A Constrained Covering Array Generator using Adaptive Penalty based Parallel Tabu Search

Yan Wang*†, Huayao Wu*, Xintao Niu*, Changhai Nie* and Jiaxi Xu†
*State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China
†School of Information and Technology
Nanjing Xiaozhuang University, Nanjing, China

***Abstract*—Since Combinatorial Testing (CT) was first proposed in 1980s, there have been more than 50 algorithms or tools published for Covering Array Generation (CAG). This paper introduces the APPTS tool, which uses a novel Adaptive Penalty based Parallel Tabu Search (APPTS) algorithm to generate as small-sized Constrained Covering Arrays (CCA) as possible. Instead of simply reusing constraint solver or forbidden tuple-based techniques to handle constraints, APPTS incorporates a penalty term into the fitness function to handle the constrained search space, and employs an adaptive penalty mechanism to dynamically adjust the penalty weight in different search phases. Our experiments demonstrate that the APPTS tool performs well in terms of covering array size.**

***Index Terms*—Constrained covering array, Combinatorial testing, Tabu search, Adaptive penalty, Parallelization**

1. Introduction

In recent years, the functions of software systems are becoming more and more powerful and complex. In order to support multi platforms and multi scenarios, most software systems are designed as configurable systems. However, the sets of all possible inputs for these systems are often too large to be tested exhaustively. Combinatorial Testing (CT) is a popular method for sampling test cases from the potentially enormous input space of modern large and complex software systems [1]. CT uses a mathematical object, i.e., a *t-way* Covering Array (CA), or a Constrained Covering Array (CCA) if there are constraints between parameter values, as the test suite, aiming to cover all possible value combinations between any *t* parameters.

Assume that the behavior of a Software Under Test (SUT) is impacted by k parameters, and each parameter can take a_i discrete values ($1 \leq i \leq k$). A t-way CCA with respect to a set of constraints is an $n \times k$ array that satisfies: (1) each row of the array is a valid test case, which contains k test values, one for each test parameter; (2) the array covers every valid t-way combination. That is, for each valid t-tuple of parameter values $(v_{i_1}, \ldots, v_{i_t})$, where some t parameters are assigned to fixed values ($1 \leq t \leq k$), there exists at least one test in the array that contains the tuple. The value of t indicates the coverage strength, and n is the size of CCA. When there is no constraint in the system, a CCA then becomes a CA.

One of the fundamental activities of CT is to generate a covering array with as small size as possible. Unfortunately, the minimum array size of any given test model remains unknown [2], and researchers have tried various methods to solve it. In this paper, we introduce the APPTS tool, which uses a novel Adaptive Penalty based Parallel Tabu Search algorithm to generate constrained covering arrays.

APPTS[1] introduces a penalty term into the fitness function for constraints handling, which differs from previous algorithms that mainly rely on constraints solver or minimum forbidden tuples [3] [4] [5], APPTS further employs an adaptive mechanism to determine the penalty weight incorporated into the fitness function, so that the search behavior of Tabu Search (TS) can be dynamically adjusted in different search phases; A new combination of neighbourhood function and tabu strategy is also proposed, hoping to further enhance the search ability of the conventional TS-based algorithms; In addition, APPTS uses a parallelization mechanism to speed up the computation of fitness values of candidate solutions, which could make APPTS search more spaces under a given running time budget.

The rest of this paper is organized as follows. Section 2 describes the APPTS tool. Section 3 empirically evaluates the effectiveness and efficiency of the tool. And Section 4 concludes this paper and gives the future work.

2. APPTS Tool

In this section, we first introduce the input of the APPTS tool (Section 2.1) and the method used for constraint conversion (Section 2.2). After that, we explain the complete covering array generation process (Section 2.3). Finally, we show how to use the tool (Section 2.4).

This work was supported in part by National Key Research and Development Program of China (2018YFB1003800), National Natural Science Foundation of China (61902174, 62072226), and Natural Science Foundation of Jiangsu Province (BK20190291).

