e-Informatica Software Engineering Journal, Volume 9, Issue 1, 2015, pages: 87–105, DOI 10.5277/E-INF150107

Construction of Variable Strength Covering Array for Combinatorial Testing Using a Greedy Approach to Genetic Algorithm

Priti Bansal*, Sangeeta Sabharwal*, Nitish Mittal*, Sarthak Arora**
*Netaji Subhas Institute of Technology, University of Delhi
**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

The limitation of time and budget usually prohibits exhaustive testing of interactions between components in a component based software system. Combinatorial testing is a software testing technique that can be used to detect faults in a component based software system caused by the interactions of components in an effective and efficient way. Most of the research in the field of combinatorial testing till now has focused on the construction of optimal covering array (CA) of fixed strength t which covers all t-way interactions among components. The size of CA increases with the increase in strength of testing t, which further increases the cost of testing. However, not all components require higher strength interaction testing. Hence, in a system with k components a technique is required to construct CA of fixed strength t which covers all t-way interactions among k components and all t_i-way (where $t_i > t$) interactions between a subset of k components. This is achieved using the variable strength covering array (VSCA). In this paper we propose a greedy based genetic algorithm (GA) to generate optimal VSCA. Experiments are conducted on several benchmark configurations to evaluate the effectiveness of the proposed approach.

Keywords: combinatorial testing, variable strength covering array, genetic algorithm, greedy approach

1. Introduction

The increasing dependence on software systems in every field, such as medicine, agriculture, communication systems has increased the need to perform software testing in an effective and efficient manner so as to ensure the delivery of reliable and quality software. In the case of a component based software system, interactions among components are often complex and they may cause interaction errors. It is therefore important to check all the possible interactions among various components to uncover faults caused by their interactions. As each component may have multiple configurations, testing all possible combinations of components is practically impossible due to time and cost constraints. Furthermore, the number of test cases increases exponentially with the increase in number of components. A sampling strategy is therefore required to select a subset of configurations to be tested from the large interaction space. Combinatorial testing is a testing technique that samples the set of configurations in such a way that it covers all t-way (t denotes the strength of testing) interactions of components [1].

Covering arrays (CAs) and mixed covering arrays (MCAs) are combinatorial structures that have enjoyed a wide range of application in the field of software and hardware testing [2]. Due to the importance of CAs, significant research has been carried out to construct CAs of optimal size by the researchers in the past. A CA constructed to perform t-way (2-way, 3-way, etc.) testing checks only all t-way interactions of components. Empirical studies show that a test set covering all possible 2-way combinations of input parameter values is effective for software systems [1, 3–5]. Dalal et al. [6] showed that testing all pair-wise interactions in a software system finds a large percentage of the existing faults. Kuhn et al. [7] examined fault reports for many software systems and concluded that more than 70% of the faults are triggered by a 2-way interaction of the input parameters. Faults can also be caused by the interaction of more than two parameters. In order to uncover faults caused by the interaction of more than two components, it is required to test higher strength interactions of components. Empirical studies in Kuhn et al. [7] and Kuhn and Reilly [8] show that most of the faults are triggered by a relatively low degree of interactions and suggest the need to perform testing up to $t = 6$.

Consider a Graphical User Interface (GUI) based on a windowing system which has five components, each with three possible values as shown in Table 1. For exhaustively testing the components' interactions in this system, 243 test cases are required whereas only 11 test cases for 2-way testing and 37 test cases for 3-way testing are required respectively. Evidently, the increase in strength of testing leads to the increase in number of test cases. However, it is quite often the case that certain components have stronger interactions while others may have few or none [9]. Hence, it is not desirable to perform higher strength interaction testing of all the components. A better way to test the system is to identify the subsets of components which are involved in stronger interactions and apply higher strength interaction testing only on these subsets to uncover the faults caused by their interactions. This is achieved using the variable strength covering array (VSCA), which is a CA or MCA of fixed strength t and also contains a set of disjoint CAs or MCAs of strength greater than t. As mentioned above, the example shown in Table 1 requires 11 test cases for 2-way testing. Assume, first four components have stronger interactions compared to the fifth component. So it is feasible to perform 3-way testing only on the first four components, which additionally requires 16 test cases as illustrated in Figure 1 and Figure 2. Consequently, a total of 27 test cases are required for variable strength testing against 37 test cases required for a complete 3-way testing. We can see that VSCA achieves higher strength interaction coverage with the reduced number of test cases. So it is advantageous to find an effective technique to construct optimal VSCA to perform testing of a component based system efficiently.

Kernel	**DS**	**WM**	**DSCP**	**GI**
FreeBSD	Weston	Awesome	Wayland	KDE Plasma
FreeBSD	X.Org	Compiz	X11	Aqua
XNU	X.Org	OpenBox	Wayland	KDE Plasma
XNU	KWin	Awesome	X11	KDE Plasma
XNU	Weston	Compiz	X11	Gnome Shell
Linux	KWin	Compiz	Wayland	Aqua
XNU	Weston	Awesome	Quartz	Aqua
Linux	X.Org	Compiz	Wayland	KDE Plasma
Linux	X.Org	Awesome	Wayland	Gnome Shell
FreeBSD	KWin	OpenBox	Quartz	Gnome Shell
Linux	Weston	OpenBox	X11	Aqua

Figure 1. CA (11, 2, 3^5)

Kernel	**DS**	**WM**	**DSCP**	**GI**
FreeBsd	X.Org	Compiz	Quartz	Aqua
XNU	Kwin	Awesome	Quartz	KDE Plasma
FreeBsd	Weston	Compiz	Wayland	KDE Plasma
Linux	Weston	Compiz	X11	Gnome Shell
Linux	Kwin	OpenBox	Wayland	KDE Plasma
FreeBsd	X.Org	Awesome	X11	Aqua
FreeBsd	Kwin	Compiz	X11	KDE Plasma
XNU	X.Org	OpenBox	Quartz	Gnome Shell
FreeBsd	Weston	Awesome	Quartz	Gnome Shell
FreeBsd	Kwin	Awesome	Wayland	Gnome Shell
XNU	Weston	Awesome	X11	Gnome Shell
FreeBsd	X.Org	OpenBox	Wayland	Gnome Shell
Linux	X.Org	OpenBox	X11	KDE Plasma
XNU	Weston	OpenBox	Wayland	Aqua
XNU	Weston	Compiz	Quartz	Gnome Shell
Linux	Kwin	Awesome	X11	Aqua
Linux	Weston	Awesome	Wayland	KDE Plasma
XNU	X.Org	Awesome	Wayland	Gnome Shell
Linux	X.Org	Compiz	Wayland	Gnome Shell
XNU	X.Org	Compiz	X11	Gnome Shell
XNU	Kwin	OpenBox	X11	Aqua
Linux	Weston	OpenBox	Quartz	Aqua
XNU	Kwin	Compiz	Wayland	Gnome Shell
Linux	X.Org	Awesome	Quartz	Gnome Shell
FreeBsd	Weston	OpenBox	X11	Gnome Shell
FreeBsd	Kwin	OpenBox	Quartz	Aqua
Linux	Kwin	Compiz	Quartz	KDE Plasma

Figure 2. VSCA (27; 2, 3^5, (3, 3^4))

Table 1. GUI based on a windowing system having five components, each with three values

Kernel	Display Server (DS)	Window Manager (WM)	Display Server Communication Protocol (DSCP)	Graphical Interface (GI)
Linux	Weston	Awesome	X11	KDE Plasma
FreeBSD	KWin	Compiz	Wayland	Aqua
XNU	X.Org	OpenBox	Quartz	Gnome Shell

The problem of constructing an optimal VSCA is NP-complete [10, 11]. Although many algebraic and computational construction methods have been proposed by the researchers to construct optimal CA/MCA, fewer strategies (greedy and meta-heuristic) exist to construct optimal VSCA. The amount of work that has been done to construct VSCA using meta-heuristic techniques such as Simulated Annealing (SA), Particle Swarm Optimization (PSO), Harmony Search (HS) and their impressive results has motivated us to explore GA to construct optimal VSCA.

