FastCA: An Effective and Efficient Tool for Combinatorial Covering Array Generation

Jinkun Lin*†, Shaowei Cai*†§, Bing He*† , Yingjie Fu*†, Chuan Luo‡, Qingwei Lin‡
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
†School of Computer Science and Technology, University of Chinese Academy of Sciences, China
‡Microsoft Research, China
{linjk, caisw, hebing, fuyj}@ios.ac.cn; {chuan.luo, qlin}@microsoft.com

***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. A major drawback of existing solvers for CA generation is that they usually need considerable time to obtain a high-quality solution, which hinders its wider applications. In this paper, we describe FastCA, an effective and efficient tool for generating constrained CAs. We observe that the high time consumption of existing meta-heuristic algorithms is mainly due to the procedure of score computation. To this end, we present a much more efficient method for score computation. Thanks to this new lightweight score computation method, FastCA can work in the gradient mode to effectively explore the search space. Experiments on a broad range of real-world benchmarks and synthetic benchmarks show that FastCA significantly outperforms state-of-the-art solvers, in terms of both the size of obtained covering array and the run time.**

Video: https://youtu.be/-6CuojQIt-k

Repository: https://github.com/jkunlin/FastCATool.git

***Index Terms*—combinatorial interaction testing, constrained covering array, search-based software testing**

I. Introduction

Modern software systems are usually highly-configurable. The behaviours of the system can be controlled by setting configurable options to meet the demand of users. However, it is challenging to validate these software, as failures may be caused by some combinations of options [1], [2], while the number of configurations grows exponentially with the number of options. To remedy this situation, the combinatorial interaction testing (CIT) approach has emerged as a popular paradigm for detecting option-combination faults of configurable software systems. It significantly reduces the number of required test cases, by systematically sampling from the configuration space.

The core task of CIT is to generate a test suite as small as possible, by which each t-way combination of values of options $\langle p_{i_1} = v_{i_1}, \ldots, p_{i_t} = v_{i_t} \rangle$ 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 the interaction of a limited number of t options, usually between two and six [1]. These failures can be revealed by a t-way CA.

§Corresponding author.

In most real-world systems, there are hard constraints on the permissible combinations of values of the options, which must be taken into account when generating CAs [3]. This gives rise to the constrained covering array generation (CCAG) problem, which aims at generating a CA of minimum size while satisfying all constraints. A major drawback of existing solvers for solving CCAG is that they usually need considerable time to obtain a high-quality solution, which hinders its wider applications. While the greedy solvers [4], [5], [6] are fast, they usually generate CAs of large size. Meta-heuristic solvers [7], [8], [9], [10] can find smaller CAs, but with the price of much more time consumption.

This work describes an efficient meta-heuristic solver for CCAG named *FastCA* [11], which can provide good solutions within much shorter time compared to state-of-the-art CCAG solvers. It utilizes a new lightweight score computation method that is much faster than the previous methods. *FastCA* employs three search modes, including the gradient mode, the greedy mode, and the random mode. Note that this is the first time a gradient step (which needs to query the scores of many operations) is implemented in meta-heuristics solvers, thanks to the lightweight score computation method. Experimental results on a broad range of real-world benchmarks and synthetic benchmarks show that *FastCA* is much faster and obtains better solutions than previous heuristics.

II. Overview

The framework of the *FastCA* solver is presented in Algorithm 1. Given an input system under test and the covering strength t, the *FastCA* solver outputs the best found covering array within the cutoff time. It is based on the local search methodology [12]. As a first step, the solver calls an initialization function to construct a CA (Line 1), which serves as a starting point for the search procedure. The initialization is provided by a well-known greedy solver *ACTS* [5]. Then, *FastCA* iteratively refines the current solution until the cutoff time is reached (Line 3–12). In the course of the procedure, whenever the solver successfully finds a CA α (i.e., all interactions are covered), it randomly removes one test case from α and continues to search for CA of smaller sizes (Line 4–6). The search procedure is guided with the score of operations, which indicates the reduction of the number of uncovered interactions. It works among three different modes.

