A reason that *TCA* does not adopt such strategies, might be that the original score computation in *TCA* is quite time-consuming, making it not affordable to check many uncovered tuples. Thanks to the lightweight score computation method proposed in this work, the dilemma no longer exists, making it possible to integrate exploitation strategies to improve the algorithm. This results in a new meta-heuristic algorithm called *FastCA*.

4.1 The Description of *FastCA*

The pseudo-code of the *FastCA* algorithm is described in Algorithm 2. In the beginning, *FastCA* calls an initialization function to construct a CA, which serves as a starting point for the search procedure (Line 1). We use a well-known greedy CCAG solver *ACTS* [21] in this procedure. While the *ACTS* solver does not aim at CAs of optimal size, it can generate a CA of reasonable size quickly.

After the initialization, *FastCA* executes a loop of iterative modification steps until the termination criterion is met (it is time budget usually). Whenever the algorithm successfully finds a CA α of size n which covers all valid tuples, it removes one test case from α and goes on to search for a CA of size $n-1$ (Line 4–6). During the search process, there are three important components in *FastCA*, including the gradient descent procedure, the greedy mode searching, and the random mode searching.

As the name indicates, in the gradient descent search procedure, *FastCA* employs a gradient descent step according to the scores of operations. It checks all uncovered valid t-tuples to search for descent operations (i.e. with positive scores). For each uncovered valid t-tuples, the algorithm calculates the score of each operation which can be performed (subject to the tabu condition) to cover the tuple. If there are any operation with positive score, then the one with the greatest score is executed.

If there is no such operation, *FastCA* executes a two-mode search process similar to *TCA*. With a probability p, it works in the random mode; otherwise (with a probability $1-p$), it works in the greedy mode. Unlike the gradient descent search procedure, only one uncovered tuple is considered in this process. One can refer to the description on *TCA* in the related work (Section 6) for details of the two-mode search process.

4.2 Gradient Descent Function

The gradient descent function of *FastCA* employs the tabu search heuristic [25, 29]. Before describing the details, we give several definitions and notations below for better understanding the algorithm.

Definition 4.1. Given a partial-CA α and a test case tc of α, an option p is a tabu option of tc if and only if $tc[p]$ has been changed during the last T search steps, where T is the tabu tenure. For simplification, we say (tc, p) is of tabu status.

Definition 4.2. Given a partial-CA α and an uncovered tuple τ, an operation $op(tc, p, v)$ is feasible if and only if it meets all these criteria: (tc, p) is not of tabu status, the resulting tc is valid and covers τ. The set of feasible operations w.r.t. an uncovered tuple τ is denoted by $opSet(\tau)$.

The pseudo-code of the gradient descent search is described in Algorithm 3. Different from the greedy mode of *TCA*, it checks all the uncovered valid t-tuples under the current partial-CA α. For each tuple τ, the score of each feasible operation is calculated using the lightweight score computation method proposed in section 3. Each time a better score is found, $bestScore$ is updated accordingly, and the responding operation is recorded (Line 5–6).

After all the scores related to uncovered valid tuples have been checked, if $bestScore > 0$, the operation $bestOp$ of which the score is $bestScore$ is applied to α (Line 8–9). Otherwise, the gradient descent process fails, and α stays unchanged.

RIGHTSLINK

Algorithm 3: Gradient_descent

Input: partial-CA α, a set of uncovered valid t-tuples S
Output: partial-CA α

```
1  bestScore ← −∞;
2  forall τ ∈ S do
3      forall op ∈ opSet(τ) do
4          Calculate score(op);
5          if score(op) > bestScore then
6              bestScore ← score(op);
7              bestOp ← op;
8  if bestScore > 0 then
9      α ← Execute bestOp to α;
10 return α
```

5 EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the efficiency and effectiveness of *FastCA* by comparing it with 4 state-of-the-art algorithms for CCAG. Since 2-way CCAG can be solved effectively by existing algorithms [36], the experiments focus on 3-way CCAG. In addition, although 4-way CA can reveal more failures of systems, it remains very difficult for current solvers and few related work report results on it. In our work, we also investigate the performance of current state-of-the-art algorithms for solving 4-way CCAG, and compare that with the performance of *FastCA*.

5.1 Research Questions

We present the research questions before describing the details of experiments.

RQ2: How does *FastCA* compare against current state-of-the-art algorithms for 3-way CCAG?

The size of CA implies the number of test case needed to validate the correctness of a system. Therefore, to find CAs as small as possible is the primary goal of CCAG. In our experiments, we set the time limit to 1 000 seconds and compare *FastCA* against its competitors.

RQ3: Can *FastCA* outperform the current state-of-the-art algorithms for 3-way CCAG with a smaller time budget?