To exploit the strength of both greedy and meta-heuristic techniques we present a technique that augments GA with a greedy technique to construct optimal VSCA efficiently. Experiments are conducted to evaluate the performance of the proposed technique with the existing techniques.

However, the problem that exists with the construction of VSCA is the existence of constraints or dependencies between components values in terms of restrictions or compulsion on components values that can coexist. For instance, in the example shown in Table 1, Quartz is a Mac technology and therefore cannot be run on Linux or FreeBSD. This constraint must be taken into account when generating test cases so that Quartz and Linux/FreeBSD do not appear in the same test case. Similarly, KDE Plasma and XNU cannot appear in the same test case as XNU does not support KDE Plasma. If constraints and dependencies are considered, then combinatorial testing becomes constrained combinatorial testing. In this paper, we focus on combinatorial testing and leave constrained combinatorial testing for future work.

The remainder of this paper is organized as follows. Section 2 gives the necessary background on combinatorial objects. Section 3 gives an overview of GA. Section 4 presents various methods available to construct VSCA. Section 5 describes the proposed strategy to generate VSCA for t-way testing. Section 6 describes the implementation and presents result of experiments performed to compare the effectiveness of the proposed approach with other existing approaches. Section 7 presents threats to validity. Section 8 concludes the paper and future plans are outlined.

2. Background

This section discusses the necessary background related to combinatorial objects.

2.1. Orthogonal Array

An orthogonal array $\mathrm{OA}_\lambda(N; t, k, v)$ is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols exactly λ times and they have the property $\lambda = N/v^t$ [12]. The use of OA in the field of software testing is limited due to the restrictions imposed on OA that all parameters have same number of values and that each pair of values can be covered the same number of times [13]. In general, OA is difficult to generate and its test suite is often quite large with $\lambda > 1$. However, OA has its advantages, such as making it relatively easy to identify the particular combination that caused a failure [11]. If an OA with $\lambda = 1$ exists for some value of k and v, then it is an optimal array. To complement OA construction and to overcome its restrictions, CA and MCA have been introduced.

2.2. Covering Array

A covering array [12] denoted by $CA_\lambda(N;t,k,v)$, 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 $CA(N;t,k,v)$. Here, k represents the number of values of each parameter and t is the strength of testing. An optimal CA contains a minimum number of rows to satisfy the properties of the entire CA. The minimum number of rows is known as covering array number and is denoted by $CAN(t,k,v)$. A CA of size $N \times k$ represents a test set where each row corresponds to a test case, each column represents a component and the values in the column represent the domain of the respective component.

2.3. Mixed Covering Array

A mixed covering array [14], denoted by $MCA(N;t,k,(v_1,v_2,\ldots,v_k))$, 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 $(1 \le i \le k)$ 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 $MCAN(t,k,(v_1,v_2,\ldots,v_k))$. A shorthand notation can be used to represent MCAs by combining equal entries in $v_i : 1 \le i \le k$. An $MCA(N;t,k,(v_1,v_2,\ldots,v_k))$ can be represented as $MCA(N;t,k,(w_1^{q_1},w_2^{q_2},\ldots,w_s^{q_s}))$, where $k = \sum_{i=1}^{s} q_i$ and $w_j|1 \le j \le s \subseteq \{v_1,v_2,\ldots,v_k\}$. 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 varying domain size.

2.4. Variable Strength Covering Array

A variable strength covering array [9], denoted by $VSCA(N;t,k,(v_1,v_2,\ldots,v_k),C)$, is an $N \times k$ CA or MCA of strength-t containing C where, C is a set of disjoint CAs or MCAs each of strength greater than t. Each element of C is a subset of VSCA and they can have variable strength of testing. For example, a $VSCA(N;2,4^3 5^3 6^2,\{CA(3,4^3),MCA(4,5^3 6^1)\})$ is shown in Fig. 3(a). Here, the overall array is an MCA having three components with four values, three with five values and two with six values each (the values of each component are labelled $0,1,2,3,\ldots$). It covers all 2-way interactions among components. In addition to this, it contains two sub arrays: a CA of strength-3 that covers all 3-way interactions among first three components with four values and an MCA of strength-4 covering all 4-way interactions among three components with five values and one component with six values. The order of columns in the sub arrays in C is important and they are listed consecutively from left to right. If a VSCA contains n identical sub arrays with the same t, k and v, they can be represented as $CA(t,v^k)^n$. For instance, $VSCA(N,2,3^{11},(3,3^4)^2)$ shown in Figure 3(b) represent a CA that covers all 2-way interactions among eleven components with three values and contains two disjoint sub arrays, each of which covers all 3-way interactions among four components with three values each.

3. Genetic Algorithm

The basics of GA were first proposed by Holland [15]. GA is a meta-heuristic search based optimization technique originating from the Darwinian theory of evolution by natural selection where fitter individuals are more likely to survive in a competing environment [16]. It is a global search technique characterized by evolution in every generation, starting with a randomly generated initial population. The initial population represents potential solutions to the given problem. Each individual in the population is associated with a fitness value that is calculated using a fitness function. The fitness function is a function of the objective that we want to optimize (maximize or minimize). The fitness value of an individual apprises us of the merit of a solution

$\overbrace{\hspace{10em}}^{\mathrm{MCA}(N;2,4^3 5^3 6^2)}$

0	1	0	1	0	1	1	0
1	0	1	0	1	2	3	1
1	2	3	1	4	3	0	1
2	3	2	4	2	3	2	2
0	2	1	2	3	0	1	5
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
3	0	0	3	4	4	5	3
0	3	3	3	2	2	4	4

$\underbrace{\hspace{4em}}_{\mathrm{CA}(3,4^3)}$ (columns 1–3) $\underbrace{\hspace{5em}}_{\mathrm{MCA}(4,5^3 6^1)}$ (columns 4–7)

(a)

$\overbrace{\hspace{14em}}^{\mathrm{CA}(N;2,3^1 1)}$

0	0	1	1	1	0	0	1	0	1	0
1	0	0	0	1	0	1	2	1	0	2
2	1	2	0	0	1	0	1	2	1	1
1	2	1	2	1	2	2	0	0	2	1
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...
0	1	0	0	0	1	2	1	1	2	0
1	1	0	2	2	2	1	0	0	1	2

$\underbrace{\hspace{5em}}_{\mathrm{CA}(3,3^4)}$ (columns 1–4) $\underbrace{\hspace{5em}}_{\mathrm{CA}(3,3^4)}$ (columns 5–8)

(b)

Figure 3. Representation of VSCA

for the given problem. In each generation, the population evolves towards better solutions by means of evolutionary operators such as selection, crossover and mutation. This process continues until a satisfactory solution is found or the maximum number of generations is reached. As compared to other meta-heuristic techniques, GA starts with a population of solutions instead of a single solution that helps GA to cover the solution search space more thoroughly and avoid its chances of getting stuck in the local minima. Moreover, GA is easy to understand and can be applied to an optimization problem which can be described with chromosome encoding. On the contrary, the complexity of crossover and mutation operations is attributed to longer run time and, furthermore, GA cannot assure a constant optimization response time which limits its use in real time applications. The basic outline of GA is shown in Figure 4.

Having described the notations, in the next section we will briefly discuss the existing state-of-the-art algorithms for constructing optimal VSCA for pair-wise testing.

4. Related Work

In an extensive survey performed by Khalsa and Labiche in [17], it has been found that 75 tools/algorithms exist to generate CA/MCA for combinatorial testing but not all of them support construction of VSCA. The strategies that support the construction of VSCA are broadly classified into computational strategies and artificial intelligence based strategies. Computational strategies use greedy approach to construct VSCA by using either one-test-at-a-time or one-parameter-at-a-time approach. The strategies based on one-test-at-a-time approach use a greedy heuristic and try to select a test case that covers the maximum number of uncovered interactions from a pool of candidate test cases. However, selecting such a test case itself is an NP-complete problem [18]. Some of the strategies that use one-test-at-a-time approach are the Test Vector Generator (TVG) [19], Pairwise Independent Combinatorial Testing (PICT) [20], Intelligent Test Case Handler[1] (ITCH), Density [21], DA-RO and DA-FO [22], and TSG [23]. The strategies that use one-parameter-at-a-time approach are ACTS [24, 25] and ParaOrder [21].