Algorithm 1: The *FastCA* algorithm

```
Input: System under test, covering strength $t$
Output: CA $\alpha^*$
1  $\alpha \leftarrow$ Initialization();
2  $\alpha^* \leftarrow \alpha$;
3  while The termination criterion is not met do
4      if there is no uncovered interaction then
5          $\alpha^* \leftarrow \alpha$;
6          Remove one row (test case) from $\alpha$;
       // Its effectiveness relies on the
          lightweight score computation
7      $\alpha \leftarrow$ Gradient_Mode();
8      if Gradient_Mode is not successful then
9          if With probability $p$ then
10             $\alpha \leftarrow$ Random_Mode();
11         else
12             $\alpha \leftarrow$ Greedy_Mode();
13 return $\alpha^*$;
```

The Gradient Mode — This is a particular mode of *FastCA*. In this mode, it collects *all* candidate operations which will cover at least one uncovered interaction. The operation with the highest score is chosen. If the score is positive, it is applied to the partial-CA α. Otherwise, the search procedure switches to the other modes. The gradient mode is the major mode of the search procedure. This mode intensively reduces the number of uncovered interactions in order to obtain high-quality solutions. More details are described in Section III-C.

The Greedy Mode — Unlike the gradient mode, this mode only considers the operations related to one uncovered interaction. It first randomly selects an uncovered interaction and calculates the score of operations which will at least cover the chosen interaction. The operation of the highest score is applied to the partial-CA α, even when its score is negative.

The Random Mode — This mode is proposed for better exploring the search space and diversifying the search. It replaces one randomly chosen test case of the partial-CA to cover a pre-selected uncovered interaction.

When the gradient mode is not successful, that is, there is no candidate operation of positive score, the *FastCA* solver works in the greedy mode with a probability p, otherwise (with a probability $1-p$) it works in the random mode.

III. Main ideas and Implementation

This section introduces two main ideas of *FastCA*, including the lightweight score computation and the gradient mode. We also describe important implementation details here.

A. Bottlenecks of Meta-heuristic Solvers

Our analysis shows that a bottleneck that hinders the performance of meta-heuristic solvers for CCAG is the high time complexity of the score computation methods. For an operation $op(tc[p] \leftarrow v)$ that modifies the test case tc by assigning v to the option p, a commonly used method for computing $score(op)$ is to check each interaction τ involving (p, v) and measures the change in the number of uncovered interactions. A simple analysis shows that there are $\binom{k-1}{t-1} \times 2$ interactions to be checked for computing the score of only one operation, where k is the number of options and t is the required covering strength.

This process is of high complexity and is very time-consuming in practice. For instance, considering the *Apache HTTP Server* instance with $k = 172$ options, and supposing the covering strength $t = 3$, we need to check 29 070 interactions for computing an operation's score. Many heuristic strategies require computing scores for numbers of operations in each step. Experimental results on *TCA* for solving real-world benchmarks show that the time consumption of score computation occupies 34.5%, 77.1%, and 77.4% of the time budget for 2, 3, and 4 way covering strength respectively [11].

B. Lightweight Score Computation

An important fact is that only uncovered interactions and 1-covered interactions (interactions covered exactly once) are related to the score of an operation. The number of these interactions are expected to be small. By surveying a state-of-the-art solver *TCA*, we found that the number of uncovered interactions and related 1-covered interactions only occupy 0.01% and 0.11% of the $\binom{k-1}{t-1} \times 2$ interactions on real-world instances, respectively. Based on this observation, the score can be computed in a lightweight manner as follows.

Lightweight_score_computation(operation** op**):

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

To calculate $make(op)$, we use an array to maintain the uncovered interactions during the search procedure and check whether each of them is covered after the op operation. Since the number of uncovered interactions is rather small, $make(op)$ can be calculated efficiently. The $break(op)$ can also be calculated with low time complexity. We use a 3-dimension array to store the 1-covered interactions indexed by test case tc and option p. In this way, for an operation $op(tc[p] \leftarrow v)$, $break(op)$ is equal to the number of interactions stored in the array of option p in tc. Therefore, the time complexity for calculating $break(op)$ is $O(1)$. To efficiently maintain (i.e., push and pop) these data structures, we also track the positions of each uncovered and 1-covered interaction in the arrays.

