e-Informatica Software Engineering Journal, Volume 10, Issue 1, 2016, pages: 9–29, DOI 10.5277/e-Inf160101

ABC-CAG: Covering Array Generator for Pair-wise Testing Using Artificial Bee Colony Algorithm

Priti Bansal[a], Sangeeta Sabharwal[a], Nitish Mittal[a], Sarthak Arora[b]
[a] *Netaji Subhas Institute of Technology, University of Delhi*
[b] *School of Computer Science and Engineering, Vellore Institute of Technology, Tamil Nadu*
bansalpriti79@gmail.com, ssab63@gmail.com, nitishmittal94@gmail.com, sarthak10193@gmail.com

Abstract

Testing is an indispensable part of the software development life cycle. It is performed to improve the performance, quality and reliability of the software. Various types of testing such as functional testing and structural testing are performed on software to uncover the faults caused by an incorrect code, interaction of input parameters, etc. One of the major factors in deciding the quality of testing is the design of relevant test cases which is crucial for the success of testing. In this paper we concentrate on generating test cases to uncover faults caused by the interaction of input parameters. It is advisable to perform thorough testing but the number of test cases grows exponentially with the increase in the number of input parameters, which makes exhaustive testing of interaction of input parameters imprudent. An alternative to exhaustive testing is combinatorial interaction testing (CIT) which requires that every t-way interaction of input parameters be covered by at least one test case. Here, we present a novel strategy ABC-CAG (Artificial Bee Colony-Covering Array Generator) based on the Artificial Bee Colony (ABC) algorithm to generate covering an array and a mixed covering array for pair-wise testing. The proposed ABC-CAG strategy is implemented in a tool and experiments are conducted on various benchmark problems to evaluate the efficacy of the proposed approach. Experimental results show that ABC-CAG generates better/comparable results as compared to the existing state-of-the-art algorithms.

Keywords: combinatorial interaction testing, pair-wise testing, covering array, artificial bee colony

1. Introduction

Testing plays a critical role in the software development life cycle (SDLC). It helps to improve the performance of the software system and ensure the delivery of quality and a reliable system. Often more than 50% of the entire software development resources are allocated to testing [1]. As the complexity of the software system increases so does the cost of testing, therefore testing the software effectively within a reasonable time and budget continues to be a challenge for the software testing community. One of the major factors in determining the quality of testing is the design of relevant test cases. Various types of testing techniques such as white-box testing and black-box testing are performed to detect faults in the software system. In black-box testing, test cases are generated from the specification of the system under test (SUT), whereas in the case of white-box testing, they are determined from the internal structure. However, in both cases the primary focus of the software testing community is to design a set of optimal test cases to uncover maximum faults in the software system within a reasonable time.

In a complex software system it has been found that the interaction of input parameters may cause interaction errors and to uncover these interaction errors, it is necessary to generate test cases that test all possible combinations of in-

put parameters. A software system with m input parameters, each having n values, will require a total of nm test cases to exhaustively test all the possible interactions among input parameters. Furthermore, the number of test cases increases exponentially with the increase in the number of parameters, which makes exhaustive testing impractical for large software systems. In the existing literature it has been reported that nearly 100% of the failures are triggered by interactions among 6 parameters. This is the main motivation behind Combinatorial Interaction Testing (CIT) which selects values of input parameters and combines them to generate test cases so as to test all t-way interactions of input parameters. CIT is a black-box testing technique which only requires information about the input parameters of the software system and their values. Empirical studies [2–5] show that a test set covering all possible 2-way combination of input parameter values is effective for software systems and can find a large percentage of the existing faults. Kuhn et al. [6] examined fault reports for many software systems and concluded that more than 70% of the faults are triggered by 2-way interaction of input parameters. Testing all 2-way interactions of input parameters values is known as pair-wise testing. The effectiveness of pair-wise testing in detecting a comparable number of faults early was a key factor driving the research work presented in this paper.

Covering Arrays (CAs) are combinatorial objects that are used to represent test sets for a system where the cardinality of values taken by each input parameter is the same. However, in a system with varying cardinalities of input parameter values, Mixed Covering Arrays (MCAs) are employed, which are a generalization of CAs that are used to represent test sets. The rows of CA/MCA correspond to test cases. As design of test cases is a crucial factor in determining the quality of software testing, the aim of CIT is to generate an optimal CA/MCA that provides 100% coverage of t-way interactions of input parameters. Lei and Tai [7] have shown that the problem of constructing optimal CA for pair-wise testing is NP complete. Many greedy [2, 3, 8–22] and meta-heuristic based optimizations algorithms/tools [23–34] have been developed by the researchers in the past with the aim of generating a near optimal CA/MCA. Both greedy and meta-heuristic techniques have merits and demerits. Greedy techniques are efficient as compared to their meta-heuristic counterparts in terms of CA/MCA generation time whereas meta-heuristic techniques generate optimal CA/MCA as compared to their greedy counterparts. The impressive results of existing meta-heuristic optimization algorithms to generate optimal CA/MCA motivated us to explore yet another optimization algorithm, namely the Artificial Bee Colony algorithm (ABC). It is proposed by Karaboga [35] and has been used to find an optimum solution for many optimization problems [36].

In this paper, we propose ABC-CAG (Artificial Bee Colony-Covering Array Generator) strategy that implements the ABC algorithm to generate optimal CA/MCA for pair-wise testing. The main contribution of this paper is to propose a strategy that integrates greedy approach and meta-heuristic approach thereby exploiting the strength of both techniques, to generate CA/MCA.

The remainder of this paper is organized as follows. In Section 2, we briefly describe combinatorial objects: CA and MCA. Section 3 discusses the existing state-of-the-art algorithms for constructing CA/MCA for pair-wise testing. In Section 4, we present a brief overview of ABC. Section 5 describes the proposed ABC-CAG strategy to generate CA for pair-wise testing. Section 6 describes the implementation of the proposed approach and presents empirical results to show the effectiveness of the proposed approach. Section 7 discusses threats to validity. Section 8 concludes the paper and future plans are discussed.

2. Background

This section discusses the necessary background related to combinatorial objects.

2.1. Covering Array

A covering array [37] denoted by $\mathrm{CA}_\lambda(N; t, k, v)$, is an $N \times k$ two dimensional array on v symbols such that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t at least λ times. If $\lambda = 1$, it means that every t-tuple needs to be covered only once and we can use the notation $\mathrm{CA}(N; t, k, v)$. Here, k represents the number of values of each parameter and t is the strength of testing. An optimal CA contains the minimum number of rows to satisfy the properties of the entire CA. The minimum number of rows is known as a covering array number and is denoted by $\mathrm{CA}\,N(t, k, v)$. A CA of size $N \times k$ represents a test set with N test cases for a system with k input parameters each having an equal number of possible values.

2.2. Mixed Covering Array

A mixed covering array [38] denoted by $\mathrm{MCA}(N; t, k, (v_1, v_2, \ldots, v_k))$, is an $N \times k$ two dimensional array, where $v_1, v_2, \ldots, v_k$ is a cardinality vector which indicates the values for every column. An MCA has the following two properties: i) Each column i $(1 \leq i \leq k)$ contains only elements from a set S_i with $|S_i| = v_i$ and ii) The rows of each $N \times t$ sub-array cover all t-tuples of values from the t columns at least once. The minimum N for which there exists an MCA is called a mixed covering array number and is denoted by $\mathrm{MCA}\,N(t, k, (v_1, v_2, \ldots, v_k))$. A shorthand notation can be used to represent MCAs by combining equal entries in $v_i : 1 \leq i \leq k$. An $\mathrm{MCA}(N; t, k, (v_1, v_2, \ldots, v_k))$ can be represented as $\mathrm{MCA}(N; t, k, (w_1^{q_1}, w_2^{q_2}, \ldots, w_s^{q_s}))$, where $k = \sum_{i=1}^{s} q_i$ and $w_j | 1 \leq j \leq s \subseteq \{v_1, v_2, \ldots, v_k\}$. Each element $w_j^{q_i}$ in the set $\{w_1^{q_1}, w_2^{q_2}, \ldots, w_s^{q_s}\}$ means that q_i parameters can take w_j values each. A MCA of size $N \times k$ represents a test set with N test cases for a system with k components each with a varying domain size.

3. Related Work

Over the past two decades mathematicians and researchers in computer science have proposed various algorithms and tools to generate CA/MCA. Mathematicians use algebraic methods to generate CA [39–42]. These methods are extremely fast, however, they are mostly designed to generate CAs only. Researchers in the field of software testing have designed greedy algorithms to construct optimal CAs and MCAs. Greedy algorithms use two approaches to construct CA/MCA: one-test-at-a-time and one-parameter-at-a-time. In one-test-at-a-time [2, 3, 8–21], CA is constructed by generating one test at a time until all the uncovered combinations are covered. Each subsequent test is generated in such a way that it can cover the maximum possible number of uncovered combinations. In the case of one-parameter-at-a-time approach, such as ACTS [22], a pair-wise test set is constructed by generating a pair-wise test set for the first two parameters and then extending it to generate a pair-wise test set for three parameters and continues to do so for each additional parameter.

Recently, meta-heuristic techniques such as simulated annealing (SA), particle swarm optimization (PSO), tabu search (TS), ant colony optimization (ACO), hill climbing (HC), genetic algorithm (GA) and cuckoo search (CS) have been used by researchers to generate optimal CA/MCA. Meta-heuristic search techniques start from a pre-existing CA/MCA or a population of CA/MCA and apply a series of transformations on them until until a CA/MCA that covers all the uncovered combinations is found. Greedy algorithms generate CA/MCA faster as compared to meta-heuristic techniques, however, meta-heuristic techniques usually generate smaller CA/MCA [38]. Table 1 gives a summary of existing tools/algorithms for constructing optimal CA/MCA.