Meta-heuristic algorithms can find smaller CAs than greedy algorithms, with more time consumption. Greedy algorithms can generate CAs in a short time, but the size of CAs are usually larger. *FastCA* is proposed with expectation that it combines the advantage of greedy algorithms and meta-heuristic algorithms. Thus, we are interested in whether *FastCA* is efficient enough to provide good solutions in short time. In our experiments, we limit the runtime of *FastCA* to 100 seconds, and compare it against other solvers with runtime limited to 1 000 seconds.

RQ4: What is the performance of *FastCA* and its competitors for 4-way CCAG?

It is well recognized that 4-way CCAG solving remains a challenge for existing CCAG solvers, due to the huge size of search space. In order to demonstrate the generality and the superiority of *FastCA*, we conduct experiments to compare *FastCA* with its competitors for solving 4-way CCAG.

5.2 State-of-the-art Competitors

In this subsection, we compare *FastCA* with 4 state-of-the-art CCAG algorithms, i.e., *TCA* [25], *HHSA* [16], *CASA* [9] and *ACTS* [38]. Unlike the former three meta-heuristic algorithms, *ACTS* is a greedy algorithm. It can construct CAs in a short time, while the size is generally larger than meta-heuristic algorithms. As it is used in the initialization of *FastCA*, we also included *ACTS* in the competition to evaluate the improvement of *FastCA*'s search procedure. These solvers are all available online.

TCA[4] is an efficient two-mode meta-heuristic algorithm. It works between the greedy mode and the random mode. As reported in the literature[25], *TCA* can generate CAs of obviously smaller size than its competitors.

HHSA[5] is a hyper-heuristic search algorithm. It works in a simulated annealing framework, using a reinforcement learning agent to dynamically choose different strategies without active supervision.

CASA[6] is a simulated annealing algorithm which is improved from previous algorithms by using a reorganized search space based on the CCAG problem structure.

ACTS[7] is an in-parameter-order algorithm, which can handle constraints efficiently. While the former three solvers are written in C++, *ACTS* is developed using Java.

5.3 Experimental Setup

The experimental setup used in this section is the same as the one described in Section 2.2. In the last row of the tables, we report the number of instances where *FastCA* finds better or equal sized CA than its competitors. In addition, in order to show the efficiency of *FastCA*, we also evaluate *FastCA* with the cutoff time of 100 seconds per run for solving 3-way CCAG on all testing benchmarks.

5.4 Experimental Results

In this subsection, we present the experimental results to answer each research question mentioned in Section 5.1.

Results on comparing *FastCA* against its competitors for 3-way CCAG (RQ2): Tables 5 and 6 show the results of *FastCA* (Column '*FastCA (1000s)*') and its competitors for 3-way CCAG. When the cutoff time is limited to 1 000 seconds, as can be seen from the tables, *FastCA* significantly outperforms other solvers on both the real-world and synthetic benchmarks. On the metrics of CA size, *FastCA* finds better solutions than the best competitor *TCA* on 29 out of 56 instances, and finds equal sizes on the remaining ones.

For the Real-world and IBM benchmarks, we focus on the 'difficult' instances which need more than 100 seconds averaged runtime for all solvers excepting the greedy algorithm *ACTS*. On these instances, the size of CAs found by *FastCA* are usually much smaller than its competitors. For example, on the apache instance, the averaged size found by *FastCA* is 134.7, while the number for *TCA*

[4] https://github.com/jkunlin/TCA
[5] http://www0.cs.ucl.ac.uk/staff/Y.Jia/projects/cit_hyperheuristic/downloads/Comb_Linux_64.tar.gz
[6] http://cse.unl.edu/~citportal/
[7] http://www.flossic.com/ACTS/ACTS2.92.zip

RIGHTSLINK

Table 5: Comparing *FastCA* against state-of-the-art competitors for 3-way CCAG on the Real-world and IBM benchmarks. The running time is measured in CPU second. The cutoff time is set to 100s for *FastCA*, and 1000s for the other solvers