C. The Gradient Mode

The main procedure of meta-heuristics algorithms is searching for CAs of iteratively smaller size. The effectiveness of this search procedure relies on the combination of intensification and diversification strategies which are guided with the scores of operations [12], [13], [14]. The state-of-the-art solver *TCA* interleaves between the greedy mode (intensification) and the random mode (diversification) [10]. However, the greedy mode only considers operations involving one selected uncovered interaction. Therefore, there is a considerable probability that the score of the chosen operation is negative, even when there are other operations of positive score.

Thanks to the lightweight score computation method, some algorithmic strategies that are not affordable previously now can be implemented with affordable time consumption. Instead of considering only one uncovered interaction, the gradient mode of *FastCA* searches for good operations in a much larger space (Section II). Whenever operations of positive score exist, the gradient mode leads to a reduction of the number of uncovered interaction. Although the gradient mode involves much more score computations than the greedy mode, the enhancement of the search outweighs the time consumption if the lightweight score computation method was used.

D. Constraint Support

To generate a valid covering array, the constraints on the permissible combination of values of options must be handled correctly. In *FastCA*, the constraints are encoded as the conjunctive normal form of the first-order logic formula. For example, the clause $p_1 \neq a \vee p_2 = b$ means that if the value of option p_1 is set to a, then p_2 must be set to b. In the initialization, we use the greedy solver *ACTS* [5] to generate a covering array subjecting to all constraints. After the initialization, *FastCA* prohibits operations which violate the constraints, so the validity of the partial-CA is guaranteed during the entire search procedure. Noting that since each test case is a complete assignment to all options, we do not need any constraint solving in the course of the search procedure. Using the concrete value of each option, it is straightforward to check whether an operation violates the constraints.

E. Possible Extension

For industrial or safety-critical systems, it usually requires higher covering strength testing for subsets of components, while keeping relatively lower covering strength for other components. In this case, variable strength covering arrays may be more effective and efficient [15]. Currently, *FastCA* only supports generating CAs with uniform covering strength for all options. But it should be easily extended to support variable covering strength by maintaining independent data structures for score computation of each covering strength.

IV. Evaluation

The performance of *FastCA* is evaluated on real-world and synthetic benchmarks. Among them are 26 industrial instances, including 6 popular instances SPINV, SPINS, GCC, Apache HTTP Server, Bugzilla, and TCAS [3], as well as 20 instances from IBM covering a broad range of applications including banking systems, telecommunications, healthcare, etc [16]. The other 30 instances are synthetic instances generated based on the characteristics of the industrial instances [8]. The covering strength is set to 3, which is well studied by previous literature [8], [9], [10].

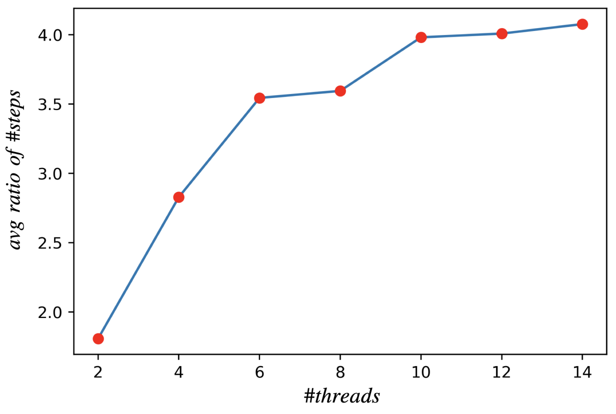

Fig. 1. Statistical results comparing search steps for *FastCA* with different numbers of threads in 100s. The horizontal axis is the number of threads and the vertical axis is the averaged ratio of the number of search steps using multiple and single threads

While greedy solvers are faster, meta-heuristic solvers can produce better solutions. *FastCA* is developed with the expectation that it combines the advantage of greedy solvers and meta-heuristic solvers. Therefore, we evaluate *FastCA* against state-of-the-art solvers in terms of both the size of CAs and the time consumption. The competitor solvers include three meta-heuristic solvers *TCA* [10], *HHSA* [9] and *CASA* [8], as well as a greedy solver *ACTS* [5]. We report the smallest size (‘min’) and the averaged size (‘avg’) found by the respective algorithm over 10 runs, with the cutoff time set to 1000 seconds for each run. The running time (‘time’) is averaged over 10 runs for finding the optimized CAs. To further illustrate the efficiency of *FastCA*, we also report the results of *FastCA* in a much shorter time budget of 100 seconds.