Table 1. List of existing tools/algorithms to construct CA/MCA for pair-wise testing

S. No.	Tool/Algorithm	Maximum strength support(t)	Technique employed	Test generation strategy	Constraint handling
1	Test cover [39]	4	algebraic	one parameter at a time	✓
2	TConfig [40]	2			✗
3	CTS [41]	4			✓
4	Algebraic method [42]	2			✗
5	AETG [2]	2	greedy	one test at a time	✓
6	TCG [8]	2			✓
7	ITCH [9]	6			✓
8	TVG [10]	6			✓
9	AllPairs [11]	2			✗
10	PICT [12]	6			✓
11	Jenny [13]	8			✓
12	Density [14]	3			✗
13	DA-RO [15]	3			✗
14	DA-FO [15]	3			✗
15	TSG [16]	3			✗
16	G2Way [17]	2			✗
17	GTWay [18]	12			✗
18	MT2Way [19]	2			✗
19	EPS2Way [20]	2			✗
20	CASCADE [21]	6			✓
21	ACTS (IPOG) [22]	6	greedy	one parameter at a time	✗
22	Paraorder [14]	3			✗
23	GA [23]	3	genetic algorithm	test based generation	✗
24	ACA [23]	3	ant colony optimization		✗
25	PSO [24]	2	particle swarm optimization		✗
26	TSA [25, 26]	6	tabu search		✗
27	SA [27]	6	simulated annealing		✗
28	PPSTG [28]	6	particle swarm optimization		✗
29	CASA [29]	3	simulated annealing		✓
30	GAPTS [30]	2	genetic algorithm		✗
31	PWiseGen [31]	2	genetic algorithm		✗
32	GA [32]	2	genetic algorithm		✗
33	CS [33]	6	cuckoo search		✗
34	FSAPSO [34]	4	adaptive particle swarm optimization		✗
35	PSO [24]	2	meta-heuristic	particle swarm optimization	parameter based generation	✗

4. Artificial Bee Colony (ABC) Algorithm

The application of Artificial Intelligence (AI) based algorithms to solve various optimization problems in the field of software testing is an emerging area of research. AI based algorithms can be classified into different groups depending upon the criteria being considered such as population based, iterative based, stochastic based, deterministic, etc. [43]. Population based algorithms commence with a population of individuals (initial solutions) and evolve the population to generate a better solution by performing various recombination and mutation operations. Population based AI algorithms are further categorized as evolutionary algorithms (EA) and swarm intelligence (SI) based algorithms. The well-known EAs are Genetic Algorithm (GA), Genetic Programming (GP), Evolutionary Programming (EP), Evolution Strategy (ES) and Differential Evolution (DE). SI based algorithms are inspired by the collective behaviour of social insect colonies and other animal societies [44]. Various algorithms have been developed by researchers by modelling the behaviour of different swarms of social insects such as ants and bees, flocks of birds or schools of fishes. The well known SI based algorithms are Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO) and algorithms based on the specific intelligent behaviour of honey bee swarms such as honey bee mating optimization (HBMO) [45], ABC [36], Bee Colony optimization (BCO) [46] and Bee Swarm optimization(BSO) [47].

ABC is a swarm-based algorithm which simulates the intelligent foraging behavior of a honey bee swarm. Many researchers have compared the performance of ABC with other optimization algorithms such as GA, PSO, ACO and DE by evaluating their performance on various numerical functions which consist of unimodal and multimodal distributions [48]. In [49], Mala et al. proposed parallel ABC to generate optimal test suites for white box testing of software systems with path coverage, state coverage and branch coverage as test adequacy criteria and compared the performance of parallel ABC with sequential ABC, GA and random testing. The results of comparison showed that ABC is more effective than other optimization algorithms. Optimization algorithms are characterized by a trade-off between two mechanisms, namely exploration and exploitation and it is desirable to have a suitable balance between the two. the exploration process refers to the ability of the algorithm to look out for a global optimum whereas; exploitation process refers to the ability of applying the knowledge of previous solutions to look for better solutions (local search). In the ABC algorithm, an artificial bee colony is divided into three groups: employed bees, onlooker bees and scouts. Exploitation is done by means of employed bees and onlooker bees, and exploration is done by means of scouts. The number of employed bees or onlooker bees is equal to the number of individuals (solutions) in the population. In ABC the position of a food source represents a possible solution to the optimization problem and the nectar amount of the food source corresponds to the fitness (quality) of the food source. The various steps of ABC are as follows:

Step 1: Generation of initial population – ABC starts by generating an initial population of SN possible solutions to the given optimization problem randomly. Each solution $x_i \{i = 1, \ldots, SN\}$ is a D-dimensional vector, where D is the number of optimization parameters.

Step 2: Employed Bees Phase – Each employed bee selects a solution $x_i \{i = 1, \ldots, SN\}$ and tries to produce a new solution $v_i \{i = 1, \ldots, SN\}$ by updating the selected solution x_i using Equation (1). It then applies a greedy selection between the old solution and the newly generated solution and selects the one which has higher fitness (nectar amount of the food source).

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) \tag{1}$$

Here,

x_{ij} (or v_{ij}) denotes the j^{th} dimension of x_i (or v_i),

$j \in \{1, 2, \ldots, D\}$ is a randomly selected dimension,

x_k is a randomly selected neighbour of $x_i | k \in \{1, 2, \ldots, SN\}$,

```
generate initial population
iteration=1
while (solution not found and iteration ≤ maximum number of iterations)
   employed bees phase
   onlooker bees phase
   scout phase
   memorize the best solution achieved so far
   iteration = iteration + 1
end while
```

Figure 1. ABC algorithm

SN is the number of food sources (solutions) in the population.
Although k is determined randomly, it has to be different from i. ϕ_{ij} is a random number between $[-1, 1]$. Subsequently, once all employed bees have performed the search operation, a probability is assigned to each solution $x_i | 1 \leq i \leq SN$ which is calculated using Equation (2).

$$P(x_i) = \frac{\text{fitness}(x_i)}{\sum_{n=1}^{SN} \text{fitness}(x_n)} \tag{2}$$

Step 3: Onlooker Bees Phase – The ensuing phase of onlooker bees selects a solution based on the probability assigned to them and performs modification on the selected solution using Equation (1). Then a greedy selection is applied as done in case of employed bees.
Step 4: Scout Phase – Scouts look out for a solution which has not been improved by employed bees or onlooker bees through a predefined number of cycles called the limit and replaces it with a randomly generated solution.
Step 2 – Step 4 are repeated until a solution is found or a maximum number of iterations is reached. For further explanation of the ABC algorithms, readers can refer to [35, 43]. ABC is good at exploration but poor in exploitation as employed bees and onlooker bees only modify a small part of the solution instead of taking the global best, which may lead to the trapping of the ABC in local minima [48]. In order to maintain a good balance between exploration and exploitation, various variants of ABC namely GABC [50], I-ABC [48] and PS-ABC [48] were proposed by researchers.
The outline of ABC is shown in Figure 1.

5. ABC-CAG Strategy to Generate CA

In this paper, we propose aa ABC-CAG strategy which applies ABC, a stochastic search based optimization algorithm to solve the problem of constructing an optimal CA/MCA for pair-wise testing. From now onwards CA refers to both CA and MCA unless mentioned explicitly. We start by defining a search space, which in our case consists of the input domain of all input parameters of the SUT. Let us consider a SUT having k input parameters. For each input parameter $IP_j \mid 1 \leq j \leq k$, the possible values of IP_j are represented by an integer number between $[0, v_j)$ where v_j is the maximum number of values that IP_j can have. We start by generating an initial population of PS individuals where an individual in our case is a CA that corresponds to a food source in ABC and represents a possible solution to the given problem. Each covering array $\text{CA}_i \mid i \in \{1, 2, \ldots, PS\}$ in the population is a $N \times k$ dimensional array. Let us consider a web based application where the customer has different options of the operating system, browsers, display, processor and memory which they may use as shown in Table 2.

In order to test the system thoroughly, we need to test the system on all possible configurations, e.g. Android, Safari, 240×320, dual core, 512 MB, etc. which would require a total of $5 \times 3 \times 3 \times 3 \times 4 = 540$ test cases, whereas only 20 test cases will be required to test all pair-wise combinations of features. A possible solution (set of test cases/MCA) to the test pair-wise interactions of features in Table 2 is shown in Figure 2.

Table 2. A web based application

Operating System	Browser	Display	Processor	Memory
Android	Opera mini	128×160	single core	256 MB
iOS	Safari	240×320	multi core	512 MB
Windows	Chrome	800×1280	dual core	1 GB
Blackberry				2 GB
Symbian				

$$\begin{vmatrix} \textit{Blackberry} & \textit{Safari} & \textit{800×1280} & \textit{single core} & \textit{256 MB} \\ \textit{iOS} & \textit{Opera mini} & \textit{800×1280} & \textit{dual core} & \textit{256 MB} \\ . & . & . & . & . \\ . & . & . & . & . \\ \textit{Windows} & \textit{Chrome} & \textit{126×160} & \textit{multi core} & \textit{512 MB} \end{vmatrix}$$

Figure 2. MCA of size 20×5 for pair-wise testing

When generating the initial population of $N \times k$ CAs, N is unknown at the start of the search process. So there are two possibilities. The first one is to use the method suggested by Stardom [51], where we set loose lower bound (LB) and upper bound (UB) on the size of an optimal CA and then apply binary search repeatedly to find the smallest CA. The second one, in case the size of N is known in advance, i.e. best bound achieved in the existing state-of-the-art, we start with the known size and try to optimize it further.