Recently the search based software testing (SBST) is increasingly gaining importance and is been used to solve a wide range of problems in software testing. Meta-heuristic techniques are being used by the SBST community to find an optimal solution for software testing problems. The problem of generating an optimal CA/MCA/VSCA is also considered as a SBST problem [26, 27]. Meta-heuristic techniques start by searching over a large set of feasible solutions and can often find better solutions with

[1] IBM Intelligent Test Case Handler

```
generate an initial population P randomly
evaluate fitness of each individual in P using the fitness function
while ((generation ≤ maximum generation) and solution not found)
   select a subset of individuals P’ from current generation for offspring production
   apply crossover to P’
   apply mutation to P’
   replace low fitness individuals in P by offspring in P’
   evaluate P
end while
return individual with highest fitness
```

Figure 4. Outline of basic GA

fewer computational efforts efforts as compared to other algorithms, iterative methods or simple heuristics [28]. To the best of our knowledge, only five meta-heuristic based strategies to generate VSCA exist in the literature. Table 2 list features of all tools/algorithms that use meta-heuristic techniques and some selected tools that use greedy techniques to construct CA/MCA/VSCA.

5. The Proposed Approach for Construction of VSCA

In this section, we present our proposed strategy of VSCA-GA that aims to generate an optimal VSCA to cover all (100%) t-way and t_i-way interactions between components in a component based system. Here, t denotes the overall strength of VSCA and t_i denotes the strength of sub arrays. VSCA-GA uses a greedy based approach to GA to generate optimal VSCA. Let $\text{VSCA}(N; t, k, (v_1, v_2, \ldots, v_k), C)$ represent a VSCA configuration. VSCA-GA starts by creating an initial population of P_{size} individuals where each individual chromosome represents a candidate solution which is a VSCA of size $N \times k$. Here, N the number of rows of VSCA corresponds to test cases and k represents the number of components in a component based system. At the start of the search process N is unknown, so we use the method suggested by Stardom [39], where we start with a large random array and apply binary search repeatedly until a solution is found. In case the size of N is known in advance, i.e. best bound achieved in the existing state-of-the-art, we can start with the known size and try to optimize it further. An individual chromosome in the population is represented by $\text{VSCA}_f | 1 \leq f \leq P_{\text{size}}$ and each VSCA_f in the population has a fitness associated with it which is defined as the total number of distinct variable strength interactions covered by it. It is calculated as

$$\text{Fitness}(\text{VSCA}_f) = \sum_{i=t,t_1,\ldots,t_n} \text{number of distinct } i\text{-way interactions covered by } \text{VSCA}_f \quad (1)$$

Where, t is the overall strength of VSCA, $t_1, t_2, \ldots, t_n$ are the strength of sub-arrays.

After initialization, GA searches the solution space by applying genetic operators such as selection, crossover and mutation repeatedly to find the best solution. The process continues until a solution is found or the maximum number of iterations is reached. In case VSCA-GA starts by taking N from the existing literature and a solution is found at this N, then the size of VSCA is decreased by one, otherwise the size of VSCA is increased by one and VSCA-GA is executed again in both cases. When VSCA-GA is re-executed, we seed the initial population by supplying the best VSCA generated in the previous run. If the size of VSCA in the current run is less than that in the previous run, we decrease the size of seeded VSCA by one by removing the test case that contributes least to the fitness of VSCA whereas, if the size of VSCA in the current run is greater than the size in the previous run, then we add a randomly generated test case to the existing VSCA. The various steps of VSCA-GA strategy are explained below.

Table 2. Comparison of various tools/algorithms for constructing CA/MCA/VSCA for CIT

S. No.	Tool / Algorithm	Variable Strength	Maximum Strength Support(t)	Technique Employed		Test Generation Strategy	Constraint Handling
1	AETG [1]	✗	2				✓
2	ITCH[1]	✓	6				✓
3	TVG [19]	✓	6				✓
4	PICT [20]	✓	6			One Test at a Time	✓
5	Density [21]	✓	3				✗
6	DA-RO [22]	✓	3	Greedy			✗
7	DA-FO [22]	✓	3				✗
8	TSG [23]	✓	3				✗
9	Jenny [29]	✗	8				✓
10	ACTS (IPOG) [24, 25]	✓	6			One Parameter at a Time	✓
11	ParaOrder [21]	✓	3				✗
12	SA [9]	✓	3		Simulated Annealing		✗
13	GA [30]	✗	3		Genetic Algorithm		✗
14	ACA [30]	✗	3		Ant Colony Optimization		✗
15	ACS [31]	✓	3		Ant Colony Optimization		✗
16	TSA [32]	✗	6		Tabu Search		✗
17	GAPTS [33]	✗	2		Genetic Algorithm	One Test at a Time	✗
18	PWiseGen [34]	✗	2	Meta-heuristic	Genetic Algorithm		✗
19	VS-PSTG [35]	✓	6		Particle Swarm Optimization		✗
20	HSS [36]	✓	15		Harmony Search		✓
21	HSTCG [37]	✓	7		Harmony Search		✓
22	CASA [27]	✗	3		Simulated Annealing		✓
23	PSO [38]	✗	2		Particle Swarm Optimization	One Parameter at a Time	✗
24	PSO [38]	✗	2		Particle Swarm Optimization		✗

5.1. A Greedy Approach to Generate Initial Population

When GA is used to construct VSCA, the role of initial population on the performance of GA cannot be ignored as it can affect the convergence speed and quality of the final solution [40, 41]. Generally, initial population is generated randomly. However, recognizing the effect of initial population on GA performance, several population initialization methods for GA have been proposed in the past by the researchers [41–46]. Here, we present a greedy approach for generating a good quality initial population of VSCA, which is achieved by focusing on the coverage of maximum number of possible uncovered interactions.

Let us consider the system under test consisting of k components where a component is represented by $C_m | 1 \leq m \leq k$ and each component C_m can take values from 0 to $(v_m - 1)$ (v_m is the number of possible values of component C_m). The $j^{\text{th}} | 1 \leq j \leq v_m$ value of component C_m is represented by val_{mj}. To generate an initial population of VSCAs, VSCA-GA starts by computing and storing all the possible t-way and t_i-way interactions between the values of all the components in an interaction list L, based on the configuration of VSCA. Then, it calculates the number of uncovered interactions of each value val_{mj} of every component C_m, stores it in a variable $N_{\text{uncovered}}(\text{val}_{mj})$ and assigns a probability of selection denoted by $P(\text{val}_{mj})$ to each of them. The probability of selection assigned to a value val_{mj} of component C_m represents its chances of getting selected when a test case is created. In our case, the probability assigned to a value v_{mj} of component C_m depends upon the number of uncovered interactions of val_{mj} as well as the total number of uncovered interactions of component C_m. Initially all values val_{mj} of a component C_m are involved in an equal number of uncovered interactions, therefore each of them will have an equal probability of getting selected. For instance, if a component has four possible values then initially each one of them will have the probability of selection equal to 0.25. VSCA-GA generates the first test case tc_{f1} in $\text{VSCA}_f | 1 \leq f \leq P_{\text{size}}$ by selecting a value of each component randomly as each one of them have an equal probability of selection. After the generation of first test case, VSCA-GA updates the interaction list L by eliminating interactions that are covered in tc_{f1}. Let the value val_{ms} of component C_m be selected in tc_{f1} and the number of interactions covered by val_{ms} in tc_{f1} is $N_{\text{covered}}(\text{val}_{ms})$, then the number of interactions of val_{ms} left uncovered is denoted by $N'_{\text{uncovered}}(\text{val}_{ms})$ and is calculated as:

$$N'_{\text{uncovered}}(\text{val}_{ms}) = N_{\text{uncovered}}(\text{val}_{ms}) - N_{\text{covered}}(\text{val}_{ms}) \quad (2)$$