The experimental results on the industrial instances are presented in Table I, which is taken from [11]. As it is shown, under the same time budget of 1000 seconds, *FastCA* outperforms all its competitors significantly and achieves the smallest covering arrays (cover all interactions) on most instances. It is impressive that even within only one-tenth of the time budget, *FastCA* still stands out to be the best solver. More experimental results are available on the GitHub repository.

A. Speedup of Parallel Execution

To solve difficult instances more efficiently, parallelism is supported in *FastCA*. In the course of the search procedure, the score computation of operations, as well as the applying of the selected operation, can be executed in parallel. We run the multi-thread version of *FastCA*, which parallelizes both score computation and operation applying, on three difficult instances out of six industrial instances, including Apache, GCC, and SPINV. On the other three easy instances (Bugzilla, SPINS, and TCAS), the benefit brought by multi-threads may not cover the cost of inter-thread communication.

Figure 1 presents the speedups in terms of step numbers when using different numbers of threads. It is shown that the

TABLE I

COMPARING *FastCA* AGAINST STATE-OF-THE-ART COMPETITORS FOR 3-WAY CCAG ON THE INDUSTRIAL BENCHMARKS. THE RUNNING TIME IS MEASURED IN CPU SECOND AND AVERAGED OVER 10 RUNS. THIS TABLE IS TAKEN FROM [11]

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)	

performance of *FastCA* is further improved for solving difficult instances with the support of multi-threads.

V. CONCLUSION

We present a new solver *FastCA* for CCAG, which can provide high quality solutions within much shorter time compared to state-of-the-art CCAG solvers. Based on the lightweight score computation method, *FastCA* works efficiently in the new gradient mode to explore the search space more widely and intensively. Experimental results show that *FastCA* outperforms all state-of-the-art CCAG solvers significantly on the metrics of the CA size and the run time. It is able to achieve better solutions even within a much shorter time budget.

The testing benchmarks and the detailed experimental results are available at https://github.com/jkunlin/FastCATool.

VI. ACKNOWLEDGEMENT

The authors would like to thank Prof. Jian Zhang for his valuable discussion and comments on this paper. Shaowei Cai was supported by Youth Innovation Promotion Association of Chinese Academy of Sciences [No. 2017150] and Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

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

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

[3] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of ISSTA 2007*, 2007, pp. 129–139.

[4] 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 Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[5] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *ICST 2013*, 2013, pp. 242–251.

[6] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proceedings of ASE 2016*, 2016, pp. 614–624.

[7] P. Galinier, S. Kpodjedo, and G. Antoniol, "A penalty-based tabu search for constrained covering arrays," in *Proceedings of GECCO 2017*, 2017, pp. 1288–1294.

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

[9] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proceedings of ICSE 2015*, 2015, pp. 540–550.

[10] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation," in *Proceedings of ASE 2015*, 2015, pp. 494–505.

[11] J. Lin, S. Cai, C. Luo, Q. Lin, and H. Zhang, "Towards more efficient meta-heuristic algorithms for combinatorial test generation," in *Proceedings of ESEC/FSE 2019*, 2019, pp. 212–222.

[12] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[13] C. Luo, S. Cai, K. Su, and W. Huang, "CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability," *Artificial Intelligence*, vol. 243, pp. 26–44, 2017.

[14] P. Merz and B. Freisleben, "Fitness landscapes, memetic algorithms, and greedy operators for graph bipartitioning," *Evol. Comput.*, vol. 8, no. 1, pp. 61–91, 2000.

[15] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of ICSE 2003*, 2003, pp. 38–48.

[16] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proceedings of SEW 2006*, 2006, pp. 153–158.