As discussed in Section 4, the fitness of a CA in the population is the measure of its quality. In our case, the fitness of a CA is defined as the total number of distinct 2-way interactions covered by it and is calculated using Equation (3) as:

$$\text{fitness}(\text{CA}_i) = |\{2-\text{way interactions covered by } \text{CA}_i\}| \quad \forall\, i \in \{1, \ldots, PS\} \quad (3)$$

The aim of ABC-CAG is to generate a covering array that maximizes the objective function $f|f : \text{CA}_i \rightarrow I^+$, where I^+ represents a set of positive integers. The objective function f tells us, how good a solution it is for a given problem. In our case f can be calculated as shown in Equation (4):

$$f(\text{CA}_i) = \text{fitness}(\text{CA}_i) \mid i \in \{1, \ldots, PS\} \quad (4)$$

Now, ABC-CAG tries to generate a covering array CA_{max} that maximizes the objective function f as shown in Equation (5):

$$\text{CA}_{\max} = \text{CA}_i \,|f(\text{CA}_i) = \max(f(\text{CA}_i) \quad \forall\, i \in \{1, \ldots, PS\}) \quad (5)$$

Having defined the problem representation and the fitness calculation, the next sub-section describes the various steps of the ABC-CAG strategy.

5.1. Generation of Initial Population

In EA's the role of the impact of the initial population on their performance cannot be ignored as it can affect the convergence speed and quality of the final solution [52]. Many population initialization methods exist to generate initial population. The most common method used to generate the initial population in EA is the random method. ABC-CAG uses the Hamming distance approach proposed by Bansal et al. [32] to generate a good quality initial population of CAs. The motive behind the use of the Hamming distance approach is to generate test cases in a CA in such a way that each new test case covers the maximum number of possible distinct 2-way interactions not covered by the previous test cases. The Hamming distance between two test cases (rows) in a CA is the number of positions in which they differ. Let PS be the population size and N be the size of CA. For each CA in PS, 50% (first N/2) of the test cases are created randomly. Let $tc_1, tc_2, \ldots, tc_i$ represent the test cases in CA generated till now. To create the next test case tc_j where $j = i + 1$,

a candidate test case tc is generated randomly and the Hamming distance of tc from tc_k, for all $k : 1 \leq k \leq i$, denoted by distance(tc), is calculated as:

$$\text{distance}(tc) = \sum_{k=1}^{i} \text{HD}(tc_k, tc) \quad (6)$$

Where, $\text{HD}(tc_k, tc)$ is the Hamming distance between tc_k and tc. An average distance denoted by avg_distance(tc) is calculated as follows.

$$\text{avg\_distance}(tc) = \text{distance}(tc)/(j-1) \quad (7)$$

Candidate test case tc is included in the test set TS only if

$$\text{avg\_distance}(tc) \geq \alpha \times N_{IP} \quad (8)$$

Where, α is a diversity factor whose value ranges from 0.3 to 0.4 and N_{IP} is the number of input parameters. Equation (8) implies that a candidate test case tc is included only if it covers at least 30%-40% of distinct input parameter values as compared to those covered by the existing test cases. The process is repeated until the remaining $N/2$ test cases are generated for the CA. The use of the Hamming distance to create $N/2$ test cases in each CA enhances the quality of initial population as compared to one generated using the random technique.

5.2. The Employed Bees Phase

The number of employed bees is equal to the population size PS. In ABC, each employed bee selects a dimension of a food source randomly and uses the local information to modify the solution using Equation (1). The new solution replaces the old one only if the former is better than the latter. However, in ABC-CAG each employed bee uses a greedy approach to select a test case (dimension) of a CA_i (solution). The impetus behind the greedy selection of test case in a CA_i by an employed bee is to formulate a new CA'_i from the selected CA_i in such a way that CA'_i contains all the test cases of the selected CA_i except its worst test case. The worst test case of CA_i is replaced in CA'_i by a test case generated using the information of a randomly selected neighbouring CA_m in an attempt to increase the overall coverage of 2-way interactions between input parameters. To select the worst test case in CA_i the fitness of each test case in CA_i is calculated by counting the number of distinct pairs covered by each one of them. For instance, suppose we have a CA with nine test cases as shown in Figure 3, for the system shown in Table 2.

We calculate the number of distinct pairs covered by each test case as shown in Table 3 and an employed bee selects the test case that covers the least number of distinct pairs (test case TC8 in Table 3). The value of each input parameter $IP_j\{j = 1, 2, \ldots, k\}$ of the test case covering the least number of distinct pairs is modified based on the values of the respective input parameters in the corresponding test case of the randomly selected neighbouring CA, i.e. CA_m using Equation (9).

$$\text{CA}'_{iqj} = \text{CA}_{iqj} + \phi_{iqj}(\text{CA}_{iqj} - \text{CA}_{mqj}) \quad (9)$$

Here, $i \in \{1, 2, \ldots, PS\}$, $q \in \{1, 2, \ldots, N\}$ represent the index of the worst test case in $\text{CA}_i, j \in \{1, 2, \ldots, k\}$ and $m \in \{1, 2, \ldots, PS\} \mid m \neq i$. Since ϕ_{iqj} is a random number between $[-1, 1]$, it is quite possible that a non-integral value may get generated as a result of calculation performed using Equation (9). To avoid such a condition, whenever a non-integer value is generated for an input parameter, it gets rounded to the nearest integer number. After rounding off the value, if it does not fall in the range $[0, v_j)$ then a value is selected randomly from the input domain of the respective parameter and it replaces the existing value of the selected parameter.

5.3. The Onlooker Bees Phase

Subsequently, the fitness of each CA in the search space is calculated and a probability is assigned to each of them using Equation (2). In ABC-CAG, to select a CA on the basis of the probability assigned to them, a random number is generated in the range $[0, 1]$ and based on the interval in which the random number falls; a covering array CA_i is selected by an onlooker bee. Unlike the traditional ABC which is good at exploration but

iOS	*Chrome*	*240×320*	*dual core*	*1 GB*
Android	*Chrome*	*800×1280*	*multi core*	*256 MB*
Blackberry	*Safari*	*800×1280*	*multi core*	*256 MB*
Blackberry	*Chrome*	*128×160*	*single core*	*2 GB*
Symbian	*Opera mini*	*128×160*	*multi core*	*512 MB*
Windows	*Safari*	*800×1280*	*dual core*	*512 MB*
iOS	*Opera mini*	*800×1280*	*multi core*	*2 GB*
Android	*Chrome*	*128×160*	*multi core*	*512 MB*
Blackberry	*Opera mini*	*240×320*	*single core*	*1 GB*

Figure 3. CA of size 9×5

Table 3. Calculation of distinct pairs covered by each test case of CA

TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
iOS, Chrome	**Android**, **Chrome**	Blackberry, Safari	Blackberry, Chrome	Symbian, Opera mini	Windows, Safari	iOS, Opera mini	**Android**, **Chrome**	Blackberry, Opera mini
iOS, 240×320	Android, 800×1280	Blackberry, 800×1280	Blackberry, 128×160	Symbian, 128×160	Windows, 800×1280	iOS, 800×1280	Android, 128×160	Blackberry, 240×320
iOS, dual core	**Android**, **multi core**	Blackberry, multi core	*Blackberry*, **single core**	Symbian, multi core	Windows, dual core	iOS, multi core	**Android**, **multi core**	*Blackberry*, *single core*
iOS, 1 GB	Android, 256 MB	Blackberry, 256 MB	Blackberry, 2 GB	Symbian, 512 MB	Windows, 512 MB	iOS, 2 GB	Android, 512 MB	Blackberry, 1 GB
Chrome, 240 × 320	Chrome, 800 × 1280	**Safari**, **800×1280**	*Chrome*, *128×160*	Opera mini, 128×160	**Safari**, **800×1280**	Opera mini, 800×1280,	*Chrome*, *128×160*	Opera mini, 240×320
Chrome, dual core	**Chrome**, **multi core**	Safari, multi core	Chrome, single core	*Opera mini*, *multi core*	Safari, dual core	*Opera mini*, *multi core*	**Chrome**, **multi core**	Opera mini, single core
Chrome, 1 GB	Chrome, 256 MB	Safari, 256 MB	Chrome, 2 GB	Opera mini, 512 MB	Safari, 512 MB	Opera mini, 2 GB	Chrome, 512 MB	Opera mini, 1 GB
240×320, dual core	**800×1280**, **multi core**	**800×1280**, **multi core**	128×160, single core	*128×160*, *multi core*	800×1280, dual core	**800×1280**, **multi core**	*128×160*, *multi core*	240×320, single core
240×320, *1 GB*	**800×1280**, **256 MB**	**800×1280**, **256 MB**	128×160, 2 GB	***128×160***, ***512 MB***	800×1280, 512 MB	800×1280, 2 GB	***128×160***, ***512 MB***	*240×320*, *1 GB*
dual core, 1GB	**multi core**, **256 MB**	**multi core**, **256 MB**	single core, 2 GB	*multi core*, *512 MB*	dual core, 512 MB	multi core, 2 GB	*multi core*, *512 MB*	single core, 1 GB
Distinct pairs covered by each test case:								
9	4	6	8	6	9	8	**3**	8

poor at exploitation, ABC-CAG takes advantage of the global best CA denoted by CA^{best} in the population (based on gbest-guided ABC (GABC) developed by Zhu and Kwong [50]) to guide the search of a candidate solution and modifies the selected CA_i. Like an employed bee, an onlooker bee selects the worst test case (dimension) of the selected CA_i and replaces it with a test case that is generated by using the information of the global best CA i.e., CA^{best} and a randomly selected neighbouring CA i.e., CA_m using Equation (10).

$$\text{CA}'_{iqj} = \text{CA}_{iqj} + \phi_{iqj}(\text{CA}_{iqj} - \text{CA}_{mqj}) + \psi_{iqj}(\text{CA}^{\text{best}}_{qj} - \text{CA}_{iqj}) \quad (10)$$

Here, $\text{CA}^{\text{best}}_{qj}$ is the value of j^{th} parameter of q^{th} test case of the global best CA^{best}, ψ_{iqj} is a uniform random number in [0, C], where C is a non-negative constant. The GABC technique drives the new candidate solution CA'_i towards the global best solution, thereby improving its exploitation capabilities.

