Empirically Identifying the Best Genetic Algorithm for Covering Array Generation

Liang Yalan[1], Changhai Nie[1], Jonathan M. Kauffman[2],
Gregory M. Kapfhammer[2], Hareton Leung[3]

[1]Department of Computer Science and Technology, Nanjing University
[2]Department of Computer Science, Allegheny College
[3]Department of Computing, Hong Kong Polytechnic University
lyl-49@163.com, changhainie@nju.edu.cn, {kauffmj, gkapfham}@allegheny.edu,
cshleung@inet.polyu.edu.hk

1 Introduction

With their many interacting parameters, modern software systems are highly configurable. Combinatorial testing is a widely used and practical technique that can detect the failures triggered by the parameters and their interactions. One of the key challenges in combinatorial testing is covering array generation, an often expensive process that is not always amenable to automation with greedy methods. The 2-way covering array is the most common and can be defined as follows [1]:

Definition (2-way covering array): If an $N \times n$ array has the following properties: (1) each column i ($1 \leq i \leq n$) contains only elements from the set V_i with $a_i = |V_i|$ and (2) the rows of each $N \times 2$ sub-array cover all 2-tuples of values from the 2 columns at least once, then it is called a *2−way covering array (CA)*.

Table 1. Covering array of the SUT.

pa_1	pa_2	pa_3	pa_4
0	0	0	0
1	0	2	1
2	1	2	0
2	0	1	2
1	1	0	2
0	1	1	1
2	2	0	1
1	2	1	0
0	2	2	2

During combinatorial testing, each row of a CA represents a test case. In order to more clearly describe the concept of a covering array, we develop a concrete example. Suppose there are 4 parameters (pa_1, pa_2, pa_3, and pa_4) in a system under test (SUT), each with 3 values (0, 1, 2). If we want to cover all 54 pair-wise interactions between every 2 parameters in the SUT, then only 9 test cases are needed in Table 1's covering array. In real-world testing efforts, covering arrays can save testing time while still detecting many failures.

Researchers have proposed many techniques to generate covering arrays. As one of the evolutionary search methods, the genetic algorithm often has been effectively applied to solve many complex optimization problems in this and other fields. However, the performance of a genetic algorithm is not always stable and thus significantly impacted by its configurable parameters. Previous studies [2][3][4] have not considered either the exploration of the genetic algorithm's optimal configuration or ways to improve its performance for covering array generation.

In order to close this knowledge gap, we designed three classes of experiments (i.e., a pair-wise, base choice, and hill climbing experiment) to systemically examine the impacts of and interactions between the genetic algorithm's five configurable parameters (i.e., population size, number of generations, crossover probability, mutation probability, and genetic algorithm variant).

Overall, the goal of this paper is to answer the following two questions: (1) Is there an improved configuration of a genetic algorithm for a particular pair-wise SUT? and (2) Is there a common improved configuration for all pair-wise SUTs? In confirmation of the prevailing wisdom about genetic algorithms, the empirical results

affirm the first question and refute the second one. Moreover, this paper provides a comprehensive overview of the empirical trade-offs associated with a genetic algorithm's performance on covering array generation and highlights circumstances that lead to interesting and counter-intuitive results. Since our experiments use discrete values for the genetic algorithm's parameters, in future work we plan to employ another type of genetic algorithm to further improve these configurations in a continuous value space, as discussed in greater detail in Section 4.

2 Genetic Algorithm for Covering Array Generation

As one of the typical evolutionary search methods, the genetic algorithm is an effective technique that can search for the optimal solution for a wide variety of complex optimization problems [2][3][4]. This paper applies the genetic algorithm described in Figure 1 to the problem of 2-way covering array generation.