1. https://github.com/GIST-NJU/APPTS

2.1. Input

The input of APPTS tool is a CTWedge format file[2] in which a tester specifies the name of the test model, parameters, parameter values and constraints between parameters (if has). Fig. 1 shows an example of a simple test model.

```
Model MCAC_test
Parameters:
Par0: {PAR0_0,PAR0_1,PAR0_2}
Par1: {PAR1_0,PAR1_1,PAR1_2}
Par2 : Boolean
Par3: {PAR3_0,PAR3_1,PAR3_2,PAR3_3}
Par4: {PAR4_0,PAR4_1,PAR4_2,PAR4_3}
Par5: {PAR5_0,PAR5_1,PAR5_2,PAR5_3}
Par6: {PAR6_0,PAR6_1,PAR6_2}
Par7 : Boolean

Constraints:
# (Par4 != PAR4_2 OR Par5 != PAR5_3) #
# ((Par0 != PAR0_0 => (not (Par7 = true))) <=> (not (Par4 != PAR4_1))) #
# (Par7 = false AND Par0 != PAR0_2)#
```

Figure 1. An example of test model

From Fig. 1, we can see that the test model is named 'MCAC_test'. It has 8 parameters, where Para2 and Para7 are boolean and others are enumerative. There are 3 constraints in logical formula with logical (including and, or, not, implication, equivalence), equal and unequal operators.

2.2. Constraint Conversion

In APPTS, the invalid solutions are allowed in the search process, and the fitness function used is defined as [6]:

$$fitness(\alpha) = U(\alpha) + \lambda \times W(\alpha), \quad (1)$$

where α is the current solution in the search process and $U(\alpha)$ is the number of valid t-way combinations that are not covered by α, $W(\alpha)$ is the number of Minimum Forbidden Tuples (MFT) that are involved in α, and λ ($\lambda \geq 0$) is the penalty weight. APPTS adopts an adaptive mechanism to determine the penalty weight during the search. More details about the adaptive mechanism can be seen in [6].

Here, an MFT is a forbidden tuple of minimum size that covers no other Forbidden Fuples (FT), which indicates that a particular value assignment of some parameters is not allowed. However, the constraints in the input file are represented as logical formula. So the constraints provided by the input file must be converted to MFTs. To this end, the following steps are used.

- Step1: covert the constraints in logical formula to Conjunctive Normal Form (CNF) by a converter in CTWedge[3]. For example, the logical formula (Para1 = value1_1 => Para2 = value2_1) is converted to the CNF (Para1 != value1_1 OR Para2 = value2_1).
- Step2: covert CNFs to FTs by transforming equal operator into unequal operator. For example, suppose that parameter 'Para2' has three values named 'value2_1', 'value2_2' and 'value2_3', then the CNF (Para1 != value1_1 OR Para2 = value2_1) is coverted to two FTs (Para1 != value1_1 OR Para2 != value2_2) and (Para1 != value1_1 OR Para2 != value2_3)
- Step3: derive MFTs from FTs by the method described in [7]. For example, given two FTs (Para1 != value1_1 OR Para2 != value2_2) and (Para1 != value1_1), then one MFT (Para1 != value1_1) will be derived, because (Para1 != value1_1 OR Para2 != value2_2) is covered by (Para1 != value1_1) so it is not an MFT.

For example, Fig. 2 gives the MFTs obtained by using the above steps on the constraints described in Fig. 1.

```
(Par4 != PAR4_0)
(Par4 != PAR4_2)
(Par4 != PAR4_3)
(Par7 != true)
(Par0 != PAR0_2)
```

Figure 2. MFTs of the constraints described in Fig. 1

2.3. Covering Array Generation Process

To generate a covering array, APPTS works in two layers: *outer search* and *inner search*.