Instance	*FastCA (1000s)*		*FastCA (100s)*		*TCA (1000s)*		*ACTS (1000s)*		*HHSA (1000s)*		*CASA (1000s)*	
	min (avg)	time	min (avg)	time	min (avg)	time	min	time	min (avg)	time	min (avg)	time
apache	**133 (134.7)**	716.77	**141 (142.7)**	79.12	154 (156.1)	871.68	173	7.92	–	>1000	245 (247.9)	920.36
bugzilla	**48 (48)**	17.35	**48 (48)**	17.35	**48 (48)**	9.96	68	0.44	60 (60.9)	481.55	61 (64.6)	36.38
gcc	**75 (76.8)**	561.74	**79 (80.6)**	75.44	82 (83.6)	802.79	108	9.48	–	>1000	134 (140)	943.47
spins	**80 (80)**	1.17	**80 (80)**	1.17	**80 (80)**	3.55	98	0.37	**80** (85.7)	59.55	94 (100.5)	7.14
spinv	**195 (196)**	415.45	**196 (197.4)**	65.26	198 (200.2)	152.27	286	1.27	–	>1000	224 (233.1)	734.92
tcas	**400 (400)**	0.47	**400 (400)**	0.47	**400 (400)**	0.1	405	0.32	**400 (400)**	337.88	**400** (404.1)	4.27
Banking1	**45 (45)**	4.11	**45 (45)**	4.11	**45 (45)**	0.28	58	2.07	**45 (45)**	9.9	**45** (46.2)	0.09
Banking2	**30 (30)**	0.66	**30 (30)**	0.66	**30 (30)**	< 0.01	39	0.44	**30 (30)**	1.1	**30** (30.4)	0.35
CommProto.	**41 (41)**	16.68	**41 (41)**	16.68	**41 (41)**	1.4	49	3.22	**41 (41)**	76.94	**41** (42.2)	0.25
Concurrency	8 (8)	0.31	8 (8)	0.31	8 (8)	**<0.01**	8	0.51	8 (8)	7.51	8 (8)	**<0.01**
Healthcare1	**96 (96)**	0.8	**96 (96)**	0.8	**96 (96)**	0.04	105	0.65	**96 (96)**	53.61	**96** (96.6)	0.3
Healthcare2	**50 (50.9)**	225.47	**50 (51.4)**	26.74	52 (52)	129.18	67	1.26	51 (52.1)	23.5	53 (55.1)	6.16
Healthcare3*	**151 (151.5)**	325.85	**151 (152.4)**	48.69	154 (154.8)	283.19	209	0.92	177 (186.9)	373.89	170 (175)	237.96
Healthcare4	**238 (239)**	417.3	**240 (240.7)**	56.61	240 (241.2)	651.75	294	1.21	320 (346.667)	725.21	278 (286.7)	835.15
Insurance	**6851 (6851)**	1.74	**6851 (6851)**	1.74	**6851 (6851)**	10.07	6866	0.54	–	>1000	7027 (7156.4)	770.29
NetworkMg.	**1100 (1100)**	1.14	**1100 (1100)**	1.14	**1100 (1100)**	0.55	1125	0.59	**1100 (1100)**	440.68	1124 (1136.8)	5.72
Proc.Comm1*	**104 (104.8)**	160.59	**105 (105.3)**	32.23	108 (108.5)	273.87	163	0.63	114 (117.6)	90.78	117 (120.7)	111.51
Proc.Comm2	**125 (125.6)**	189.36	**125 (126.2)**	53.81	126 (126.6)	516.91	161	1.64	140 (148.2)	572.5	140 (145)	236.73
Services*	**813 (815.2)**	685.53	**829 (834.2)**	81.4	842 (848.5)	218.69	963	10.35	840 (860)	789.42	856 (894)	464.39
Storage1	25 (25)	2.05	25 (25)	2.05	25 (25)	**<0.01**	25	1.52	25 (25)	15.53	25 (25)	< 0.01
Storage2	**54 (54)**	0.09	**54 (54)**	0.09	**54 (54)**	< 0.01	74	0.03	**54 (54)**	15.9	**54** (55.8)	0.02
Storage3	**222 (222)**	3.43	**222 (222)**	3.43	**222 (222)**	7.68	239	1.54	224 (225.1)	675.16	241 (245.8)	1.83
Storage4	**910 (910)**	3.62	**910 (910)**	3.62	**910 (910)**	34.68	990	0.76	960 (960)	853.39	926 (951.6)	723.84
Storage5	**1705 (1706.9)**	445.17	**1707 (1710.3)**	72.45	1710 (1712.3)	796.41	1879	2.93	–	>1000	1877 (1958.3)	971.23
SystemMg.	**45 (45)**	1.65	**45 (45)**	1.65	**45 (45)**	0.88	60	0.49	**45** (45.2)	16.6	47 (48.3)	0.3
Telecom	**120 (120)**	0.57	**120 (120)**	0.57	**120 (120)**	0.12	126	0.53	**120 (120)**	12.4	**120** (120.4)	0.37
#Better (#Euqal) of *FastCA*					10 (16)		24 (2)		16 (10)		24 (2)	

is 156.1, for *CASA* is 247.9; *HHSA* fails to solve apache within the time limit. The gap between different solvers is even larger on the synthetic benchmark as can be seen from Table 6.