However, in case the best CA i.e., CA^{best} gets selected per se, based on the generated random number; the ABC-CAG modifies it by replacing its worst test case by a smart test case. A smart test case is constructed by selecting the value for each parameter greedily. For each parameter, the value whose occurrence in the best CA is minimum is selected. The replacement of the worst test case in the best CA by a smart test case is done to make sure that certain new pairs get covered by this replacement. An example to illustrate the selection and modification done by onlooker bees phase is shown in Table 4.

Let us consider a system having configuration $(N; 2, 2^2 3^3)$ as shown in Table 4a.

The total number of distinct 2-way interactions in this system is 67. Let our population size PS be 8 which means that the population consists of 8 CAs and let us assume that the size of each CA array is 7×5 which means that a CA consists of 7 test cases. After an initial population of CAs is generated, each employed bee modifies a CA as discussed in Section 5.2. The fitness and the probability assigned to each CA generated after being modified by the employed bee is shown in Table 4b.

Here, CA_4 is the global best CA. Let a random number 'r' be generated to select a CA by the onlooker bee. There are two cases:
Case 1: When the global best CA is different from the CA selected by the onlooker bee – let r be 0.8. Based on the value of r, covering array CA_7 gets selected by the onlooker bee and lets CA_1 be the randomly selected neighbour of CA_7. Also according to the fitness values, CA_4 is CA^{best}. The test cases of CA1, CA4 and CA7 are shown in Tables 4c–4e.

After calculating the number of distinct pairs covered by each test case of CA_7, it is clear that TC2 covers the least number of distinct pairs i.e., 1. So the value of each parameter in the test case TC2 of CA_7 is modified using Equation (10). For performing calculations, the values of each input parameter have been mapped to integer values (0/1/2) and the new covering array CA'_7 generated after modification is shown in Table 4f.

After modification by the onlooker bee based on the global best CA and the randomly selected neighbouring CA, the fitness of the newly generated covering array CA'_7 increases by 2 and becomes 47.
Case 2: When the global best CA and the CA selected by the onlooker bee is the same – In this case the worst test case of the global best covering array CA^{best} is replaced by a smart test case. However, in our case the global best covering array CA_4 has three test cases: TC1, TC3 and TC4 that cover the least number of distinct pairs. Here, ABC-CAG selects a test case randomly from the three test cases covering the least number of distinct pairs. Let the randomly selected test case be TC3. Now, TC3 will be replaced by a smart test case which in this case is 'b1 b2 b3 c4 b5' as these values have the least number of occurrences in CA_4. The replacement of the worst test case of CA_4 by the smart test case increases its fitness by 2.

The above procedure is repeated for each onlooker bee.

5.4. The Scouts Phase

ABC's exploration strategy is effectuated by scout bees replacing a food source abandoned by an employed bee with a randomly generated food source. To further enhance the exploration capability of ABC, we use a greedy approach to select a CA instead of the primitive approach followed by ABC. In ABC-CAG, a scout replaces the worst CA (least fitness) in the population by a new CA. In ABC, the food sources that cannot be improved through a predetermined threshold called the limit are abandoned. The aforementioned abandoned food sources are thereupon replaced by randomly created new food sources by an artificial scout. ABC-CAG necessitates setting the frequency of the scout operation with discretion: a very high value of frequency will proliferate diversity of the population and avoid getting stuck in local minima but concurrently makes it difficult to converge to a good solution, whereas a lower value of frequency will result in early convergence leading to a suboptimal solution. Hence, it is required to set the frequency of scout denoted by f_{scout} to an optimal value.

Table 4. Example: the selection and modification done by the onlooker bees phase

a) A $(N; 2, 2^2 3^3)$ system

IP1	IP2	IP3	IP4	IP5
a1	a2	a3	a4	a5
b1	b2	b3	b4	b5
		c3	c4	c5

b) The fitness and the probability assigned to each CA

CA	CA_1	CA_2	CA_3	CA_4	CA_5	CA_6	CA_7	CA_8
Fitness	49	47	51	52	45	44	45	47
Probability	0.119	0.114	0.134	0.137	0.118	0.116	0.118	0.114

c) CA_1 (randomly selected neighbouring CA)

TC1					TC2					TC3					TC4					TC5					TC6					TC7				
a1	a2	a3	a4	b5	b1	a2	b3	a4	c5	a1	a2	c3	a4	c5	a1	b2	a3	c4	b5	a1	b2	a3	a4	c5	b1	a2	c3	b4	b5	b1	b2	b3	c4	a5

d) CA_4 (best CA)

TC1					TC2					TC3					TC4					TC5					TC6					TC7				
a1	a2	a3	b4	b5	a1	a2	c3	b4	b5	b1	b2	c3	b4	c5	a1	a2	b3	a4	c5	a1	a2	c3	c4	c5	a1	b2	a3	c4	c5	b1	a2	a3	a4	c5

e) CA_7 (CA selected by onlooker on the basis of probability)

TC1					**TC2**					TC3					TC4					TC5					TC6					TC7				
a1	a2	a3	b4	b5	**a1**	**a2**	**c3**	**b4**	**b5**	b1	b2	c3	b4	c5	a1	a2	b3	a4	c5	a1	a2	c3	c4	c5	a1	b2	a3	c4	c5	b1	a2	a3	a4	c5
Minimum distinct pairs covered by each test case:																																	
2					1					6					5					2					4					4				

f) CA'_7 (CA_7 after modifications)

TC1					**TC2**					TC3					TC4					TC5					TC6					TC7				
a1	a2	a3	b4	b5	**b1**	**a2**	**b3**	**c4**	**c5**	b1	b2	c3	b4	c5	a1	a2	b3	a4	c5	a1	a2	c3	c4	c5	a1	b2	a3	c4	c5	b1	a2	a3	a4	c5

ABC-CAG replaces the worst CA by a randomly generated CA after every f_{scout} generation. For example, in the example given in Section 5.3, a scout will replace the worst covering array i.e., CA_6 by a randomly created new CA.

All the three phases, namely the employed bees phase, the onlooker bees phase and the scout phase are perpetuated until a solution is found or the maximum number of generations is reached.

6. Evaluation

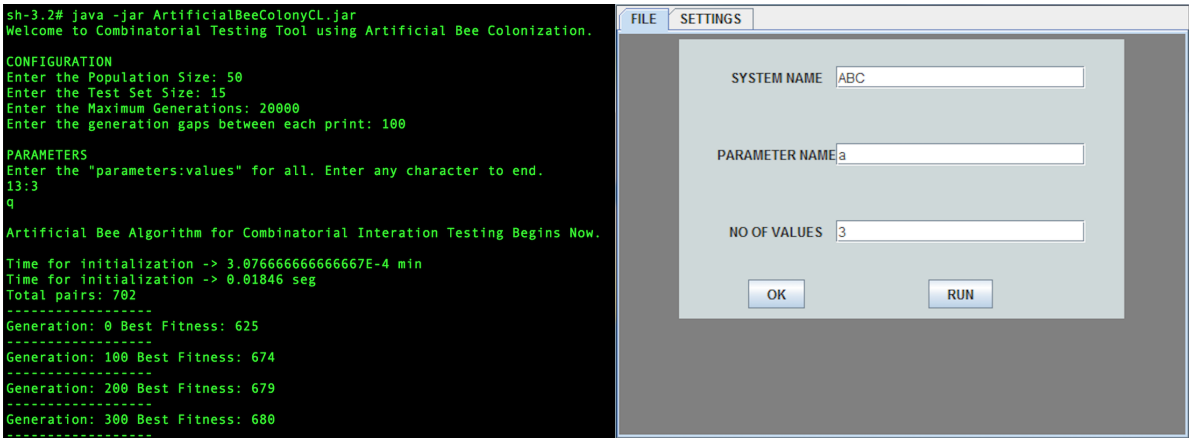
We start our evaluation by presenting three research questions in Section 6.1. Then we outline our implementation and experimental design in Section 6.2. The results and analysis are shown in Section 6.3.

6.1. Research Questions

Many greedy and meta-heuristic techniques have been proposed by researchers in the past with the aim of generating an optimal CA. The ultimate goal of the research work presented in this paper is to develop a strategy that generates optimal CA as compared to existing state-of-the-art algorithms/tools. Our first objective is therefore to measure the effectiveness of the proposed approach:

RQ1: (Comparison of ABC-CAG with existing techniques) How effective is ABC-CAG in generating an optimal CA with respect to the existing state-of-the art algorithms/tools?