Let $P_{old}(\text{val}_{ms})$ denote the probability of selection of value val_{ms} before selection, then after selection the probability of val_{ms} becomes:

$$P_{\text{new}}(\text{val}_{ms}) = P_{\text{old}}(\text{val}_{ms}) \times \frac{N'_{\text{uncovered}}(\text{val}_{ms})}{N_{\text{uncovered}}(\text{val}_{ms})} \quad (3)$$

The decrease in the probability of value val_{ms} is calculated using Equation 4 and is distributed among the remaining values $\text{val}_{mj} | 1 \leq j \leq v_m$ and $j \neq s$ of component C_m according to Equation 5.

$$P_{\text{decrement}}(\text{val}_{ms}) = P_{\text{old}}(\text{val}_{ms}) \times \left(1 - \frac{N'_{\text{uncovered}}(\text{val}_{ms})}{N_{\text{uncovered}}(\text{val}_{ms})}\right) \quad (4)$$

$$P_{\text{new}}(\text{val}_{mj}) = P_{\text{old}}(\text{val}_{mj}) + \left(\frac{N_{\text{uncovered}}(\text{val}_{mj})}{\sum_{j=1\ to\ v_m}^{j \neq s} N_{\text{uncovered}}(\text{val}_{mj})} \times P_{decrement}(\text{val}_{ms})\right) \quad (5)$$

Equation 5 increases the probability of the value val_{mj} of component C_m based on the number of its uncovered interactions and the total number of uncovered interactions of the remaining values (except val_{ms}) of component C_m. Hence, the higher the number of remaining uncovered interactions of a value, the higher will be the increase in its probability and vice versa, thereby getting greedy by extending a higher opportunity of selection to the values with maximum uncovered interactions. Once the probability of each value of every component is updated, the number of uncovered interactions of the selected value $\text{val}_{mj}\, \forall m$ is updated by assigning the value of $N'_{\text{uncovered}}(\text{val}_{ms})$ to $N_{\text{uncovered}}(\text{val}_{ms})$. The succeeding test cases $tc_{fi} | 2 \leq i \leq N$ are generated by selecting a value for each component based on the probabilities that are updated after the generation of every test case. Since each value val_{mj} of a component C_m may have different probability of selection, to select a value of a component, a random number is generated in the range $[0, 1]$ and based on the interval in which the random number falls; the value val_{mj} of the component C_m is selected. For instance, consider a component C_m having four possible values and assume that at some point of time during the test case generation process, the probability of selection of each of the four values val_{m1}, val_{m2}, val_{m3} and val_{m4} becomes 0.20, 0.35, 0.35 and 0.10 respectively. If the generated random number lies in the range $[0, 0.2]$ then value val_{m1} is selected, if it lies in the range $(0.2, 0.55]$ then value val_{m2} is selected, if it lies in the range $(0.55, 0.9]$ then value val_{m3} is selected otherwise val_{m4} is selected. An example to illustrate the greedy approach to generate initial population is given below.

Example: Let us consider a component based a system having configuration $(N; 2, 2^3 3^1, CA(3, 2^3))$ as shown below:

C_1	C_2	C_3	C_4
a1	a2	a3	a4
b1	b2	b3	b4
			c4

To construct a VSCA in the initial population, VSCA-GA assigns a probability of selection to each value of every component. Initially, each value of a component has an equal number of uncovered interactions; therefore each of them will have an equal probability of selection. The probability of selection of each value of every component is shown below:

C_1	C_2	C_3	C_4
P(a1)=0.5	P(a2)=0.5	P(a3)=0.5	P(a4)=0.333
P(b1)=0.5	P(b2)=0.5	P(b3)=0.5	P(b4)=0.333
			P(c4)=0.333

The first test case TC_1 is constructed by selecting a value for each component randomly from their respective input domain. Let TC_1 be: a1, b2, a3, b4.

Now, VSCA-GA changes the probability of selection of each value of every component based on the number of their uncovered interactions using Equations 2–5. The new probabilities become:

C_1	C_2	C_3	C_4
P(a1)=0.32	P(a2)=0.68	P(a3)=0.32	P(a4)=0.166
P(b1)=0.68	P(b2)=0.32	P(b3)=0.68	P(b4)=0.417
			P(c4)=0.417

Subsequently, for generating the next test case TC_2, VSCA-GA generates random numbers. Let the random numbers generated be 0.2, 0.5, 0.2 and 0.3 for each component respectively. Therefore, TC_2 will be: a1, a2, a3, b4.

Now, VSCA-GA again changes the probability of selection of each value of every component based on the number of their uncovered interactions using Equations 2–5. The new probabilities become:

C_1	C_2	C_3	C_4
P(a1)=0.18	P(a2)=0.43	P(a3)=0.18	P(a4)=0.235
P(b1)=0.82	P(b2)=0.57	P(b3)=0.82	P(b4)=0.209
			P(c4)=0.556

The same procedure is repeated to construct the remaining $(N-2)$-test cases. Once a VSCA is generated, the same procedure is repeated to generate all the remaining VSCAs in the initial population. Notably, every time a new VSCA is generated, the interaction list L is reinitialized to store all the possible t-way and t_i-way interactions between the values of all the components based on the configuration of VSCA.

5.2. A Greedy Approach to Perform Crossover

The next step after initialization is the application of selection, crossover and mutation operators repeatedly to generate optimal VSCA that covers all possible t-way and t_i-way interactions. The crossover operator combines the genes of two or more parents to generate an offspring. It is based on the idea that the exchange of information between good chromosomes will generate even better offspring [47]. There are many variations of the crossover method, namely single-point crossover, two-point crossover, multi-point crossover, uniform crossover, etc. The number of crossover points determines how many segments are exchanged between the parents. The length (number of genes) of each segment may vary and it depends on the position of crossover points. VSCA-GA performs a crossover at the boundaries of test cases and the length of a segment is always equal to one (i.e. one test case). When a crossover is performed, it is quite possible that during the exchange of information between parents, some good features of a parent may get lost. In our case, based on the configuration of VSCA each test case $tc_{fi} | 1 \leq i \leq N$ in $VSCA_f$ covers some fixed number of t-way and t_i-way interactions, out of which some interactions are distinctly covered by tc_{fi} only. When a random crossover is performed, it may happen that during the exchange of information between two parent VSCAs say $VSCA_1$ and $VSCA_2$, the best test case tc_{1i} covering maximum number of distinct interactions in $VSCA_1$ may get exchanged with the test case tc_{2i} of $VSCA_2$. This may result in the gain of new interactions as well as loss of existing distinct interactions covered by tc_{1i} in $VSCA_1$ thereby, reducing the net gain in fitness after the crossover. The net gain in fitness is calculated using Equation 6.

$$\begin{aligned}&\text{Net gain in fitness}(VSCA_f) = \\ &\quad \text{Number of new interactions gained} - \\ &\text{Number of existing distinct interactions lost} \end{aligned} \tag{6}$$

In order to minimize the loss of existing distinct interactions and to maximize the net gain in fitness during a crossover, VSCA-GA uses a greedy approach to perform a crossover. It takes the number of test cases which are to be exchanged during the crossover as input (NTC) instead of the number of crossover points, which helps it in selecting the test cases greedily for crossover. VSCA-GA starts by selecting VSCAs using roulette wheel selection to become parents during the crossover. In the roulette wheel selection, a probability is being assigned to each individual in the population. This probability is calculated on the basis of the fitness of the individual and thus the individuals with higher fitness have better chances of getting selected for reproduction. Out of the two parents selected using roulette wheel for crossover, VSCA-GA chooses a parent with higher fitness. Let the higher fitness parent be parent_1 then VSCA-GA calculates the number of distinct t-way and t_i-way interactions covered by each test case of parent_1. Subsequently, it checks whether the number of test cases to be exchanged (NTC) is equal to the number of test cases covering least number of distinct interactions. There are three possibilities:

1. The number of test cases covering the least number of distinct interactions is equal to NTC – In this case VSCA-GA performs crossover by exchanging the test cases that cover the least number of distinct interactions in parent_1 by the respective test cases of parent_2. For instance, consider a system A having configuration $(N; 2, 3^5, CA(3, 3^4))$ (the value of each component is labelled 0, 1, 2) and let N be 7 which means that the VSCA will consist of 7 test cases represented by $\text{TC}_1, \text{TC}_2, \ldots, \text{TC}_7$. Each test case $\text{TC}_i | 1 \leq i \leq 7$ contains a value 0/1/2 corresponding to each component. Let NTC be 2. After calculating the number of distinct interactions covered by each of the 7 test cases in parent_1, it has been found that two test cases TC_2 and TC_5 cover the least number of distinct interactions (i.e. 4) in parent_1. Hence, the number of test cases covering the least number of distinct interactions in parent_1 is equal to NTC. Accordingly, a crossover is performed by exchanging TC_2 and TC_5 in parent_1 with the respective test cases TC_2 and TC_5 of parent 2 as shown in Figure 5(a).
2. NTC is greater than the number of test cases covering least number of distinct interactions. Here VSCA-GA selects first NTC test cases in parent_1 when sorted in the ascending order by the number of distinct interactions covered by them and applies a crossover at these positions. For instance, in the aforementioned system A, let NTC be 3. Here, NTC is greater than the number of test cases covering the least number of interactions, so the crossover is performed by exchanging TC_2, TC_5 and TC_4 (which covers next least number of interactions i.e. 7 after TC_2 and TC_5) with the respective test cases of parent_2 as shown in Figure 5(b).
3. The number of test cases covering the least number of distinct interactions is greater than NTC: Here VSCA-GA calculates all the t-way and t_i-way interactions covered by the respective test cases of parent_2 and performs crossover by exchanging test cases that cover the maximum number of interactions not covered by parent_1. Again, for the aforementioned system A, two test cases TC_2 and TC_5 in parent_1 cover the least number of distinct interactions (i.e., 4). Let NTC be 1, which is less than the number of test cases covering the least number of distinct interactions in parent_1. In this case, VSCA-GA calculates the number of interactions covered by the respective test cases of parent_2, in our case TC_2 and TC_5. It is clear from Figure 5(c), that TC_5 covers 5 interactions as compared to TC_2 which covers only 1 interaction, not covered in parent_1. Hence, our strategy performs a crossover by exchanging test cases TC_5 in parent_1 and parent_2. By choosing the test case in parent_1 that covers the least number of distinct interactions and exchanging it with a test case of parent_2 that covers maximum number of interactions not covered by parent_1, we ensure that the resulting offspring is of better quality than its parent by minimizing the loss of existing interactions and maximizing the gain.

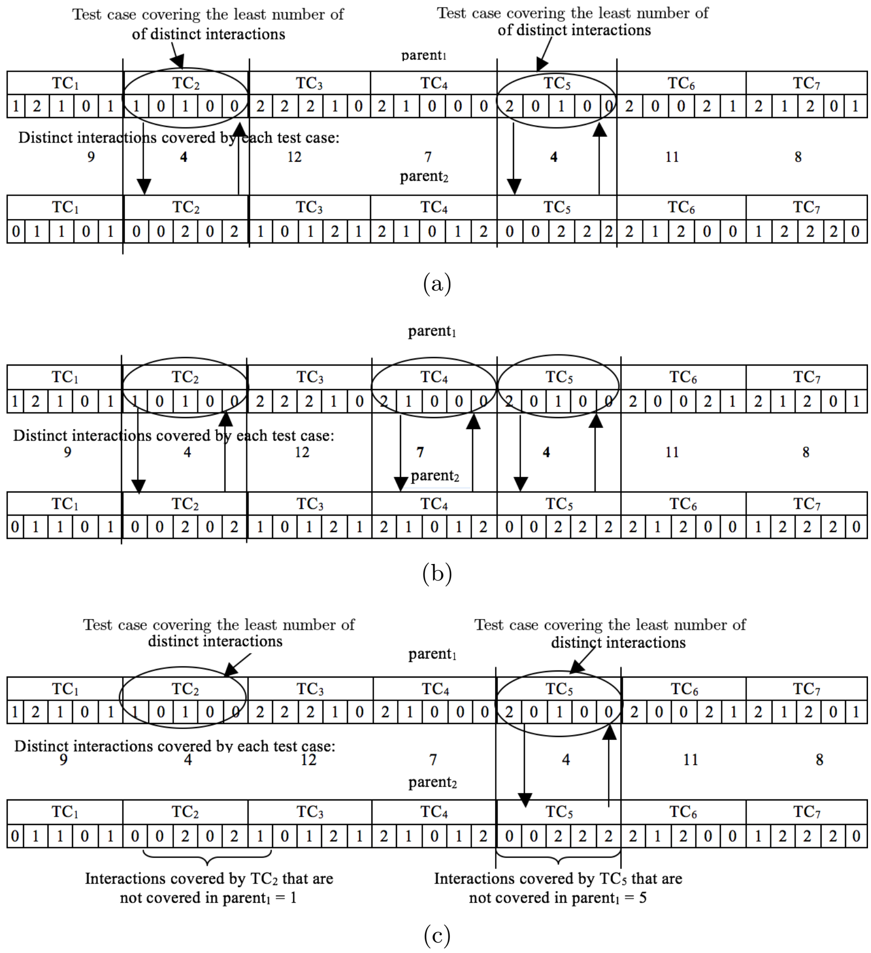
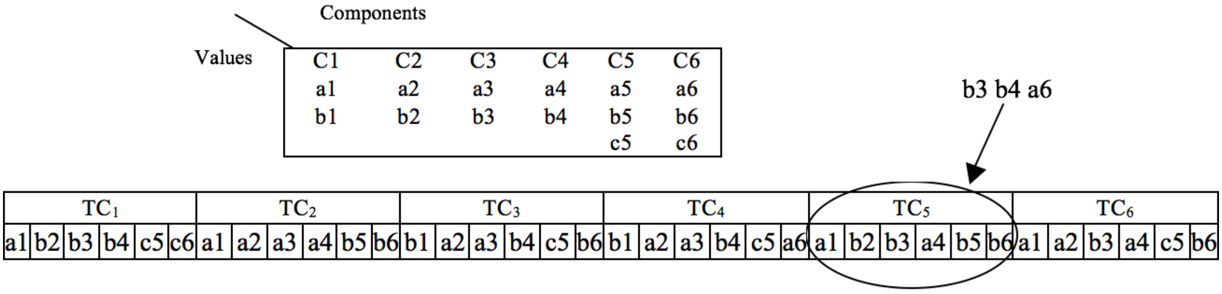


Figure 5. Multipoint crossover VSCA$(N; 2, 3^5, CA(3, 3^4))$

5.3. A Greedy Approach to Perform Mutation

Mutation has a significant effect on the performance of GA as the mutation operator randomly modifies, with a given probability, one or more genes of a chromosome, thus increasing the diversity of the population and avoids getting stuck in the local minima. In traditional GA, every individual has an equal probability of getting mutated irrespective of their fitness [48]. Thus the probability of an individual with the highest fitness to be disrupted by a mutation is equal compared to the one with the lowest fitness. Hence a mutation strategy is needed to mutate an individual to maximize improvement in fitness by minimizing fitness loss due to the mutation. Here, we present a greedy mutation strategy to perform a mutation. First, we select an individual VSCA$_f$ for a mutation and list all the t-way and t_i-way interactions left uncovered by the selected individual. Subsequently starting from the highest strength (t_h) uncovered interaction, we check interactions of strength t_h that occurs multiple times in VSCA$_f$ and replace one of its occurrences with the uncovered t_h interaction in an attempt to increase its overall fitness. However, when an existing interaction is replaced with an uncovered interaction, then in addition to the gain of new interactions some old distinct interactions may get lost. Hence, to maximize the net gain after mutation, we calculate the number of distinct interactions covered by the multiple occurring interactions in the respective test cases and replaces the one which covers the least number of distinct interactions. In case more than one test case covers the least number of distinct interactions we replace the one