Since *FastCA* uses the greedy algorithm *ACTS* as its initialization, we study how much the search procedure can reduce the size of CAs. As shown in the tables, *FastCA* generates CAs of significant smaller size than *ACTS* on almost all the instances. Besides, for 11 out of 26 real-world instances and 24 out of 30 synthetic instances, it reduces the size of CAs found by *ACTS* by more than 20 test cases. In particular, on the Services instance, it reduces the size from 963 to 813, totally 150 test cases.

These results indicate that *FastCA* advances the state of the art for solving the 3-way CCAG problem.

Results on comparing *FastCA*, with considerably less cutoff time, against its competitors for 3-way CCAG (RQ3): The results are presented in Tables 5 and 6 (Column '*FastCA (100s)*'). *FastCA* with the cutoff time of 100 seconds still finds better or equal sizes of CAs than all its competitors with the 10 times longer cutoff time on all testing instances. Besides, the gaps between *FastCA* and its competitors observed in RQ2 also appear here. Particularly, on the gcc instance, *FastCA* finds CAs of averaged size 80.6 in 75.44 seconds, while *TCA* finds 83.6 in 802.79 seconds and *CASA* finds 140 in 943.47 seconds. Overall, the runtime for *FastCA* is much shorter than other solvers for achieving solutions with better or equal size.

When it comes to the comparison between *FastCA* and *ACTS*, it is shown that 100 seconds is sufficient to dramatically reduce the size of CAs. Averagely, more than 30 test cases are reduced by the search procedure of *FastCA*. Therefore, if a test case takes more than three seconds to run in the test process, then *FastCA* is preferred. For the situation where CA is repeatedly used, such as regression testing, the advantage of *FastCA* in practice is more obvious.

Results on the performance of *FastCA* and its competitors for 4-way CCAG (RQ4): Table 7 presents the results of 4-way CCAG solving on the Real-world and IBM benchmarks; the results on the Synthetic benchmarks are available online[8]. Since the *HHSA* algorithm available online cannot solve 4-way CCAG, we do not report its result here.

It is shown that *FastCA* outperforms other algorithms on all instances. It can generate smaller CAs in shorter time than its competitors. For many instances, such as spinv, Healthcare4 and Storage4, the sizes of CAs found by *FastCA* is several hundreds or even thousands smaller than those found by other solvers, indicating the superiority of *FastCA* on 4-way CCAG.

Discussion on the results of *FastCA* for 2-way CCAG: For comprehensive evaluation, we summarize the empirical results for 2-way CCAG here. On all instances, *FastCA* finds CAs of smaller or equal size than its competitors. In particular, it finds better CAs than *TCA* on 15 instances. To achieve solution of equal size, *FastCA* is also faster than other solvers on nontrivial instances.

Discussion on the effectiveness of the components underlying *FastCA*: In order to illustrate the contribution of gradient descent search to *FastCA*, we remove this component from *FastCA*, resulting in an alternative version named *FastCA*-. Experimental results on real-world benchmarks for 3-way CCAG show that with the gradient descent search, *FastCA* generates smaller CAs on 10

[8]https://github.com/jkunlin/fastca

RIGHTSLINK

Table 6: Comparing *FastCA* against state-of-the-art competitors for 3-way CCAG on the Synthetic benchmarks. The running time is measured in CPU second. The cutoff time is set to 1000s for all solvers.