In addition to evaluating the effectiveness of ABC-CAG, it is important to check whether ABC-CAG is comparable in terms of runtime



a) Command Line Interface (CLI) b) Graphical User Interface (GUI)

Figure 4. ABC-CAG interfaces

with the state-of-the art algorithms. Our next research question is:

RQ2: (Efficiency of ABC-CAG) How efficient is ABC-CAG in generating an optimal CA?

ABC is a meta-heuristic search algorithm and all search algorithms are randomized in nature. Randomized algorithms are used to solve problems where it is not possible to solve the problem in a deterministic way within a reasonable time and they are associated with a certain degree of randomness as part of their logic. Due to the randomized nature of search algorithms, running them on the same problem instance multiple times produces different results. It is therefore important to analyse their results and compare them against simpler alternatives. This motivates our next research question:

RQ3: (Effectiveness of ABC-CAG) How effective is ABC-CAG when applied to the problem of generating an optimal CA for pair-wise testing as against existing meta-heuristic techniques?

6.2. Experimental Design

To answer the research questions asked in Section 6.1, we have implemented ABC-CAG using Java. Two types of external interfaces have been provided: Command Line Interface (CLI) and Graphical User Interface (GUI) which are shown in Figure 4a and Figure 4b, respectively.

ABC-CAG takes population size PS and maximum number of iterations NI as input. As discussed in Section 5, in ABC-CAG there is an option of whether we want to start from a known N or we want to start with a large random array whose size is calculated as suggested by Stardom [51]. If we start with a known N, then the user has to supply the value of N as input to ABC-CAG.

To answer RQ1 and RQ2, three sets of experiments were conducted. In the first experiment, a Traffic collision avoidance system (TCAS) benchmark, which has been used by many researchers [25, 26, 53] in the past to compare CA generation strategies, is taken to evaluate the performance of ABC-CAG with respect to the existing state-of-the-art algorithms. TCAS has 12 control parameters with 2 parameters having 10 values, 1 having 4 values, 2 having 3 values and 7 having 2 values each. It can be represented by an MCA instance MCA$(N; 2, 12, 10^2 4^1 3^2 2^7)$.

In the second experiment, we took a case study of various features of a printer that are available while printing a document as shown in Figure 5. The printer case study is a practical example that models and illustrates the concept of combinatorial testing.

It is clear from Figure 5 that there are 7 features that a user can set during printing. However, the feature 'Resolution' is only for information and a user cannot change its value. So, we consider only 6 features and regard them as input parameters. Out of these 6 input parameters, 3 parameters

have 2 values each, 2 parameters have 3 values each whereas the remaining 1 parameter has 4 values as shown in Table 5. This problem can be represented by an MCA instance $MCA(N; 2; 5; 2^3 3^2 4^1)$.

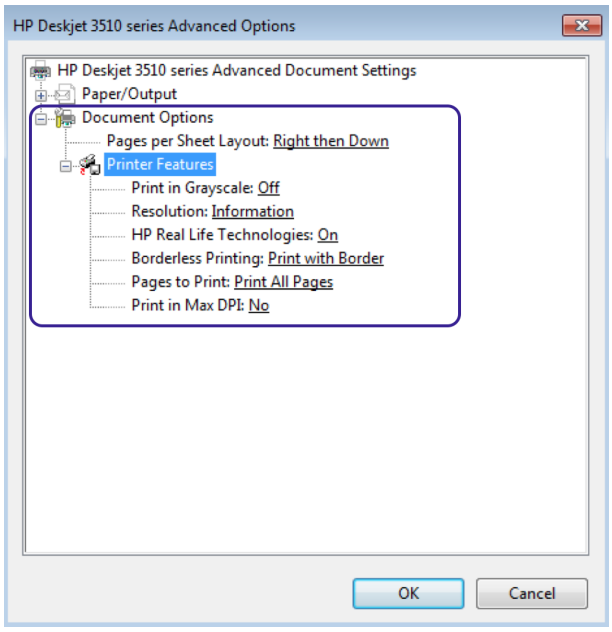


Figure 5. Various features of a printer

To ensure fair comparison and to evaluate the effectiveness of ABC-CAG, in the third experiment we took a dataset that consists of 20 benchmark problems selected carefully from the existing literature [25–28,38,54] on pair-wise testing as shown in Table 6. Each benchmark problem in the dataset is either an instance of CA or MCA. For example the problem 3^3 in Table 6 represents a CA instance $\mathrm{CA}(N; 2, 3, 3^3)$ which means that the system has 3 input parameters each having 3 possible values and the strength of testing is 2. Similarly the problem $5^1 3^8 2^2$ in Table 6 represents an MCA instance $MCA(N; 2, 11, 5^1 3^8 2^2)$ which means that the system has 11 input parameters with 1 parameter having 5 values, 8 parameters having 3 values and 2 parameters having 2 values each and the strength of testing is 2.

As CA size is absolute and is independent of the system configuration, to answer RQ1, we compared CA sizes based on the data available in the existing literature [21, 25–28, 33, 34, 38, 54]. However, CA generation time is dependent on the system configuration, so to answer RQ2 and to ensure a fair comparison, we limited our comparison based on CA generation time to only those algorithms whose implementations are publicly available, which includes greedy algorithms based tools, namely AllPairs, Jenny, TVG, ACTS(IPOG), and meta-heuristic algorithms based tools namely CASA, PWiseGen [55]. The CA generation time was obtained by executing the dataset of Table 5 on these tools under Windows using an INTEL Pentium Dual Core 1.73 GHZ processor with 1.00 GB of memory.

To answer RQ3, we need to perform a statistical test to compare ABC-CAG with meta-heuristic techniques. We compared ABC-CAG with two meta-heuristic tools namely PWiseGen and CASA.

PWiseGen starts with a known array size N and tries to generate a CA that covers 100% pair-wise combinations of input parameter values. Therefore, to compare ABC-CAG and PWiseGen we ran each problem on both of them 30 times and noted down the fitness of the CAs/MCAs obtained during these runs. The value of N was kept the same for a problem during these 30 runs. To compare ABC-CAG and CASA, we ran each problem on ABC-CAG and CASA 30 times and noted down the size of generated CA/MCA.

6.3. Results and Analysis

Here we present the results of experiments conducted to answer RQ1, RQ2 and RQ3.

6.3.1. Comparison of ABC-CAG with Existing Techniques (RQ1)

Results: The result of experiments performed to compare ABC-CAG with the existing techniques for TCAS, the printer case study and the dataset shown in Table 6 are shown in Table 7, Table 8 and Table 9 respectively. Entries marked '–' mean that the results are not available. Since ABC-CAG is a randomized algorithm and it gives different results when run multiple times on the same problem instance, therefore we report the best as well as average CA size obtained over multiple runs.

Table 5. Various features of a printer

HP Real Life Technologies	Borderless Printing	Print in max DPI	Pages per sheet layout	Print in grayscale	Pages to print
On	Print with border	No	Left then down	Off	Print all pages
Off	Print borderless	Yes	Down then left	High quality grayscale	Print even pages only
			Right then down	Black ink only	Print odd pages only
			Down then right		

Table 6. Dataset

Sno.	Benchmark Problems	k (number of input parameters)	Total number of pairs
1	3^3	3	27
2	3^4	4	54
3	3^{13}	13	702
4	5^{10}	10	1125
5	10^{20}	20	19000
6	2^{100}	100	19800
7	$4^5 3^4$	9	454
8	$5^1 3^8 2^2$	11	492
9	$7^2 6^2 4^2 3^2 2^2$	10	854
10	$8^2 7^2 6^2 5^2$	8	1178
11	$6^4 4^5 2^7$	16	1556
12	$5^1 4^4 3^{11} 2^5$	21	1944
13	$6^1 5^1 4^6 3^8 2^3$	19	1992
14	$6^2 4^9 2^9$	20	2052
15	$6^5 5^5 3^4$	14	2074
16	$7^1 6^1 5^1 4^5 3^8 2^3$	19	2175
17	$6^9 4^3 2^7$	19	3000
18	$6^7 4^8 2^3$	18	3004
19	$4^{15} 3^{17} 2^{29}$	61	14026
20	$4^1 3^{39} 2^{35}$	75	17987

Analysis: It is clear from Tables 7–9 that ABC-CAG generates CA of smaller size as compared to greedy algorithms. When compared to meta-heuristic techniques, ABC-CAG generates comparable results in the case of TCAS and the printer case study, whereas it outperforms GA, ACA, PSO, PPSTG, PWiseGen and CS in the case of benchmark problems given in dataset of Table 6. When compared to CASA, it can easily be seen from Table 9 that in 60% cases the size of CAs generated by ABC-CAG and CASA are the same. In 25% of cases ABC-CAG generates smaller CAs whereas in only 15% of cases CASA outperforms ABC-CAG. In the case of TS, the results are comparable. Overall, ABC-CAG outperforms all greedy and most of the meta-heuristic techniques and generates a smaller size CA for pair-wise testing.