The outer search determines the size of covering array that will be generated. At the beginning, this algorithm applies a well-known greedy constructor, ACTS [8], for initialization. Next, at each iteration, the algorithm will remove a test case from the current solution at random, and invokes the inner search to try to transform it to a covering array. The above process will repeat until the inner search cannot find such a covering array, or the running time of the algorithm exceeds the cutoff time.

The inner search tries to find a particular covering array of the given size. The core generation algorithm is TS, which iteratively modifies one unit of the current solution, until it becomes a covering array or the cutoff time reaches. Different from other TS based algorithms [3] [4] [5], APPTS extends the search space to invalid solutions, uses a novel neighborhood function and tabu strategy, and calculates the fitness of the current solution in parallel mode [6].

Furthermore, in order to make the APPTS tool more practically applicable, and also balance covering array sizes and generation times, the algorithm will terminate as long as the solutions constructed in the recent iterations have not been improved (with respect to the number of uncovered t-way combinations), In such a scenario, the algorithm is unlikely to generate a covering array with the given size.

2.4. Usage of the Tool

To obtain covering arrays, the tester should provide the test model file (as shown in Fig. 1), coverage strength (2-6), and cutoff time (measured in seconds). Here cutoff time is

2. https://fmselab.github.io/ct-competition/
3. https://github.com/fmselab/ctwedge

optional, and the default value is 3600 seconds. If Java 11 or higher version is installed on the machine, the tester can run the command: 'java -jar appts.jar strength modelfileName cutoffTime'. If Docker is available, the tester can run the command: 'docker-compose up'. It will invoke the 'docker-compose.yml' file, where the coverage strength, model file name and cutoff time are specified.

The covering array generated by APPTS will be saved into a CSV file. The first line is the names of the parameters and the following lines are the test cases. Fig. 3 shows an output of 2-way CCA of the test model in Fig. 1. We can check that the CCA contains all valid 2-way combinations and all test cases in the CCA are also valid.

```
Par0,Par1,Par2,Par3,Par4,Par5,Par6,Par7
PAR0_1,PAR1_1,false,PAR3_0,PAR4_1,PAR5_1,PAR6_1,false
PAR0_0,PAR1_2,true,PAR3_1,PAR4_1,PAR5_2,PAR6_2,false
PAR0_1,PAR1_0,true,PAR3_2,PAR4_1,PAR5_3,PAR6_0,false
PAR0_0,PAR1_1,false,PAR3_3,PAR4_1,PAR5_0,PAR6_0,false
PAR0_1,PAR1_2,true,PAR3_0,PAR4_1,PAR5_0,PAR6_2,false
PAR0_1,PAR1_0,false,PAR3_1,PAR4_1,PAR5_0,PAR6_1,false
PAR0_0,PAR1_1,false,PAR3_2,PAR4_1,PAR5_0,PAR6_2,false
PAR0_0,PAR1_1,true,PAR3_1,PAR4_1,PAR5_1,PAR6_0,false
PAR0_0,PAR1_2,false,PAR3_2,PAR4_1,PAR5_1,PAR6_1,false
PAR0_1,PAR1_0,true,PAR3_3,PAR4_1,PAR5_1,PAR6_2,false
PAR0_0,PAR1_0,false,PAR3_0,PAR4_1,PAR5_2,PAR6_0,false
PAR0_1,PAR1_1,true,PAR3_2,PAR4_1,PAR5_2,PAR6_1,false
PAR0_1,PAR1_2,false,PAR3_3,PAR4_1,PAR5_2,PAR6_0,false
PAR0_0,PAR1_1,false,PAR3_0,PAR4_1,PAR5_3,PAR6_1,false
PAR0_0,PAR1_2,true,PAR3_1,PAR4_1,PAR5_3,PAR6_2,false
PAR0_1,PAR1_0,false,PAR3_3,PAR4_1,PAR5_3,PAR6_1,false
```

Figure 3. The output of 2-way CCA of test model described in Fig. 1

3. Experiments

This section presents the experiments performed to evaluate the effectiveness and efficiency of the APPTS tool.