Instance	*FastCA (1000s)*		*FastCA (100s)*		*TCA (1000s)*		*ACTS(1000s)*		*HHSA(1000s)*		*CASA(1000s)*	
	min (avg)	time	min (avg)	time	min (avg)	time	min	time	min (avg)	time	min (avg)	time
Syn_1	**243 (244.6)**	471.01	**247 (248.3)**	81.69	254 (255.6)	847.41	293	3.3	–	>1000	358 (366.3)	899.32
Syn_2	**133 (134)**	498.59	**136 (138)**	72.75	141 (143.7)	469.11	174	1.67	–	>1000	174 (182.8)	800.84
Syn_3	**51 (51)**	7.18	**51 (51)**	7.18	**51 (51)**	37.39	71	0.36	59 (59.5)	146.55	59 (61.1)	2.76
Syn_4	**80 (80)**	5.49	**80 (80)**	5.49	**80 (80)**	40.02	102	0.72	100 (101.7)	560.71	96 (103.6)	88.82
Syn_5	**330 (332)**	573.81	**338 (339.3)**	87.73	410 (413.9)	991.75	386	13.76	–	>1000	1068 (1069.3)	947.22
Syn_6	**96 (96)**	10.36	**96 (96)**	10.36	**96 (96)**	137.35	119	1.22	–	>1000	118 (122.6)	624.67
Syn_7	**25 (25)**	23.34	**25 (25.1)**	13.27	**25** (25.3)	353.54	35	0.44	26 (26.3)	107.35	27 (27.8)	4.91
Syn_8	**256 (257.5)**	479.04	**260 (262.4)**	89.28	268 (270.9)	873.29	326	4.19	–	>1000	389 (402.5)	962.69
Syn_9	**60 (60)**	4.84	**60 (60)**	4.84	**60 (60)**	8.09	84	0.81	80 (80)	0.14	70 (76.6)	384.72
Syn_10	**277 (280.5)**	520.4	**287 (288.9)**	93.92	323 (329)	992.59	329	9.42	–	>1000	795 (798.1)	947.41
Syn_11	**270 (271.8)**	338.9	**272 (274.6)**	62.37	284 (285.5)	909.77	318	3.79	–	>1000	396 (408.8)	870.45
Syn_12	**216 (216)**	214.18	**217 (217.9)**	84.54	237 (238.8)	980.72	263	6.94	–	>1000	367 (390.5)	953.29
Syn_13	**180 (180)**	32.35	**180 (180)**	32.35	**180** (181.7)	732.81	200	5.45	–	>1000	277 (290.7)	944.55
Syn_14	**216 (216)**	9.24	**216 (216)**	9.24	**216 (216)**	159.48	244	2.69	–	>1000	261 (270.4)	885.73
Syn_15	**150 (150)**	7.56	**150 (150)**	7.56	**150 (150)**	117.57	173	1.97	–	>1000	165 (169.2)	542.73
Syn_16	**96 (96)**	23.97	**96 (96)**	23.97	**96 (96)**	98.77	117	2.58	–	>1000	119 (123.5)	756.91
Syn_17	**216 (216.1)**	429.06	**216 (218.9)**	77.1	228 (230.7)	923.09	265	6.39	–	>1000	338 (351.9)	929.21
Syn_18	**280 (282)**	521.37	**284 (287.6)**	88.85	301 (303.8)	971.85	344	7.24	–	>1000	446 (449)	931.68
Syn_19	**316 (318)**	527.65	**330 (331.7)**	96.81	486 (492.4)	989.92	373	21.58	–	>1000	–	>1000
Syn_20	**411 (412.1)**	623.28	**418 (421.2)**	94.78	501 (511.6)	991.69	463	12.82	–	>1000	1026 (1050.9)	873.88
Syn_21	**216 (216)**	7.39	**216 (216)**	7.39	**216 (216)**	88.39	235	3.05	–	>1000	243 (250.1)	956.52
Syn_22	**144 (144)**	5.16	**144 (144)**	5.16	**144 (144)**	45.96	164	2.07	–	>1000	162 (170.7)	521.59
Syn_23	**36 (36)**	2.62	**36 (36)**	2.62	**36 (36)**	4.49	48	1.02	38 (39.5)	155.17	37 (39.2)	2.68
Syn_24	**284 (285.7)**	430.78	**286 (290.1)**	88.37	299 (303.1)	919.7	341	4.75	–	>1000	448 (462.2)	966.87
Syn_25	**350 (352.4)**	539.08	**357 (358.6)**	89.48	393 (395)	993.33	404	7.55	–	>1000	566 (589.9)	972.05
Syn_26	**160 (161.9)**	685.26	**163 (164.7)**	78.1	167 (169.5)	773.54	207	2.98	–	>1000	216 (220.1)	871.81
Syn_27	**180 (180)**	5.33	**180 (180)**	5.33	**180 (180)**	45.08	204	2	–	>1000	194 (201.6)	579.78
Syn_28	**367 (369.9)**	756.4	**379 (382.5)**	96.4	503 (506.5)	989.24	420	20.25	–	>1000	–	>1000
Syn_29	**125 (125)**	39.91	**125 (125)**	39.91	**125** (125.4)	708.87	154	4.67	–	>1000	186 (192)	920.3
Syn_30	**66 (66.9)**	418.63	**68 (68.3)**	53.39	69 (69.7)	432.07	100	1.25	–	>1000	82 (88.8)	455.48
#Better (#Equal) of *FastCA*					19 (11)		30 (0)		30 (0)		30 (0)	

out of 26 instances than *FastCA-*. For the other 16 instance, *FastCA* and *FastCA-* generate CAs of the same size, and most of them are solved within 10 seconds.

We study the importance of the lightweight score computation by comparing *FastCA* with its alternative *FastCAp* implemented with the previous score computation method on 26 real-world instances of 3-way CCAG. 15 instances are easy for both algorithms - they all find the same sized CAs in 10 seconds. *FastCA* finds better solutions than *FastCAp* on the rest 11 instances, and takes less time (100 to 200 seconds less) on 9 of them.