Test case	Distinct interaction of Strength-2	Distinct interaction of Strength-3	Total distinct interactions
TC_5	3	2	5
TC_6	2	3	5

Figure 6. Greedy Mutation VSCA$(N; 2, 2^4 3^2, MCA(3, 2^2 3^2))$

which covers the least number of higher strength interactions. For instance, consider a system having configuration $(N; 2, 2^4 3^2, MCA(3, 2^2 3^2))$ as shown in Figure 6. It is evident from Figure 6 that the VSCA selected for mutation does not cover the triplet 'b3 b4 a6'. When examining the VSCA, it is found that the triplet 'b3 a4 b6' is covered by both TC_5 and TC_6. Hence, one occurrence of 'b3 a4 b6' can be replaced by 'b3 b4 a6'. To replace an occurrence of 'b3 a4 b6' by 'b3 b4 a6', the proposed greedy approach to mutation calculates the total number of distinct interactions of strength-2 and strength-3 covered by b3, a4 and b6 in TC_5 and TC_6. In our case both TC_5 and TC_6 cover an equal number of distinct interactions, so the proposed approach replaces 'b3 a4 b6' in TC_5 which covers a smaller number of distinct interactions of higher strength '3' by the uncovered triplet 'b3 b4 a6'.

The overall VSCA-GA strategy can be found in Appendix.

6. Experimental Results

To assess the effectiveness of VSCA-GA strategy, we implemented the proposed strategy by extending an open-source tool PWiseGen [49]. It is an open-source tool written in Java to generate pair-wise (2-way) test set using GA. It does not provide support for the construction of CA of strength $t > 2$ as well as VSCA construction. We have extended PWiseGen by adding the capability to generate VSCA of strength up to 6 using the greedy strategies proposed in Section 5 to generate the initial population, crossover and mutation. We call it PWiseGen-VSCA.

To compare the performance of PWiseGen-VSCA with the existing greedy based strategies such as IPOG, PICT, ITCH, TVG, DA-RO, DA-FO, ParaOrder, TSG and AI based strategies such as SA, ACS, VS-PSTG, HSS and HSTCG based on VSCA size, we performed experiments on a set of four benchmark problems taken from Cohen et al. [9], Ahmed et al. [35] and Alsewari and Zamli [36]. As the VSCA size is not dependent on the execution environment, we compare our result directly with the results published in literature [9, 21–23, 31, 35, 36] with respect to VSCA size.

The results of experiments conducted to compare VSCA size on four VSCA configurations with various sub-configuration settings are shown in Table 3, Table 4, Table 5 and Table 6 respectively. Cells marked NA (not available) in the table signify that the results are not available in the publications and the cells marked '–' signify that the tool/algorithm does not support the specified strength. As VSCA-GA produces non-deterministic results, we ran each configuration 30 times on PWiseGen-VSCA and reported the best VSCA size obtained over 30 runs. It can be observed from Table 3, Table 4, Table 5 and Table 6 that AI-based strategies generally perform better than their greedy counterparts.

When AI-based strategies are compared to each other, we can see that SA and ACS support construction of VSCA of strength $t \leq 3$

Table 3. VSCA Size for VSCA configuration VSCA$(N; 2, 3^{15}, C)$

{C}	No. of interactions	PICT	ITCH	DA-RO	DA-FO	Para Order	TVG	TSG	IPOG	SA	ACS	VS-PSTG	HSS	PWiseGen-VSCA	
														Best	Average
ϕ	945	35	31	21	20	33	22	20	21	16	19	19	20	16	16.33
CA$(3, 3^3)$	972	81	48	28	29	27	27	27	27	27	27	27	27	27	27
CA$(3, 3^3)^2$	999	729	59	28	29	33	30	27	28	27	27	27	27	27	27
CA$(3, 3^3)^3$	1026	785	69	28	30	33	30	28	29	27	27	27	27	27	27
CA$(3, 3^4)$	1053	105	59	32	34	27	35	33	38	27	27	30	27	27	27.9
CA$(3, 3^5)$	1215	131	62	40	42	45	41	40	41	33	38	38	38	33	34.13
CA$(3, 3^6)$	1485	146	61	46	46	49	48	48	48	34	45	45	45	40	42.13
CA$(3, 3^7)$	1890	154	68	53	53	54	54	51	51	41	48	49	51	47	48.2
CA$(3, 3^9)$	3213	177	94	60	60	62	62	59	63	50	57	57	60	57	57.33
CA$(3, 3^{15})$	13230	83	132	70	78	82	81	82	83	67	76	74	77	74	75.8
CA$(3, 3^4)$, CA$(3, 3^5)$, CA$(3, 3^6)$	1863	1376	114	46	46	44	53	48	48	34	40	45	45	40	41.5
CA$(4, 3^4)$	1026	245	103	–	–	–	81	–	81	–	–	81	81	81	81
CA$(4, 3^5)$	1350	301	118	–	–	–	103	–	100	–	–	97	94	91	91
CA$(4, 3^7)$	3780	505	189	–	–	–	168	–	165	–	–	158	159	158	158.3
CA$(5, 3^5)$	1188	730	261	–	–	–	243	–	243	–	–	243	243	243	243
CA$(5, 3^7)$	6048	1356	481	–	–	–	462	–	461	–	–	441	441	441	441
CA$(6, 3^6)$	1674	2187	745	–	–	–	729	–	729	–	–	729	729	729	729

Table 4. VSCA Size for VSCA configuration VSCA$(N; 2, 3^{20}10^2, C)$

{C}	No. of interactions	PICT	ITCH	DA-RO	DA-FO	Para Order	TVG	TSG	IPOG	SA	ACS	VS-PSTG	HSS	PWiseGen-VSCA	
														Best	Average
ϕ	3010	100	NA	100	100	100	101	100	102	100	100	102	106	100	100.33
CA$(3, 3^{20})$	33790	940	NA	100	105	103	103	100	102	100	100	105	109	100	100
MCA$(3, 3^{20}10^2)$	73990	423	NA	401	409	442	423	411	442	304	396	481	450	440	446
CA$(4, 3^3 10^1)$	3280	810	NA	–	–	–	270	–	270	–	–	270	270	270	274.53
MCA$(5, 3^3 10^2)$	5710	2430	NA	–	–	–	2700	–	2700	–	–	2700	2700	2700	2700
MCA$(6, 3^4 10^2)$	11110	7290	NA	–	–	–	8100	–	8100	–	–	8100	8100	8100	8100

Table 5. VSCA Size for VSCA configuration VSCA$(N; 2, 4^3 5^3 6^2, C)$

{C}	No. of interactions	PICT	ITCH	DA-RO	DA-FO	Para Order	TVG	TSG	IPOG	SA	ACS	VS-PSTG	HST-CG	HSS	PWiseGen-VSCA	
															Best	Average
ϕ	663	43	48	41	40	49	44	39	40	36	41	42	43	42	37	38.93
CA$(3, 4^3)$	727	384	97	64	64	64	67	64	67	64	64	64	64	64	64	64
MCA$(3, 4^3 5^2)$	1507	781	164	131	132	141	132	125	132	100	104	124	120	116	120	121.3
CA$(3, 5^3)$	788	750	145	125	125	126	125	125	126	125	125	125	125	125	125	125
MCA$(4, 4^3 5^1)$	983	1920	354	–	–	–	320	–	320	–	–	320	320	320	320	320
CA$(3, 4^3)$, CA$(3, 5^3)$	852	8000	194	125	125	129	125	125	126	125	125	125	NA	125	125	125
MCA$(4, 4^3 5^1)$, MCA$(4, 5^2 6^2)$	1883	288000	1220	–	–	–	900	–	900	–	–	900	NA	900	900	900
CA$(3, 4^3)$, MCA$(4, 5^3 6^1)$	1477	48000	819	–	–	–	750	–	750	–	–	750	NA	750	750	750
MCA$(4, 4^3 5^2)$	2503	2874	510	–	–	–	496	–	479	–	–	472	454	453	458	459.23
MCA$(3, 4^3 5^3 6^1)$	4290	1266	254	207	211	247	237	197	215	171	201	206	NA	212	204	206.76
MCA$(3, 5^1 6^2)$	843	900	188	180	180	180	180	180	180	180	180	180	180	180	180	180
MCA$(3, 4^3 5^3 6^2)$	7080	261	312	256	261	307	302	239	263	214	255	260	264	263	260	260.33
MCA$(5, 4^3 5^2)$	2263	9600	1639	–	–	–	1600	–	1600	–	–	1600	NA	1600	1600	1600
MCA$(5, 4^3 5^3)$	11463	15048	2520	–	–	–	2583	–	2487	–	–	2430	2430	2430	2434	2436.53