3.1. Experimental Setup

In [6], we compared APPTS with seven state-of-the-art CCAG algorithms: CASA [9], HHSA-H [10], Calot [11], CATS [5], SQ-CHiP [12], FastCA [3] and WCA [4] on 55 benchmarks (cutoff time = 1000 seconds). The experimental results show that APPTS outperforms existing algorithms in terms of covering array size, and found 4 and 18 new upper bounds on 2-way and 3-way CCA sizes, respectively. All experimental data can be obtained from the website[4].

In this study, we further used APPTS to generate 2-way covering arrays on 35 new test models. Specifically, 25 test models come from the competition website[5], and the remaining 10 models were generated by using the benchmark generator of the competition[6]. These models are listed in the upper and lower half of Table 1, respectively. Table 1 also summarizes the number of parameters and values of each parameter, as well as the number of constraints. For example, if a test model has 10 parameter, 4 out of which have 3 values and 6 out of which have 5 values, then the test model is denoted by $3^4 5^6$. The constraints here are represented by logical formula as in the CTWedge format file.

All experiments were carried out on a machine with 8-core Intel Xeon E5-2640 2.6GHz CPU and 40GB RAM, running Ubuntu 15.1 operating system. To account the randomness of the algorithm, the execution is repeated 10 times on each subject with cutoff time = 300 seconds. We report the smallest size ('min_size') and the average size ('avg_size') over 10 runs. In addition, on each subject, we report the running time ('avg_time') required for finding the smallest covering arrays. If the tool failed to generate a covering array during all 10 runs, we report the size and time as '–'. For comparison, we also give the sizes of covering arrays generated and the average times consumed by ACTS[7].

TABLE 1. BENCHMARKS OF THE EXPERIMENT

Benchmark	Parameters	Constraints
BOOLC_0	2^{11}	45
BOOLC_1	2^{7}	2
BOOLC_2	2^{2}	63
BOOLC_3	2^{5}	9
BOOLC_4	2^{16}	58
MCAC_0	$2^{8}18^{2}32^{1}34^{1}42^{1}48^{1}$	93
MCAC_1	$2^{5}3^{1}12^{1}21^{1}25^{1}38^{1}39^{1}40^{1}43^{1}46^{1}49^{1}$	89
MCAC_2	$2^{13}4^{2}5^{1}16^{1}27^{1}32^{1}45^{1}$	15
MCAC_3	$2^{7}3^{1}11^{1}18^{1}20^{1}21^{1}22^{1}26^{2}36^{1}40^{1}46^{1}50^{1}$	63
MCAC_4	$2^{11}9^{1}13^{1}15^{1}21^{1}30^{1}31^{1}38^{1}46^{1}$	81
MCA_0	$2^{8}6^{1}7^{1}22^{1}33^{1}36^{1}38^{2}39^{1}44^{1}45^{1}49^{1}$	0
MCA_1	$2^{3}17^{1}24^{1}$	0
MCA_2	$2^{5}8^{1}14^{1}37^{1}$	0
MCA_3	$2^{11}3^{1}4^{1}9^{1}16^{1}19^{1}20^{1}31^{1}44^{1}$	0
MCA_4	$2^{47}1^{10}1^{5}1^{18}1^{27}1^{33}1^{43}1$	0
UNIFORM_ALL_0	8^{19}	0
UNIFORM_ALL_1	14^{16}	0
UNIFORM_ALL_2	5^{17}	0
UNIFORM_ALL_3	16^{7}	0
UNIFORM_ALL_4	16^{14}	0
UNIFORM_BOOLEAN_0	2^{15}	0
UNIFORM_BOOLEAN_1	2^{14}	0
UNIFORM_BOOLEAN_2	2^{14}	0
UNIFORM_BOOLEAN_3	2^{19}	0
UNIFORM_BOOLEAN_4	2^{8}	0
BOOLC_5	2^{18}	8
BOOLC_6	2^{20}	11
MCAC_5	$2^{7}8^{1}9^{1}10^{1}13^{1}16^{1}20^{1}22^{1}25^{3}34^{1}38^{1}$	3
MCAC_6	$2^{7}7^{1}8^{1}12^{1}13^{1}29^{2}46^{1}$	2
MCA_5	$2^{12}20^{1}21^{1}24^{2}34^{1}39^{1}49^{2}$	0
MCA_6	$2^{6}18^{1}39^{1}41^{1}$	0
UNIFORM_ALL_5	2^{17}	0
UNIFORM_ALL_6	16^{17}	0
UNIFORM_BOOLEAN_5	2^{8}	0
UNIFORM_BOOLEAN_6	2^{12}	0