Indeed, a premise of employing the gradient descent search to improve the algorithm is the lightweight score computation. To give evidence on this, we remove gradient descent search from *FastCAp*, leading to *FastCAp-*. Experiments on 3-way CCAG show that the gradient descent component only improve on 3 instances while decrease the performance on 8 instances on the metric of size of CAs. It makes *FastCAp* worse than *FastCAp-*.

5.5 Threats to Validity

It seems that the cutoff time setting is a threat to validity to our experimental results, since 1000 seconds might not be enough to exploit the performance of *HHSA* and *CASA*. However, compared to the current best-known solutions for most instances reported in literature we could found, 1 000 seconds (even 100 seconds) are enough for *FastCA* to achieve much better solutions. In fact, as reported in the papers of *HHSA* [16] and *CASA* [10], although *HHSA* and *CASA* cost much more time (even more than 10 hours on some instances), the solutions found are still much worse than *FastCA*. Besides, we run *HHSA* and *CASA* on real-world benchmarks with the cutoff time of 5 hours, and *FastCA* (using the cutoff of only 1000 seconds) still significantly outperforms such results.

Since *TCA* is the main competitor in our experiment, we hence follow the experimental setup in *TCA* paper [25] and use 1000 seconds as the cutoff.

6 RELATED WORK

Practical algorithms for solving CCAG can be roughly categorized into three main groups: constraint-encoding algorithms, greedy algorithms and meta-heuristic algorithms. Constraint-encoding algorithms focus on efficient methods to encode CCAG into constrained optimization problem, such as SAT and MaxSat problems [1, 36]. While constraint-encoding algorithms are effective for solving 2-way CCAG, it required improvements to handle 3-way CCAG [36].

Greedy algorithms can usually generate CAs in a short time. Although the size of CAs is not necessarily small, greedy algorithms show its superior in some scenarios where highly optimized test suite is not the primary consideration. One-test-at-a-time (*OTAT*) and in-parameter-order (*IPO*) are two main approaches of greedy algorithms. The well-known *AETG* algorithm is the first one using the *OTAT* strategy [4], and since then a number variants were proposed to improve its performance [2, 3, 7, 33, 35, 40]. The *IOP*

RIGHTSLINK

Table 7: Comparing *FastCA* against state-of-the-art competitors for 4-way CCAG on the Real-world and IBM benchmarks. The running time is measured in CPU second. The cutoff time is set to 1000s for all solvers.

Instance	*FastCA (1000s)*		*TCA (1000s)*		*ACTS (1000s)*		*CASA (1000s)*	
	min (avg)	time	min (avg)	time	min	time	min (avg)	time
apache	–	> 1000	–	> 1000	–	> 1000	–	> 1000
bugzilla	**166 (166.7)**	670.13	170 (170.9)	834.17	242 (242)	3.8	271 (278.5)	910.57
gcc	–	> 1000	–	> 1000	–	> 1000	–	> 1000
spins	**308 (308)**	10.77	311 (317.1)	428.88	393 (393)	0.44	360 (366.9)	781.91
spinv	**1105 (1110.7)**	878.74	1634 (1653.2)	979.58	1631 (1631)	26.37	–	> 1000
tcas	**1200 (1200)**	6.26	**1200 (1200)**	32.55	1435 (1435)	0.43	1228 (1244)	820.78
Banking1	139 (**139**)	5.36	139 (**139**)	< 0.01	139 (**139**)	0.9	139 (142.3)	0.03
Banking2	**71 (71)**	3.45	**71 (71)**	6.97	96 (96)	0.35	74 (77.4)	31.57
CommProtocol	**83 (83)**	8.53	**83 (83)**	2.92	97 (97)	1.28	84 (85.1)	2.45
Concurrency	8 (8)	< 0.01	8 (8)	< 0.01	8 (8)	0.3	8 (8)	< 0.01
Healthcare1	**300 (300)**	1.93	**300 (300)**	1.42	341 (341)	0.43	301 (303.4)	6.18
Healthcare2	**166 (168.1)**	733.72	172 (173.4)	456.59	220 (220)	0.65	184 (187.4)	123.55
Healthcare3	**737 (741.2)**	659.84	766 (770.4)	916.78	1004 (1004)	2.23	1147 (1161.3)	957.1
Healthcare4	**1332 (1341)**	775.18	1728 (1740.5)	997.19	1644 (1644)	5.93	2557 (2598.5)	956.8
Insurance	**75361 (75361)**	159.06	76234 (76270.9)	998.19	75764 (75764)	2.35	972594 (10623174)	968.31
NetworkMgmt	**5610 (5610)**	336.54	**5610 (5610)**	500.86	6267 (6267)	0.61	5944 (5974.9)	970.08
ProcessorComm1	**487 (490.4)**	486.82	491 (495)	577.39	670 (670)	0.52	574 (581.9)	915.21
ProcessorComm2	**584 (585.6)**	405.9	591 (593.4)	867.96	744 (744)	2.69	840 (860.5)	941.5
Services	**6404 (6406.9)**	875.72	6418 (6422.2)	928.33	6855 (6855)	7.92	7081 (7227.3)	943.63
Storage1	25 (25)	1.84	25 (25)	< 0.01	25 (25)	0.7	25 (25)	0.02
Storage2	**162 (162)**	17.68	**162 (162)**	0.15	195 (195)	0.02	**162** (163.6)	0.33
Storage3	**570 (570)**	63.69	**570 (570)**	304.76	752 (752)	0.84	1085 (1085)	35.52
Storage4	**5506 (5508.2)**	927	6625 (6695.5)	999.14	6636 (6636)	2.15	9278 (9370.4)	947.33
Storage5	**11004 (11020.7)**	977.27	14065 (14086.4)	993.27	13292 (13292)	17.63	–	> 1000
SystemMgmt	**135 (135)**	2.64	**135 (135)**	2	152 (152)	0.36	136 (140)	6.25
Telecom	**360 (360)**	2.3	**360 (360)**	2.62	392 (392)	0.38	365 (365.4)	19.96
#Better (#Equal) of *FastCA*			12 (14)		21 (5)		21 (5)	