6.3.2. Efficiency of ABC-CAG (RQ2)

Results: The time (in seconds) taken by each of the publicly available tools, namely Jenny, All-Pairs, TVG, ACTS (IPOG), CASA, PWiseGen, and the proposed algorithm ABC-CAG for generating CA for TCAS, the printer case study and

Table 7. Comparison of MCA sizes generated for TCAS

Problem Instance	Pairs	ACTS (IPOG)	PICT	AllPairs	ITCH	Jenny	TConfig	TVG	TSA	CASA best	CASA avg.	PWise-Gen	ABC-CAG best	ABC-CAG avg.
$10^2 4^1 3^2 2^7$	837	100	100	100	120	108	108	100	100	100	100	101	100	100

Table 8. Comparison of MCA sizes generated for printer case study

Problem Instance	Pairs	ACTS (IPOG)	PICT	AllPairs	Jenny	TVG	CASA best	CASA avg.	PWise-Gen	ABC-CAG best	ABC-CAG avg.
$3^2 2^3 4^1$	105	12	14	16	14	13	12	12	12	12	12

Table 9. Comparison of CA/MCA sizes generated for the 20 benchmark problems

Benchmark Problems	AETG	TCG	All-Pairs	PICT	Jenny	CAS-CADE	ACTS (IPOG)	GA	ACA	PSO	TS	PPSTG	CASA best	CASA avg.	PWise-Gen	CS	FSAPO	ABC-CAG best	ABC-CAG avg.
3^3	–	–	9	10	9	–	9	–	–	9	–	–	9	9	9	9	9	9	9
3^4	9	–	9	12	11	9	9	9	9	9	–	9	9	9	9	9	9	9	9
3^{13}	15	20	17	20	18	–	19	18	18	18	–	17	16	16.24	16	20	16	15	15.93
5^{10}	–	–	47	47	45	–	45	–	–	–	–	45	38	39.7	43	–	–	41	41.6
10^{20}	180	218	197	216	193	–	227	227	232	213	–	–	192	193.33	224	–	–	210	211.5
2^{100}	10	16	14	16	16	–	16	14	14	–	–	–	11	11	11	–	–	10	10.43
$4^5 3^4$	–	–	22	26	26	–	24	–	–	–	19	–	19	20	21	–	–	19	19.83
$5^1 3^8 2^2$	20	20	20	20	23	–	19	17	17	17	15	21	15	16.24	16	21	18	15	15.96
$7^2 6^2 4^2 3^2 2^2$	–	–	54	56	57	–	53	–	–	–	–	–	49	49.3	50	–	–	49	49.6
$8^2 7^2 6^2 5^2$	–	–	64	80	76	–	72	–	–	–	64	–	64	65.13	72	–	–	64	64.4
$6^4 4^5 2^7$	–	–	45	55	53	–	44	–	–	–	38	–	41	52.8	47	–	–	39	41.9
$5^1 4^4 3^{11} 2^5$	30	30	27	32	32	–	26	26	27	27	22	–	22	23.7	26	–	–	22	23.9
$6^1 5^1 4^6 3^8 2^3$	34	41	34	38	40	–	36	33	34	35	30	39	30	30.26	33	43	35	30	30.2
$6^2 4^9 2^9$	–	–	38	41	44	–	39	–	–	–	36	–	36	36.4	39	–	–	36	36.16
$6^5 5^5 3^4$	–	–	53	59	56	–	56	–	–	–	50	–	47	49.33	55	–	–	51	52.5
$7^1 6^1 5^1 4^5 3^8 2^3$	45	45	43	46	50	–	43	43	43	43	42	49	42	42	43	51	–	42	42.16
$6^9 4^3 2^7$	–	–	59	67	64	–	61	–	–	–	51	–	52	53.23	61	–	–	51	51.66
$6^7 4^8 2^3$	–	–	53	63	63	–	54	–	–	–	47	–	48	49.86	57	–	–	47	48.1
$4^{15} 3^{17} 2^{29}$	37	33	35	38	39	–	33	38	38	38	30	30	30	30.53	33	–	–	30	30.43
$4^1 3^{39} 2^{35}$	27	–	26	29	31	–	28	29	28	27	22	–	22	22.9	24	–	–	22	22.73

the dataset of Table 6 are shown in Table 10, Table 11 and Table 12, respectively.

Analysis: It is evident from Tables 10–12 that meta-heuristic techniques take a longer time to generate CA as compared to their greedy counterparts. However, the extra time taken by meta-heuristic techniques allows them to generate smaller CA/MCA than greedy algorithms. When we compare the time taken by CASA, PWiseGen and ABC-CAG, it has been observed that out of the three meta-heuristic techniques, CASA takes the minimum time to generate CA whereas the time taken by PWiseGen and ABC-CAG is comparable.

6.3.3. Effectiveness of ABC-CAG (RQ3)

Results: To compare ABC-CAG and PWiseGen, we performed a statistical test namely Welch's t-test on the fitness of CAs/MCAs generated during 30 runs for each benchmark problem in the dataset except the two benchmark problems CA (3^3) and CA (3^4), TCAS and the printer case study, as the fitness of CAs generated during 30 runs of TCAS, the printer case study and the two aforementioned benchmark problems were the same for both techniques. Hence, there is no reason for performing the Welch t-test on these two problems. Each problem was run 30 times as a minimum of 30 data points are required for Welch's t-test [56]. From Table 7 and Table 9, it is evident that ABC-CAG generates smaller CAs than PWiseGen, however by performing Welch's t-test we try to assess whether the difference is significant or not. To do this, we formulate null hypothesis H_0 as:

There is no difference between the average fitness of CAs generated by ABC-CAG and the average fitness of CAs generated by PWiseGen.

When a statistical test is performed, two types of errors are possible: (I) we reject the null hypothe-

Table 10. Comparison of time (in seconds) to generate MCA for TCAS

Problem Instance	ACTS (IPOG)	AllPairs	Jenny	TVG	CASA	PWiseGen	ABC-CAG
$10^2 4^1 3^2 2^7$	0.07	0.13	0.545	0.09	0.726	20.34	0.5

Table 11. Comparison of time (in seconds) to generate MCA for the printer case study

Problem Instance	ACTS (IPOG)	AllPairs	Jenny	TVG	CASA	PWiseGen	ABC-CAG
$3^2 2^3 4^1$	0.01	0.016	0.015	0.10	0.269	0.32	0.15

Table 12. Comparison of time (in seconds) to generate CA/MCA for the 20 benchmark problems

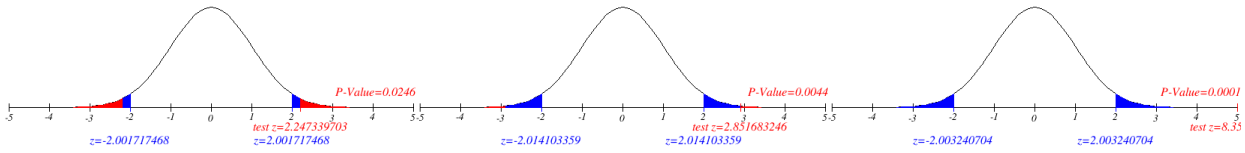
Problem Instance	ACTS (IPOG)	AllPairs	Jenny	CASA	PWiseGen	ABC-CAG
3^3	0.047	0.059	0.034	0.104	0.0546	0.06
3^4	0.002	0.012	0.027	0.151	4.392	0.106
3^{13}	0.014	0.018	0.144	2.23	29.428	70.98
5^{10}	0.003	0.013	0.309	3.608	47.017	123.708
10^{20}	0.53	0.009	2.615	10852.35	675.168	2057.943
2^{100}	0.078	0.071	1.208	4.6985	701.25	1654.17
$4^5 3^4$	0.001	0.012	0.15	0.6785	20.88	37.2
$5^1 3^8 2^2$	0.002	0.013	0.095	0.8145	22.62	47.1
$7^2 6^2 4^2 3^2 2^2$	0.001	0.01	0.141	2.54	135.42	91.2
$8^2 7^2 6^2 5^2$	0.003	0.007	0.183	10.28	50.04	100.5
$6^4 4^5 2^7$	0.006	0.01	0.329	26.36	83.52	133.4
$5^1 4^4 3^{11} 2^5$	0.015	0.016	0.202	3.15	84.12	74.9
$6^1 5^1 4^6 3^8 2^3$	0.016	0.009	0.229	11.2	40.68	78.65
$6^2 4^9 2^9$	0.016	0.009	0.247	2.21	46.38	180.8
$6^5 5^5 3^4$	0.015	0.015	0.249	106.56	87.18	246.87
$7^1 6^1 5^1 4^5 3^8 2^3$	0.016	0.016	0.383	1.44	50.58	66.24
$6^9 4^3 2^7$	0.016	0.015	0.319	7.06	152.82	633.23
$6^7 4^8 2^3$	0.016	0.015	0.295	140.323	135.42	315.34
$4^{15} 3^{17} 2^{29}$	0.031	0.017	0.741	65.8	705.921	1843.56
$4^1 3^{39} 2^{35}$	0.016	0.031	1.96	122.2	752.35	2143.23

sis H_0 when it is true and (II) we accept the null hypothesis H_0 when it is false. These two types of errors are conflicting means minimizing the probability of one increasing the probability of the other. In general there is more emphasis on not committing a type I error. When performing Welch's t-test, the p-value denotes the probability of type I error. The significant level α of a test is the highest p-value that can be accepted to reject H_0. Traditionally, $\alpha = 0.05$ is used during experimentation. We have conducted the Welch t-test at both $\alpha = 0.05$ and $\alpha = 0.01$.The results of the Welch t-test performed to compare average fitness of CAs/MCAs obtained by ABC-CAG and PWiseGen at $\alpha = 0.05$ and at $\alpha = 0.01$ are shown in Table 13. For space reason, we show the graphs depicting the t-values and p-values of only few benchmark problems at $\alpha = 0.05$ and at $\alpha = 0.05$ in Figure 6 and Figure 7, respectively.