Table 6. VSCA Size for VSCA configuration VSCA$(N; 2, 10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1, C)$

{C}	No. of interactions	PICT	ITCH	Density	Para Order	TVG	IPOG	SA	ACS	VS-PSTG	HSS	PWiseGen-VSCA Best	PWiseGen-VSCA Average
ϕ	1266	102	119	NA	NA	99	90	NA	NA	97	94	92	93.96
MCA$(3, 10^1 9^1 8^1)$	1986	31256	765	NA	NA	720	720	NA	NA	720	720	720	720
MCA$(3, 7^1 6^1 5^1)$	1476	19515	301	NA	NA	210	211	NA	NA	210	210	210	210
MCA$(3, 4^1 3^1 2^1)$	1290	2397	140	NA	NA	99	90	NA	NA	97	94	92	92.6
MCA$(3, 10^1 9^1 8^1 7^1)$	3680	22878	806	NA	NA	784	772	NA	NA	742	740	740	745.03
MCA$(3, 10^1 9^1 8^1)$, MCA$(3, 7^1 6^1 5^1)$	2196	NA	947	NA	NA	720	720	NA	NA	720	720	720	720
MCA$(3, 10^1 9^1 8^1)$, MCA$(3, 7^1 6^1 5^1)$, MCA$(3, 4^1 3^1 2^1)$	2220	NA	968	NA	NA	720	720	NA	NA	720	720	720	720
MCA$(4, 5^1 4^1 3^1 2^1)$	1386	1200	237	–	–	123	142	–	–	120	120	120	120
MCA$(5, 10^1 9^1 4^1 3^1 2^1)$	3426	124157	2276	–	–	2160	2160	–	–	2160	2160	2160	2160
MCA$(6, 7^1 6^1 5^1 4^1 3^1 2^1)$	6306	NA	5157	–	–	5040	5043	–	–	5040	5040	5040	5040

Table 7. VSCA generation time (in seconds) for VSCA configuration VSCA$(N; 2, 3^{15}, C)$

{C}	IPOG	TVG	PWiseGen-VSCA
ϕ	0.077	0.056	2.976
CA$(3, 3^3)$	0.009	0.071	1.32
CA$(3, 3^3)^2$	0.025	0.062	13.5
CA$(3, 3^3)^3$	0.023	0.076	5.424
CA$(3, 3^4)$	0.012	0.088	60.042
CA$(3, 3^5)$	0.03	0.098	11.4
CA$(3, 3^6)$	0.013	0.141	48.06
CA$(3, 3^7)$	0.023	0.161	57.6
CA$(3, 3^9)$	0.019	0.304	97.8
CA$(3, 3^{15})$	0.048	2.008	211.08
CA$(3, 3^4)$, CA$(3, 3^5)$, CA$(3, 3^6)$	0.008	0.302	30.78
CA$(4, 3^4)$	0.025	0.108	11.4
CA$(4, 3^5)$	0.011	0.189	5431.8
CA$(4, 3^7)$	0.013	0.862	9003.6
CA$(5, 3^5)$	0.015	0.499	6.6
CA$(5, 3^7)$	0.046	3.853	12035.4
CA$(6, 3^6)$	0.093	1.388	19.44
CA$(6, 3^7)$	0.078	11.685	21183.6

Table 8. VSCA generation time (in seconds) for VSCA configuration VSCA$(N; 2, 3^{20} 10^2, C)$

{C}	IPOG	TVG	PWiseGen-VSCA
ϕ	0.012	0.636	8.9544
CA$(3, 3^{20})$	0.039	5.972	915
MCA$(3, 3^{20} 10^2)$	0.085	13.559	3813.84
CA$(4, 3^3 10^1)$	0.061	1.491	1546.8
VSCA$(5, 3^3 10^2)$	0.343	27.409	274.8
VSCA$(6, 3^4 10^2)$	1.684	208.681	378

Table 9. VSCA generation time (in seconds) for VSCA configuration VSCA$(N; 2, 4^3 5^3 6^2, C)$

{C}	IPOG	TVG	PWiseGen-VSCA
ϕ	0.002	0.035	2.37
CA$(3, 4^3)$	0.002	0.041	2.106
MCA$(3, 4^3 5^2)$	0.002	0.156	433.8
CA$(3, 5^3)$	0.005	0.077	0.39
MCA$(4, 4^3 5^1)$	0.016	0.189	18.4704
CA$(3,4^3)$, CA$(3,5^3)$	0.001	0.082	0.5772
MCA$(4,4^3 5^1)$, MCA $(4,5^2 6^2)$	0.047	1.136	52.6344
CA$(3,4^3)$, MCA$(4, 5^3 6^1)$	0.032	0.699	35.9112
MCA$(4,4^3 5^2)$	0.023	0.917	6992.4
MCA$(3, 4^3 5^3 6^1)$	0.015	0.733	1173.6
MCA$(3,5^1 6^2)$	0.003	0.089	0.45
MCA$(3,4^3 5^3 6^2)$	0.011	1.621	1579.2
MCA $(5, 4^3 5^2)$	0.11	2.84	30.6
MCA $(5, 4^3 5^3)$	0.296	26.193	7485.6

Table 10. VSCA generation time (in seconds) for VSCA configuration VSCA$(N; 2, 10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1, C)$

{C}	IPOG	TVG	PWiseGen-VSCA
ϕ	0.003	0.414	7.8
MCA$(3, 10^1 9^1 8^1)$	0.003	0.865	5.148
MCA$(3, 7^1 6^1 5^1)$	0.007	0.241	6.72
MCA$(3, 4^1 3^1 2^1)$	0.002	0.131	37.518
MCA$(3, 10^1 9^1 8^1 7^1)$	0.044	2.169	2586.24
MCA$(3, 10^1 9^1 8^1)$, MCA$(3, 7^1 6^1 5^1)$	0.031	0.893	6.0684
MCA$(3, 10^1 9^1 8^1)$, MCA$(3, 7^1 6^1 5^1)$, MCA$(3, 4^1 3^1 2^1)$	0.028	0.894	7.7376
MCA$(4, 5^1 4^1 3^1 2^1)$	0.003	0.021	635.22
MCA $(5, 10^1 9^1 4^1 3^1 2^1)$	0.234	7.504	85.8
MCA $(6, 7^1 6^1 5^1 4^1 3^1 2^1)$	0.733	38.548	484.02

only whereas VS-PSTG supports construction of VSCA of strength up to 6. The published results [36] show that unlike other greedy and AI-based strategies , HSS support construction of VSCA of strength up to 15 but nothing is mentioned about the efficiency of HSS in terms of VSCA generation time. From Table 3, we can infer that PWiseGen-VSCA outperforms ACS whereas the results in Table 4 and Table 5 are comparable. Although PWiseGen-VSCA supports construction of higher interaction strength VSCA however, VSCA generation time increases with the increase in interaction strength which makes it infeasible to generate higher strength VSCAs. It is evident from Tables 3-6 that PWiseGen-VSCA generates better results as compared to VS-PSTG, HSTCG and HSS. From Tables 3–6, it is clear that SA outperforms existing state-of-the-art strategies for lower interaction strength ($t \leq 3$), however, the results generated by PWiseGen-VSCA are equal or close to SA.