strategy was first proposed by Lei and Tai [22] to generate 2-way CAs and was later generalized to t-way CAs [20].

Meta-heuristic algorithms aim at reducing the size of CAs with a price of more time consumption. Most state-of-the-art solvers for CCAG, such as *TCA* [25], *HHSA* [16] and *CASA* [9], can be categorized into this group. The main procedure of these algorithms is searching for CAs of iteratively smaller size. There are many strategies have been proposed for improving the effectiveness of the search procedure, including tabu search [29], hyper heuristics [16], simulated annealing and t-set replacement [9]. These strategies are used to guide the search to more promising areas, usually based on the scores of candidate operations.

The *TCA* algorithm uses two-mode heuristics in its searching procedure. The greedy mode of *TCA* devotes to decreasing the number of uncovered tuples, while the random mode provides opportunities to escape from local optimal and better explore the search space. In the random mode of *TCA*, it randomly deletes one test case from the current partial CA, and generates a new one to cover the pre-selected uncovered tuples. When it is in the greedy mode, *TCA* takes one uncovered tuple randomly and choose the best operation to cover it based on the score calculated. As revealed in this work, the score computation costs a primary part of time budget and can be significantly reduced by our lightweight method.

Search based software engineering [12] is a research area that reformulates software engineering problems as optimization problems and then resorts to mete-heuristics algorithms for solving. There are many studies of practical problems, such as test case generation [27], program refactoring [14], prioritization for regression testing [24], and module clustering [13], which all belong to this area of research. In this sense, our work also falls under the research area of search based software engineering.

7 CONCLUSION

The constrained covering array generating (CCAG) problem is a challenging problem in combinatorial test generation. A promising approach to this NP hard problem is to find good solutions by meta-heuristic algorithms. A big issue in meta-heuristic algorithms for CCAG is the heavy time consumption of score computation, which not only accounts for the low speed of the algorithms, but also hinders the applications of many good algorithmic ideas relying on scores. This paper addressed this issue by proposing a lightweight method for score computation. We applied it to re-implement a state-of-the-art meta-heuristic algorithm *TCA* and finds significant speedup as well as performance improvements on real-world and synthetic benchmarks. Additionally, the lightweight score computation method opens a door to make more algorithm ideas affordable and thus more elaborate algorithms can be developed. We took a step of this direction by integrating a gradient descent component to *TCA*, with the lightweight score computation method, leading to a new meta-heuristic algorithm called *FastCA*. Experiments on a broad range of real-world and synthetic benchmarks show that *FastCA* is faster and finds better solutions than existing algorithms, which remains justified even when the cutoff time (100s) for *FastCA* is set to 10% of that (1000s) for the competitors.

ACKNOWLEDGMENTS

This work was supported by Youth Innovation Promotion Association, Chinese Academy of Sciences (No. 2017150).

RIGHTSLINK

REFERENCES

[1] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. 2010. Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers. In *Proceedings of LPAR 2010*. 112–126.

[2] Renée C. Bryce and Charles J. Colbourn. 2007. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability* 17, 3 (2007), 159–182.

[3] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. 2005. A framework of greedy methods for constructing interaction test suites. In *Proceedings of ICSE 2005*. 146–155.