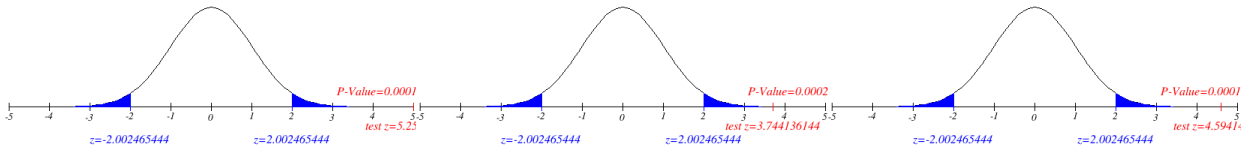
As discussed in Section 6.3.1, the performance of ABC-CAG and CASA is comparable in around 60% cases. So we calculated the standard deviation of the sizes of each CA obtained over 30 runs of TCAS, the printer case study and each of the 20 benchmark problem on both ABC-CAG and CASA to quantify the amount of dispersion or variation of the CA size obtained over these runs on the two meta-heuristic techniques. For

Table 13. Results of the Welch t-test to compare ABC-CAG and PWiseGen

Benchmark Problems	$\alpha = 0.05$				$\alpha = 0.01$			
	p (2-tailed)	t-critical	t-stat	df	p (2-tailed)	t-critical	t-stat	df
3^3	–	–	–	–	–	–	–	–
3^4	–	–	–	–	–	–	–	–
3^{13}	0.028		2.247	58	**0.028**		**2.247**	58
5^{10}	0.006		2.852	45	0.006		2.85	45
10^{20}	2.02×10^{-11}		8.355	56	2.02×10^{-11}		8.35	56
2^{100}	4.4×10^{-05}		4.416	58	4.44×10^{-05}		4.41	58
$4^5 3^4$	2.33×10^{-06}		5.25	57	2.33×10^{-06}		5.25	57
$5^1 3^8 2^2$	4.83×10^{-07}		5.716	54	2.43×10^{-07}		5.71	54
$7^2 6^2 4^2 3^2 2^2$	1.67×10^{-06}		5.33	58	1.67×10^{-06}		5.33	58
$8^2 7^2 6^2 5^2$	1.6×10^{-10}		7.86	54	1.6×10^{-10}		7.86	54
$6^4 4^5 2^7$	5.05×10^{-06}	2.00	5.04	56	5.05×10^{-06}	2.66	5.047	56
$5^1 4^4 3^{11} 2^5$	3.07×10^{-09}		7.24	48	3.07×10^{-09}		7.24	48
$6^1 5^1 4^6 3^8 2^3$	5.9×10^{-04}		3.65	53	5.9×10^{-04}		3.65	53
$6^2 4^9 2^9$	1.79×10^{-05}		4.96	35	1.7×10^{-05}		4.96	35
$6^5 5^5 3^4$	9.7×10^{-09}		6.95	55	4.04×10^{-09}		6.5	56
$7^1 6^1 5^1 4^5 3^8 2^3$	3.49×10^{-08}		6.37	57	3.49×10^{-08}		6.37	57
$6^9 4^3 2^7$	6.9×10^{-08}		6.193	57	6.92×10^{-08}		6.19	57
$6^7 4^8 2^3$	2.3×10^{-07}		5.889	56	2.3×10^{-07}		5.88	56
$4^{15} 3^{17} 2^{29}$	4.2×10^{-04}		3.744	57	0.0004		3.74	57
$4^1 3^{39} 2^{35}$	2.44×10^{-05}		4.59	57	2.44×10^{-05}		4.59	57



a) CA(3^{13}) b) CA(5^{10}) c) CA(10^{20})



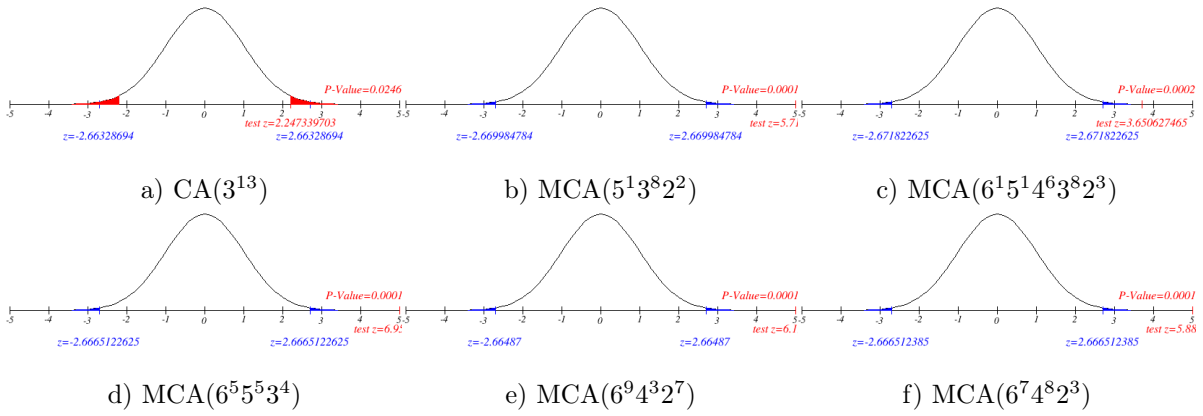
d) MCA($4^5 3^4$) e) MCA($4^{15} 3^{17} 2^{29}$) f) MCA($4^1 3^{39} 2^{35}$)

Figure 6. Results of the Welch t-test of the selected benchmark problems at $\alpha = 0.05$

TCAS and the printer case study, the size of CA generated in each of the 30 runs on ABC-CAG is the same. The same is true for CASA as well. Therefore, we plotted the average standard deviation of only the benchmark problems given in Table 6 when run multiple times on ABC-CAG and CASA as shown in Figure 8.

Analysis: It is evident from Table 13 and Figure 6 that at $\alpha = 0.05$, the value of t-stat $>$ t-critical and $p < \alpha$ for each benchmark problem. Similarly, from Table 13 and Figure 7, it can be seen that at $\alpha = 0.01$, the value of t-stat $>$ t-critical and $p < \alpha$ for each benchmark problem except CA (3^{13}). So, we reject the null hypothesis H_0 and conclude that there is a significant difference between ABC-CAG and PWiseGen.

From Figure 8, it can be seen that for most of the benchmark problems, the average standard deviation of the sizes of CA/MCA obtained over multiple runs of a problem instance on ABC-CAG is smaller than that of CASA. Even in the case of those problems where the best sizes generated by both ABC-CAG and CASA are the same, the standard deviation of ABC-CAG is smaller than



a) CA(3^{13}) b) MCA($5^1 3^8 2^2$) c) MCA($6^1 5^1 4^6 3^8 2^3$)

d) MCA($6^5 5^5 3^4$) e) MCA($6^9 4^3 2^7$) f) MCA($6^7 4^8 2^3$)

Figure 7. Results of the Welch t-test of the Selected Benchmark Problems at $\alpha = 0.01$

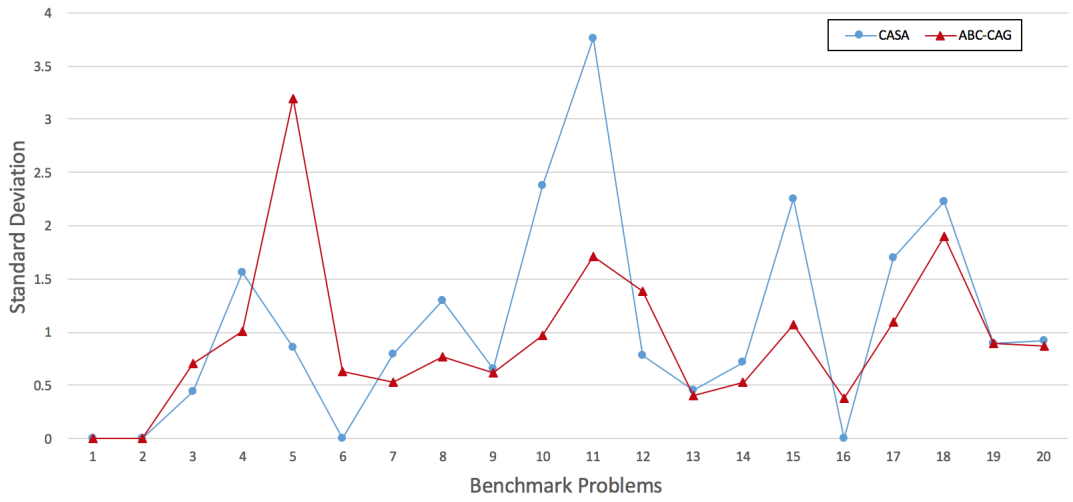


Figure 8. Average standard deviation of sizes of CAs obtained over multiple runs

CASA for 8 problems out of 10. On this basis it can be inferred that ABC-CAG is more stable as compared to CASA.

In summary, we can conclude that ABC-CAG performs better than greedy algorithms and most of the meta-heuristic techniques except TS where the results are comparable, and the outcome of statistical testing proves the validity of the results generated by ABC-CAG and the standard deviation shows that ABC-CAG is more stable as compared to CASA when run multiple times on the same problem instance.

7. Threats to Validity

An important threat to validity is that we use only 30 runs of the stochastic algorithms namely ABC-CAG, CASA and PWiseGen because of time and resource constraints. Though more runs are unlikely to change the qualitative answer to the research questions, they may affect the magnitude of the algorithmic differences.