3.2. Results

Table 2 gives the results obtained. We can see that there are 9 subjects for which our tool and ACTS can not generate covering arrays. Specifically, the constraints of 7 subjects (BOOLC_0, BOOLC_2, BOOLC_3, BOOLC_4, MCAC_1, MCAC_3 and MCAC_4) are contradictory with each other and thereby no constraints-satisfying solutions can be found. The constraints of MCAC_0 are too complex and both tools can not parse them. We notice that the longest constraint contains 1024 variables and the number of all variables is

4. https://github.com/GIST-NJU/APPTS

5. https://github.com/fmselab/ct-competition/raw/gh-pages/examples/CTWedge.zip

6. https://github.com/fmselab/CIT_Benchmark_Generator

7. https://csrc.nist.gov/acts

TABLE 2. A 2-WAY COVERING ARRAYS GENERATED BY APPTS AND ACTS

Benchmark	APPTS			ACTS	
	min_size	avg_size	avg_time	size	avg_time
BOOLC_0	-	-	-	-	-
BOOLC_1	6	6	0.23	8	0.14
BOOLC_2	-	-	-	-	-
BOOLC_3	-	-	-	-	-
BOOLC_4	-	-	-	-	-
MCAC_0	-	-	-	-	-
MCAC_1	-	-	-	-	-
MCAC_2	-	-	-	-	-
MCAC_3	-	-	-	-	-
MCAC_4	-	-	-	-	-
MCA_0	2214	2223.9	3.09	2263	0.10
MCA_1	408	408	0.01	408	0.01
MCA_2	518	518	0.01	518	0.01
MCA_3	1364	1364	0.03	1366	0.02
MCA_4	1419	1419	0.04	1424	0.02
UNIFORM_ALL_0	116	116.4	16.49	145	0.02
UNIFORM_ALL_1	327	328.0	55.78	371	0.03
UNIFORM_ALL_2	44	44.9	1.72	53	0.01
UNIFORM_ALL_3	303	305.7	9.18	338	0.01
UNIFORM_ALL_4	402	404.0	66.81	457	0.03
UNIFORM_BOOLEAN_0	7	7	0.01	10	0.01
UNIFORM_BOOLEAN_1	7	7	0.01	10	0.01
UNIFORM_BOOLEAN_2	7	7	0.02	10	0.01
UNIFORM_BOOLEAN_3	8	8	0.01	12	0.01
UNIFORM_BOOLEAN_4	6	6	0.01	8	0.01
BOOLC_5	8	8	0.30	11	0.25
BOOLC_6	8	8	9.14	14	0.81
MCAC_5	1292	1292	10.39	1308	10.16
MCAC_6	1334	1334	6.38	1336	6.08
MCA_5	2401	2401	0.15	2408	0.05
MCA_6	1599	1599	0.02	1599	0.02
UNIFORM_ALL_5	8	8	0.01	10	0.01
UNIFORM_ALL_6	429	431.5	69.85	479	0.04
UNIFORM_BOOLEAN_5	6	6	0.01	8	0.01
UNIFORM_BOOLEAN_6	7	7	0.01	10	0.01

up to 18403 in the subject. For MCAC_2, the two figures are 1024 and 3904 and the two tools can not generate any CCAs in the specified cutoff time. We note that such complex constraints in both MCAC_0 and MCAC_2 cannot be handled by ACTS, probably the most popular covering array generator in practice. Such complex constraints also differ from the mainstream benchmarks used in the current literature of covering array generation [3] [4] [5] [10] [11] [12].