Input: The description of the SUT like 3^{13} (13 parameters in a SUT, each with 3 values), the specified variant of the genetic algorithm *VGA*, the population size *m*, the number of generations *G*, crossover probability P_c, and mutation probability P_m.
Output: The covering array generated by the specified variant of the genetic algorithm.
Step 1: Initialize the test suite *TG* that can be null or include some test cases. Initialize the set of pair-wise interactions *S* that need to be covered in the SUT. Randomly generate an initial population of *m* chromosomes corresponding to *m* candidate test cases for the SUT.
Step 2: Calculate the fitness of each chromosome in the current generation. The fitness value is the number of pair-wise interactions uncovered by *TG* in each chromosome, implying that the higher the fitness value, the better the chromosome.
Step 3: Crossover and mutate (P_c and P_m give the probability of using these operators) the chromosomes in the current generation and then select (selection strategies differ in the various *VGA*s) the ones possessing the good characteristics into the next generation of the population. Search for the satisfied chromosome $chro_x$ (all pairs in $chro_x$ are uncovered by the test cases in *TG*) in the new population by calculating the fitness of all chromosomes. If $chro_x$ exists, put it into *TG* and jump to Step 5; otherwise, repeat Steps 2 and 3 until the number of repetitions accumulates to *G* and then jump to Step 4.
Step 4: Select the chromosome $chro_y$ with the highest fitness value in the current generation and put $chro_y$ into *TG*.
Step 5: Delete the pair-wise interactions that have been covered by $chro_x$ or $chro_y$ from *S*. If *S* is null, then the test suite *TG* is the desired covering array; otherwise, go back to Step 3.

Figure 1. Description of the genetic algorithm for covering array generation.

Since the goal of this experiment is to improve the performance of a genetic algorithm, we extract the five most important parameters and discuss them in detail.

(1) Population size *m*

The population of a genetic algorithm consists of *m* chromosomes. In general, if *m* is too small, then the algorithm may prematurely converge. However, if *m* is too large, then the increased computation time will delay the algorithm's completion.

(2) Number of generations *G*

The parameter *G* is the number of repetitions for which the GA's main loop will run. The algorithm will terminate quickly and not converge on a good solution when *G* is too small and the computation will incur a high time overhead when *G* is too large.

(3) Crossover probability P_c

Chromosome crossover is a mechanism for generating new individuals by combining the characteristics of two parents. If P_c is too large, the chromosomes in the population will crossover so frequently that the operator will break the existing chromosomes possessing good characteristics. If P_c is too small, then crossover will rarely be performed, often causing the algorithm not to converge on a good solution.

(4) Mutation probability P_m

The mutation operation can help to increase the diversity of the population. However, mutation will be performed rarely if P_m is too small, and the genetic algorithm will degrade into a random search if P_m is too large.

(5) Genetic algorithm variants *VGA*

The selection strategy and the fitness function of a genetic algorithm have a substantial impact on its performance. In order to investigate their effect, we alter these two operators in a systematic way to produce five variants of the genetic algorithm. The explanations of the original genetic algorithm, denoted GA, and these five varied algorithms are presented as follows:

GA: In GA, the covering array is constructed by generating the test cases one by one. In the process of evolution in the GA, the chromosome that has the highest fitness value will be selected into the next generation of the population immediately while the roulette-wheel operator, one of the typical selection strategies, will pick the others. The fitness of each chromosome is the number of uncovered pair-wise interactions that it contains. Both the crossover and the mutation strategies are multipoint which means that crossover and mutation will be performed on every gene of the chromosome with a certain probability.

GA-: The fitness function of GA- differs from the fitness function of GA in that it calculates the pair-wise interactions in every chromosome which have been covered by *TG*, changing the selection strategy from "select the superior and eliminate the inferior" to "select the inferior and eliminate the superior." The changed fitness function only operates when the chromosomes are selected into the next generation of the population. However, when the chosen test case is put into the covering array, the fitness function is still the same as GA.

GAr: In GAr, chromosomes are randomly selected for the population of the next generation.