8. Conclusion and Future Work

In this paper we have presented the Artificial Bee Colony – Covering Array Generator (ABC-CAG) that deals with the problem of constructing optimal covering arrays for pair-wise testing. The key feature of ABC-CAG is the integration of greedy and meta-heuristic algorithms which enable ABC-CAG to exploit the advantages of both techniques. Second, the use of the global best CA during onlooker bees' phase derives the solution

towards global best thereby improving the exploitation capability of onlooker bees. In addition to this, the use of smart test cases to replace the worst test case of the global best CA during the onlooker bees' phase further enhances the performance of ABC-CAG. Experiments conducted on various benchmark problems and a real world problem show that the proposed strategy generates smaller CA as compared to its greedy and meta-heuristic counterparts except TS where the results are comparable.

In future, we plan to construct CA for strength t greater than 2 and incorporate constraint handling feature in ABC-CAG.

References

[1] J.M. Glenford, *The art of software testing*. John Willey & Sons, 2011.

[2] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, "The automatic efficient test generator (AETG) system," in *Proceedings of the 5th International Symposium on Software Reliability Engineering*. IEEE, 1994, pp. 303–309.

[3] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, Vol. 23, No. 7, 1997, pp. 437–444.

[4] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in *Proceedings of the International Conference on Software Testing Analysis & Review*, San Diego, 1998.

[5] S.R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*. ACM, 1999, pp. 285–294.

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

[7] Y. Lei and K.C. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*. IEEE, 1998, pp. 254–261.

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

[9] A. Hartman, T. Klinger, and L. Raskin, "IBM intelligent test case handler," *Discrete Mathematics*, Vol. 284, 2010, pp. 149–156.

[10] J. Arshem, Test vector generator (TVG), (2010). [Online]. https://sourceforge.net/projects/tvg/

[11] AllPairs, (2009). [Online]. http://sourceforge.net/projects/allpairs/

[12] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of the 24th Pacific Northwest Software Quality Conference*, 2006, pp. 419–430.

[13] B. Jenkins, jenny: a pairwise testing tool, (2005). [Online]. http://burtleburtle.net/bob/math/jenny.html

[14] Z. Wang, B. Xu, and C. Nie, "Greedy heuristic algorithms to generate variable strength combinatorial test suite," in *The Eighth International Conference on Quality Software. QSIC'08*. IEEE, 2008, pp. 155–160.

[15] Z. Wang and H. He, "Generating variable strength covering array for combinatorial software testing with greedy strategy," *Journal of Software*, Vol. 8, No. 12, 2013, pp. 3173–3181.

[16] S.A. Abdullah, Z.H. Soh, and K.Z. Zamli, "Variable-strength interaction for t-way test generation strategy," *International Journal of Advances in Soft Computing & Its Applications*, Vol. 5, No. 3, 2013.

[17] M.F. Klaib, K.Z. Zamli, N.A.M. Isa, M.I. Younis, and R. Abdullah, "G2Way a backtracking strategy for pairwise test data generation," in *15th Asia-Pacific Software Engineering Conference, APSEC'08*. IEEE, 2008, pp. 463–470.

[18] K.Z. Zamli, M.F. Klaib, M.I. Younis, N.A.M. Isa, and R. Abdullah, "Design and implementation of a t-way test data generation strategy with automated execution tool support," *Information Sciences*, Vol. 181, No. 9, 2011, pp. 1741–1758.

[19] K.F. Rabbi, A.H. Beg, and T. Herawan, "MT2Way: A novel strategy for pair-wise test data generation," in *Computational Intelligence and Intelligent Systems*. Springer, 2012, pp. 180–191.

[20] K. Rabbi, S. Khatun, C.Y. Yaakub, and M. Klaib, "EPS2Way: an efficient pairwise test data generation strategy," *International Journal of New Computer Architectures and their Applications (IJNCAA)*, Vol. 1, No. 4, 2011, pp. 1099–1109.

[21] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinato-

rial optimization," *Journal of Systems and Software*, Vol. 98, 2014, pp. 191–207.

[22] R. Kuhn, Advanced combinatorial testing system (ACTS), National Institute of Standards and Technology, (2011). [Online]. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html#acts

[23] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference*. IEEE, 2004, pp. 72–77.

[24] X. Chen, Q. Gu, J. Qi, and D. Chen, "Applying particle swarm optimization to pairwise testing," in *IEEEProceedings of the 34th Annual Computer Software and Applications Conference*. IEEE, 2010, pp. 107–116.

[25] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *Combinatorial Optimization and Applications*. Springer, 2010, pp. 51–64.

[26] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of strengths 2 through 6 using a tabu search approach," *Discrete Mathematics, Algorithms and Applications*, Vol. 4, No. 03, 2012, p. 1250033.

[27] H. Avila-George, J. Torres-Jimenez, V. Hernández, and L. Gonzalez-Hernandez, "Simulated annealing for constructing mixed covering arrays," in *Distributed Computing and Artificial Intelligence*. Springer, 2012, pp. 657–664.

[28] B.S. Ahmed, K.Z. Zamli, and C. Lim, "The development of a particle swarm based optimization strategy for pairwise testing," *Journal of Artificial Intelligence*, Vol. 4, No. 2, 2011, pp. 156–165.

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

[30] J.D. McCaffrey, "Generation of pairwise test sets using a genetic algorithm," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1. IEEE, 2009, pp. 626–631.

[31] P. Flores and Y. Cheon, "PWiseGen: Generating test cases for pairwise testing using genetic algorithms," in *Proceedings of the International Conference on Computer Science and Automation Engineering*, Vol. 2. IEEE, 2011, pp. 747–752.

[32] P. Bansal, S. Sabharwal, S. Malik, V. Arora, and V. Kumar, "An approach to test set generation for pair-wise testing using genetic algorithms," in *Search Based Software Engineering*. Springer, 2013, pp. 294–299.

[33] B.S. Ahmed, T.S. Abdulsamad, and M.Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," *Information and Software Technology*, Vol. 66, 2015, pp. 13–29.

[34] T. Mahmoud and B.S. Ahmed, "An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use," *Expert Systems with Applications*, Vol. 42, No. 22, 2015, pp. 8753–8765.

[35] D. Karaboga, "An idea based on honey bee swarm for numerical optimization," Erciyes University, Engineering Faculty, Computer Engineering Department, Tech. Rep. TR-06, 2005.

[36] D. Karaboga and B. Basturk, "Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems," in *Foundations of Fuzzy Logic and Soft Computing*. Springer, 2007, pp. 789–798.

[37] A.S. Hedayat, N.J.A. Sloane, and J. Stufken, *Orthogonal arrays*. Springer Science & Business Media, 2012.

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

[39] G. Sherwood, Testcover.com, (2006). [Online]. http://testcover.com/

[40] A.W. Williams, "Determination of test configurations for pair-wise interaction coverage," in *Testing of Communicating Systems*. Springer, 2000, pp. 59–74.

[41] A. Hartman, "Software and hardware testing using combinatorial covering suites," in *Graph theory, combinatorics and algorithms*. Springer, 2005, pp. 237–266.

[42] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "A new method for constructing pair-wise covering designs for software testing," *Information Processing Letters*, Vol. 81, No. 2, 2002, pp. 85–91.

[43] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm," *Journal of global optimization*, Vol. 39, No. 3, 2007, pp. 459–471.

[44] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm intelligence: from natural to artificial*

systems. Oxford University Press, 1999.

[45] O.B. Haddad, A. Afshar, and M.A. Mariño, "Honey-bees mating optimization (HBMO) algorithm: a new heuristic approach for water resources optimization," *Water Resources Management*, Vol. 20, No. 5, 2006, pp. 661–680.

[46] D. Teodorović and M. Dell'Orco, "Bee colony optimization – a cooperative learning approach to complex transportation problems," in *Advanced OR and AI Methods in Transportation: Proceedings of 16th Mini–EURO Conference and 10th Meeting of EWGT*. Poznań: Publishing House of the Polish Operational and System Research, 2005, pp. 51–60.

[47] H. Drias, S. Sadeg, and S. Yahi, "Cooperative bees swarm for solving the maximum weighted satisfiability problem," in *Computational Intelligence and Bioinspired Systems*. Springer, 2005, pp. 318–325.

[48] G. Li, P. Niu, and X. Xiao, "Development and investigation of efficient artificial bee colony algorithm for numerical function optimization," *Applied soft computing*, Vol. 12, No. 1, 2012, pp. 320–332.

[49] D. Jeya Mala, V. Mohan, and M. Kamalapriya, "Automated software test optimisation framework – an artificial bee colony optimisation-based approach," *IET Software*, Vol. 4, No. 5, 2010, pp. 334–348.

[50] G. Zhu and S. Kwong, "Gbest-guided artificial bee colony algorithm for numerical function optimization," *Applied Mathematics and Computation*, Vol. 217, No. 7, 2010, pp. 3166–3173.

[51] J. Stardom, "Metaheuristics and the search for covering and packing arrays," Ph.D. dissertation, Simon Fraser University, 2001.

[52] B. Kazimipour, X. Li, and A. Qin, "A review of population initialization techniques for evolutionary algorithms," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2014, pp. 2585–2592.

[53] D.R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*. IEEE, 2006, pp. 153–158.

[54] Pairwise testing, (2016). [Online]. http://www.pairwise.org/

[55] P. Flores, PWiseGen, (2010). [Online]. https://code.google.com/p/pwisegen/

[56] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*. IEEE, 2011, pp. 1–10.