Next, we discuss the results of the remaining 26 feasible subjects. From Table 2, compared to ACTS, APPTS can further decrease the sizes of covering arrays in 23 out of 26 sujects. The maximum difference of sizes between them is up to 55 (UNIFORM_ALL_4). With respective to efficiency, we can see there are 20 out of 26 subjects where the time difference (measured in seconds) between the two algorithms is no more than 3 seconds. Such a difference is practically insignificant [12]. For the other 6 subjects, APPTS may take at least 60 seconds longer than ACTS, but on the other hand, APPTS can generate much smaller covering arrays. We note that the main aim of APPTS (a search-based algorithm) is to generate as small covering arrays as possible. Compared to greedy algorithm-based generators (e.g., ACTS), such search-based algorithms can typically generate smaller covering arrays, but need to take longer time to execute [13] [14] [15].

4. Conclusion and Future Work

Constrained Covering Array Generation (CCAG) is of fundamental importance for the application of CT, and has attracted extensive attentions in the literature. In this work, we introduce the APPTS tool to solve the CCAG problem. Our experimental results reveal that the APPTS tool can effectively generate small CAs/CCAs.

For some constraints representing with complex logical formula, our tool currently can not parse it. This is one of our future work. Additionally, we plan to further improve the speed of our tool.

References

[1] C. H. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[2] L. Kampel and D. E. Simos, "A survey on the state of the art of complexity problems for covering arrays," *Theor. Comput. Sci.*, vol. 800, pp. 107–124, 2019.

[3] J. K. Lin, S. W. Cai, C. Luo, Q. W. Lin, and H. Y. Zhang, "Towards more efficient meta-heuristic algorithms for combinatorial test generation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 212–222.

[4] Y. J. Fu, Z. D. Lei, S. W. Cai, J. K. Lin, and H. R. Wang, "Wca: A weighting local search for constrainted combinatorial test optimization," *Information and Software Technology*, 2020.

[5] P. Galinier, S. Kpodjedo, and G. Antoniol, "A penalty-based tabu search for constrained covering arrays," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 1288–1294.

[6] Y. Wang, H. Wu, X. Niu, C. Nie, and J. Xu, "An adaptive penalty based parallel tabu search for constrained covering array generation," *Inf. Softw. Technol.*, vol. 143, p. 106768, 2022.

[7] L. B. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Constraint handling in combinatorial test generation using forbidden tuples," in *Eighth IEEE International Conference on Software Testing, Verification and Validation*, 2015, pp. 1–9.

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

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

[10] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *37th IEEE/ACM International Conference on Software Engineering*. IEEE Computer Society, 2015, pp. 540–550.

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

[12] H. Mercan, C. Yilmaz, and K. Kaya, "Chip: A configurable hybrid parallel covering array constructor," *IEEE Trans. Software Eng.*, vol. 45, no. 12, pp. 1270–1291, 2019.

[13] J. K. Lin, C. Luo, S. W. Cai, K. L. Su, D. Hao, and L. Zhang, "TCA: an efficient two-mode meta-heuristic algorithm for combinatorial test generation (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 494–505.

[14] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.

[15] J. Torres-Jimenez and J. C. Perez-Torres, "A greedy algorithm to construct covering arrays using a graph representation," *Inf. Sci.*, vol. 477, pp. 234–245, 2019.