GA climb: GA climb is a GA variant that includes elitism. In GA climb, the chromosome with the highest fitness in every generation is protected from crossover and mutation operations. It is only replaced by the chromosome with a higher fitness value in the next generation. Following the principle of elitism, GA climb guarantees that the good characteristics of the chromosome with the highest fitness in each generation will be continually improved instead of being destroyed by the crossover and mutation operations.

GA- climb: GA- climb is a GA- variant that includes the elitism from GA climb.

GAr climb: GAr climb is a GAr variant that includes the elitism from GA climb.

By studying the performance of GA-, GAr, GA climb, GA- climb, and GAr climb, we can determine whether the selection strategy and the fitness function have an impact on the efficiency and effectiveness of the genetic algorithm for CA generation.

3 Experimental Design and Results Analysis

3.1 Experimental Design

We chose 15 representative SUTs for the experiment in consideration of the following points: (1) the time consumed by the experiments must be reasonable, (2) the experiments must consider both the SUTs with large (e.g., 10^{11}) and small (e.g., 3^{13}) covering arrays, and (3) the SUTs should have parameters with different numbers of values. In order to identify an improved configuration, we will first consider the configuration that can make the algorithm generate the smallest-size covering array. But when more than one configuration meets this condition, we will select the one that causes the algorithm to exhibit the shortest execution time.

Before conducting the main experiment, we performed a preliminary study to choose a suitable set of candidate values for the five parameters. This exploratory investigation showed that the GA's execution time increased markedly without a commensurate increase in effectiveness when m was over 6000 or G was over 1000. Furthermore, since the GA was often unstable when m or G was less than 100 and the value of P_c and P_m did not have an apparent impact on the GA, we ultimately selected the candidate value set shown in Table 2 for the GA's five parameters.

Table 2. The value set of the genetic algorithm's five configurable parameters.

Parameters	Configurations
Algorithm (*VGA*)	GA, GA-, GAr, GA climb, GA- climb, GAr climb
Population size (*m*)	100, 2100, 4100, 6100
Number of generations (*G*)	100, 600, 1100
Crossover probability (P_c)	1.0, 0.8, 0.6, 0.4, 0.2
Mutation probability (P_m)	1.0, 0.8, 0.6, 0.4, 0.2

As stated in Section 1, we designed three classes of experiments to explore the improved configuration that can greatly reduce the cost of both performing future empirical studies and using GAs in practice. In addition, for ease of representation, we denote the values of the five parameters with ordered natural numbers. For instance, {0, 1, 2, 3} will be used to represent the four values of population size in Table 2. We designed the experiment to first use a pair-wise method to produce a 2-way covering array. Then, this array is used as input to the base choice and hill climbing techniques to create two final and improved covering arrays. Additional details about the experimental procedure are as follows:

(1) Pair-wise experiment

In order to create an improved configuration with less cost than exhaustively evaluating all GA configurations, suppose that there exist pair-wise interactions between the parameters of the genetic algorithm in Table 2. With the goal of covering these interactions, this procedure produces a 2-way covering array containing a set of 34 configurations. A comparison of the performance of the genetic algorithms in these configurations leads to the best configuration C_p among them.

(2) Base choice experiment

The base choice procedure is as follows [5]: first, it chooses C_p from the pair-wise experiment as its basic configuration. To generate a set of configurations, it changes the value of one parameter of the basic configuration while leaving the other four parameters unchanged. By choosing one of the other four parameters and repeating this process until all the values of the five parameters have been covered, this method produces a new set of configurations. This experiment allows for the separate exploration of the impact of each of the five parameters.