[4] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatiorial Design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.

[5] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of ISSTA 2007*. 129–139.

[6] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2008. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering* 34, 5 (2008), 633–650.

[7] Jacek Czerwonka. 2008. Pairwise testing in the real world: Practical extensions to test-case scenarios. *Microsoft Corporation, Software Testing Technical Articles* (2008).

[8] Philippe Galinier, Segla Kpodjedo, and Giulio Antoniol. 2017. A penalty-based Tabu search for constrained covering arrays. In *Proceedings of GECCO 2017*. 1288–1294.

[9] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2009. An Improved Meta-Heuristic Search for Constrained Interaction Testing. In *Proceedings of 1st International Symposium on Search Based Software Engineering*. 13–22.

[10] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102.

[11] Jon D. Hagar, Thomas L. Wissink, D. Richard Kuhn, and Raghu Kacker. 2015. Introducing Combinatorial Testing in a Large Organization. *IEEE Computer* 48, 4 (2015), 64–72. https://doi.org/10.1109/MC.2015.114

[12] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Proceedings of FOSE 2007*. 342–357.

[13] Mark Harman, Stephen Swift, and Kiarash Mahdavi. 2005. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of GECCO 2005*. 1029–1036.

[14] Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of GECCO 2007*. 1106–1113.

[15] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Software Eng.* 40, 7 (2014), 650–670. https://doi.org/10.1109/TSE.2014.2327020

[16] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *Proceedings of ICSE 2015*. 540–550.

[17] D. Richard Kuhn and Vadim Okun. 2006. Pseudo-Exhaustive Testing for Software. In *Proceedings of SEW 2006*. 153–158.

[18] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[19] Yu Lei, Richard H. Carver, Raghu Kacker, and David Chenho Kung. 2007. A combinatorial testing strategy for concurrent programs. *Softw. Test., Verif. Reliab.* 17, 4 (2007), 207–225. https://doi.org/10.1002/stvr.369

[20] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A General Strategy for T-Way Software Testing. In *Proceedings of ECBS 2007*. 549–556.

[21] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability* 18, 3 (2008), 125–148.

[22] Yu Lei and Kuo-Chung Tai. 1998. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *Proceedings of HASE 1998*. 254–261.

[23] Chu Min Li and Wenqi Huang. 2005. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT 2005*. 158–172.

[24] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (2007), 225–237.

[25] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. 2015. TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation. In *Proceedings of ASE 2015*. 494–505.

[26] Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. 2015. CCLS: An Efficient Local Search Algorithm for Weighted Maximum Satisfiability. *IEEE Trans. Comput.* 64, 7 (2015), 1830–1843.

[27] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

[28] Hanefi Mercan, Cemal Yilmaz, and Kamer Kaya. 2018. CHiP: A Configurable Hybrid Parallel Covering Array Constructor. *IEEE Transactions on Software Engineering* (2018), To appear.

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

[30] Olivier Roussel. 2011. Controlling a Solver Execution with the runsolver Tool. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4 (2011), 139–144.

[31] Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. 2011. Using binary decision diagrams for combinatorial test design. In *Proceedings of ISSTA 2011*. 254–264.

[32] Ole Tange. 2018. *GNU Parallel 2018*. Ole Tange. https://doi.org/10.5281/zenodo.1146014

[33] Yu-Wen Tung and Wafa S Aldiwan. 2000. Automating test case generation for the new generation mission software system. In *Proceedings of IEEE Aerospace Conference 2000*, Vol. 1. IEEE, 431–437.

[34] Huayao Wu, Justyna Petke, Yue Jia, Mark Harman, et al. 2018. An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing. *IEEE Transactions on Software Engineering* (2018).

[35] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. 2016. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of ASE 2016*. 614–624.

[36] Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. 2015. Optimization of Combinatorial Testing by Incremental SAT Solving. In *Proceedings of ICST 2015*. 1–10.

[37] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. 2006. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *IEEE Transactions on Software Engineering* 32, 1 (2006), 20–34.

[38] Linbin Yu, Yu Lei, Mehra Nouroz Borazjany, Raghu Kacker, and D. Richard Kuhn. 2013. An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. 242–251.

[39] Xun Yuan, Myra B. Cohen, and Atif M. Memon. 2011. GUI Interaction Testing: Incorporating Event Context. *IEEE Trans. Software Eng.* 37, 4 (2011), 559–574. https://doi.org/10.1109/TSE.2010.50

[40] Zhiqiang Zhang, Jun Yan, Yong Zhao, and Jian Zhang. 2014. Generating combinatorial test suite using combinatorial optimization. *Journal of Systems and Software* 98 (2014), 191–207.

RIGHTSLINK