(3) Hill climbing experiment

As in base choice, the first step of the hill climbing experiment is to choose C_p as its basic configuration C_0. For example, if C_0 is <5, 0, 2, 1, 4> (which corresponds to <*VGA*=GAr climb, *m*=100, *G*=1100, P_c=0.8, P_m=0.2>), it creates six different configurations (<0, 0, 2, 1, 4>, <1, 0, 2, 1, 4>, <2, 0, 2, 1, 4>, <3, 0, 2, 1, 4>, <4, 0, 2, 1, 4>, and <5, 0, 2, 1, 4>) by changing the value of the first parameter *VGA* and leaving the other four parameters unchanged. By comparing the performance of the algorithms in these six configurations, the method gets the best configuration C_1. If

Table 3. The configurations of 15 SUTs improved by the three experiments.

SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time	SUT	*VGA*	*m*	*G*	P_c	P_m	CA Size	Run Time
4^{10}	GAr climb	100	100	0.2	0.2	28	0.234s	6^{30}	GA climb	100	1100	0.2	0.2	87	52.6s
3^{13}	GAr climb	100	1100	0.8	0.2	17	2.28s	10^{11}	GA climb	100	1100	0.8	0.2	154	19.8s
6^{10}	GA climb	6100	1100	0.2	0.2	58	402s	$7^6 6^7 5^6$	GAr climb	100	1100	0.8	0.2	82	23.5s
4^{20}	GAr climb	100	1100	0.8	0.2	35	10.1s	$8^2 7^2 6^2 5^2$	GA- climb	2100	600	0.8	0.6	70	277s
8^{10}	GA climb	2100	600	0.6	0.2	98	604s	$6^1 5^1 4^6 3^8 2^3$	GAr climb	4100	1100	0.8	0.4	36	568.1s
3^{20}	GA- climb	100	600	0.2	0.2	21	3.31s	6^4	GAr climb	100	100	0.6	0.2	41	0.03s
6^{20}	GA climb	100	1100	0.8	0.2	74	22.9s	$5^1 3^8 2^2$	GAr climb	100	100	0.8	0.2	20	0.43s
4^{30}	GAr climb	100	600	0.2	0.2	40	12.4s								

the algorithm performs best when the value of the first parameter is 2, it selects <2, 0, 2, 1, 4> as C_1. Then, based on C_1, configuration C_2 can be obtained by changing the value of the second parameter and comparing the efficiency and effectiveness values of the corresponding algorithms. Iteratively refining the genetic algorithm's configuration, this process repeats until it gets the best value of the fifth parameter and obtains C_5, the improved configuration generated by this experiment.

3.2 Experimental Results

After performing the experiments described in Section 3.1, we synthesized the results by choosing the best configuration for each SUT from the base choice and hill climbing experiments. We present these improved configurations in Table 3 and draw the following conclusions for this empirical study:

1) Across all of the SUTs, the genetic algorithm for covering array generation has different configurations and efficiency (i.e., run time) and effectiveness (i.e., covering array size) values. We can identify the improved configuration for covering array generation of each SUT in Table 3. However, these enhanced configurations are different from each other, indicating that, for the chosen SUTs, covering array generation does not have a common improved configuration.

2) By separately measuring the impact of the five configurable parameters in the base choice experiment and in consideration of the results in Table 3, we find that the configurable parameters have a differing impact on covering array generation:

VGA: The selection strategy and fitness function both have an impact on the efficiency and effectiveness of the GA. Counter to the prevailing intuition, pairing the GA- and random selection strategies with an elitist method (i.e., GA- climb and GAr climb) yields the best configuration for CA generation in ten out of the fifteen SUTs. Moreover, the *VGA*s of all the improved configurations in Table 3 use a climbing GA, showing that the efficiency and effectiveness of the elitist array generators are better than the other three genetic algorithm variants (i.e., GA, GA-, and GAr).

G: For all of the 15 SUTs, every possible value of G is evident in Table 3, with a concentration of values near a greater number of generations, confirming the intuition that a lengthier evolutionary process will improve covering array generation.

$\boldsymbol{P_m}$: The value of P_m is 0.2 for all but two of the SUTs in Table 3, suggesting that creating fewer mutated individuals leads to better covering arrays and motivating more study of the way in which smaller values for P_m will impact the GA.

$\boldsymbol{P_c}$ and $\boldsymbol{m}$: The improved configurations for each SUT have different values of P_c and m, which indicates that there is no common best value of P_c or m for the chosen SUTs. Thus, these two parameters do not have an obvious impact on CA generation.

4 Conclusions and Future Work

This paper focuses on the improvement of a genetic algorithm's efficiency and effectiveness for covering array generation. By three different experiments (pair-wise, base choice, and hill climbing experiment), we explored the differing impacts of the five parameters (i.e., population size, number of generations, crossover probability, mutation probability, and genetic algorithm variant) and their interactions. Meanwhile, we answered the questions raised at the beginning of the paper and obtained the improved configurations of 15 SUTs, as shown in Table 3. Interestingly,

the results indicate that counter-intuitive selection strategies and elitist methods are important components of a genetic algorithm for covering array generation.

Since we defined the value ranges of the five configurable parameters to be discrete, the improved configuration obtained may not be the optimal one for CA generation with a genetic algorithm. To address this problem and to further optimize the configurations, in future work we will use another genetic algorithm called GA′. This method will evolve the improved configurations with continuous value ranges for the five configurable parameters and thus be more likely to find the optimal 2-way covering array for each SUT. GA′ adopts the configurations of the genetic algorithm for covering array generation as its chromosomes. The fitness of a chromosome is constructed as $<f_1, f_2>$, where f_1 is the size of the covering array generated by a GA with this chromosome and f_2 is the corresponding execution time for covering array generation. GA′ will initially favor chromosomes with smaller values of f_1, breaking ties by selecting the chromosome with a smaller value of f_2. Moreover, since the results of the base choice experiment show that, for CA generation, roulette-wheel selection (e.g., GA climb) has no obvious superiority over random selection (e.g., GAr climb), GA′ will first incorporate truncation and tournament selection, other operators that may enable the achievement of improved efficiency. Following the design of the GA in this paper, GA′ will employ multipoint crossover and mutation. Since the execution of GA′ will be extremely time-consuming, we plan to run this experiment on a parallel computing platform such as Hadoop [6].

Ultimately, the combination of the results from running GA′ with the insights revealed in this paper will lead to a full-featured understanding of and a comprehensive framework for generating covering arrays with genetic algorithms. We anticipate that this will yield well-understood, efficient, and effective methods for performing combinatorial testing on highly configurable, modern software systems. Since the approaches presented in this paper and proposed as part of future work are generally applicable, we also intend to use them to improve other search-based techniques (e.g., simulated annealing [7]), thus enabling us to complement and extend prior work that uses other difficult-to-configure methods for combinatorial testing.

References

1 Changhai Nie, Hareton Leung, A Survey of Combinatorial Testing, *ACM Computing Surveys*, 43(2), 2011.

2 Syed A. Ghazi, Moataz A. Ahmed, Pair-wise Test Coverage Using Genetic Algorithms, In *Proc. of the Congress on Evolutionary Computation*, 2003.

3 Toshiaki Shiba, Tatsuhiro Tsuchiya, Tohru Kikuno, Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing, In *Proc. of the 28th Annual International Computer Software and Applications Conference*, 2004.

4 James D. McCaffrey, An Empirical Study of Pairwise Test Set Generation Using a Genetic Algorithm, In *Proc. of the 7th International Conference on Information Technology*, 2010.

5 Mats Grindal, Birgitta Lindström, Jefferson Offutt, Sten F. Andler, An Evaluation of Combination Strategies for Test Case Selection, *Empirical Software Engineering*, 11(4), 2006.

6 Apache Hadoop, http://hadoop.apache.org/, Date accessed: July 22, 2011.

7 Brady J. Garvin, Myra B. Cohen, Matthew B. Dwyer, Evaluating Improvements to a Meta-heuristic Search for Constrained Interaction Testing, *Empirical Software Engineering*, 16(1), 2011