Finally from Tables 3–6, we can conclude that PWiseGen-VSCA generates optimal VSCA most of the time as compared to other greedy and meta-heuristic techniques for strength ≤ 6.

It is difficult to compare PWiseGen-VSCA with the existing state-of-the-art algorithms in terms of VSCA generation time, as the generation time is dependent on the running environment and most of the algorithm implementations are not publicly available. To perform a fair comparison, we restrict the comparison of VSCA generation time against publicly available algorithm implementation: ACTS (IPOG) and TVG. These tools are run on Windows using an INTEL Pentium Dual Core 1.73 GHZ processor with 1.00 GB of memory The results of comparison made on the dataset of Tables 3–6 with respect to VSCA generation time (in seconds) are shown in Tables 7–10 respectively. It is evident from Tables 7–10 that PWiseGen-VSCA requires more time to construct VSCA as compared to ACTS (IPOG) and TVG, however, the extra time consumed by PWiseGen-VSCA allowed the construction of VSCAs of smaller size.

7. Threats to Validity

One important threat to validity of the effectiveness of our approach is that we could not use any sophisticated statistical hypothesis tests such as Welch's t-test to assess and compare PWiseGen-VSCA with the existing meta-heuristic techniques for constructing VSCA as we do not have access to the source code of any of them. Also, we could not compare the efficiency of PWiseGen-VSCA in terms of VSCA generation time with the existing meta-heuristic techniques because of the above mentioned reason.

8. Conclusion and Future Work

In this paper we have presented and evaluated VSCA-GA, a strategy based on GA to construct optimal VSCA for t-way testing. The strategy is implemented in PWiseGen-VSCA. Our strategy exploits the strength of both greedy and meta-heuristic techniques by integrating greedy technique with GA. The experiments conducted on a set of benchmark problems show that PWiseGen-VSCA outperforms the existing state-of-the-art algorithms except SA in terms of VSCA sizes. However, our results are comparable to SA which generates VSCA for strength t up to 3 whereas VSCA-GA constructs VSCA for strength t up to 6.

In future, we plan to construct VSCA to handle feature constraints and try to improve the efficiency of PWiseGen-VSCA to construct higher strength VSCA.

References

[1] 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.

[2] A. Hartman, "Software and hardware testing using combinatorial covering suites," in *Graph*

Theory, Combinatorics and Algorithms. Springer, 2005, pp. 237–266.

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

[4] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The combinatorial design approach to automatic test generation," *IEEE software*, No. 5, 1996, pp. 83–88.

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

[6] 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.

[7] 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.

[8] D.R. Kuhn and M.J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *27th Annual NASA Goddard/IEEE Software Engineering Workshop*. IEEE, 2002, pp. 91–95.

[9] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, and J.S. Collofello, "A variable strength interaction testing of components," in *27th Annual International Computer Software and Applications Conference*. IEEE, 2003, pp. 413–418.

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

[11] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, Vol. 43, No. 2, 2011, p. 11.

[12] A. Hedayat, N. Sloane, and J. Stufken, *Orthogonal Arrays*, ser. Springer Series in Statistics. Springer, New York, 1999.

[13] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Communications of the ACM*, Vol. 28, No. 10, 1985, pp. 1054–1058.

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

[15] J.H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence*. MIT press, 1992.

[16] K.F. Man, K.S. Tang, and S. Kwong, "Genetic algorithms: concepts and applications," *IEEE Transactions on Industrial Electronics*, Vol. 43, No. 5, 1996, pp. 519–534.

[17] S.K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2014, pp. 323–334.

[18] C.J. Colbourn, M.B. Cohen, and R. Turban, "A deterministic density algorithm for pairwise interaction coverage." in *IASTED Conf. on Software Engineering*. Citeseer, 2004, pp. 345–352.

[19] J. Arshem. TVG download web page. (2009). [Online]. http://sourceforge.net/projects/tvg

[20] J. Czerwonka, "Pairwise testing in real world: Practical extensions to test case generator," in *PNSQC '06: Proceedings of 24th Pacific Northwest Software Quality Conference*, 2006, pp. 419–430.

[21] 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*. IEEE, 2008, pp. 155–160.

[22] 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.

[23] 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.

[24] M. Forbes, J. Lawrence, Y. Lei, R.N. Kacker, and D.R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, Vol. 113, No. 5, 2008, pp. 287–297.

[25] L. Yu, Y. Lei, R.N. Kacker, and D.R. Kuhn, "ACTS: A combinatorial test generation tool," in *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013, pp. 370–375.

[26] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege, "A systematic review of

the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering,*, Vol. 36, No. 6, 2010, pp. 742–762.

[27] 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.

[28] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Computing Surveys (CSUR)*, Vol. 35, No. 3, 2003, pp. 268–308.

[29] B. Jenkins. Jenny download web page. (2005). [Online]. http://burtleburtle.net/bob/math/jenny.html

[30] 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.

[31] X. Chen, Q. Gu, A. Li, and D. Chen, "Variable strength interaction testing with an ant colony system approach," in *APSEC '09. Asia-Pacific Software Engineering Conference*. IEEE, 2009, pp. 160–167.

[32] 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.

[33] J.D. McCaffrey, "An empirical study of pairwise test set generation using a genetic algorithm," in *Seventh International Conference on Information Technology: New Generations (ITNG)*. IEEE, 2010, pp. 992–997.

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

[35] B.S. Ahmed and K.Z. Zamli, "A variable strength interaction test suites generation strategy using particle swarm optimization," *Journal of Systems and Software*, Vol. 84, No. 12, 2011, pp. 2171–2185.

[36] A.R.A. Alsewari and K.Z. Zamli, "Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support," *Information and Software Technology*, Vol. 54, No. 6, 2012, pp. 553–568.

[37] J. LI, D. XING, and Y. ZHAO, "Combinatorial test suite generation of variable strength based on harmony search," *Journal of Network & Information Security*, Vol. 4, No. 2, 2013, pp. 177–188.

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

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

[40] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st ed. Reading, Mass: Addison-Wesley Professional, Jan. 1989.

[41] H. Maaranen, K. Miettinen, and M.M. Mäkelä, "Quasi-random initial population for genetic algorithms," *Computers & Mathematics with Applications*, Vol. 47, No. 12, 2004, pp. 1885–1895.

[42] Y.W. Leung and Y. Wang, "An orthogonal genetic algorithm with quantization for global numerical optimization," *IEEE Transactions on Evolutionary Computation*, Vol. 5, No. 1, 2001, pp. 41–53.

[43] H. Maaranen, K. Miettinen, and A. Penttinen, "On initial populations of a genetic algorithm for continuous optimization problems," *Journal of Global Optimization*, Vol. 37, No. 3, 2007, pp. 405–436.

[44] S. Rahnamayan, H.R. Tizhoosh, and M.M. Salama, "A novel population initialization method for accelerating evolutionary algorithms," *Computers & Mathematics with Applications*, Vol. 53, No. 10, 2007, pp. 1605–1614.

[45] H. Wang, Z. Wu, J. Wang, X. Dong, S. Yu, and C. Chen, "A new population initialization method based on space transformation search," in *Fifth International Conference on Natural Computation*, Vol. 5. IEEE, 2009, pp. 332–336.

[46] 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.

[47] D. Ortiz-Boyer, C. Hervás-Martínez, and N. García-Pedrajas, "CIXL2: A crossover operator for evolutionary algorithms based on population features." *J. Artif. Intell. Res.(JAIR)*, Vol. 24, 2005, pp. 1–48.

[48] S.M. Libelli and P. Alba, "Adaptive mutation in genetic algorithms," *Soft Computing*, Vol. 4, No. 2, 2000, pp. 76–80.

[49] P. Flores. PWiseGen download web page. (2010). [Online]. https://code.google.com/p/pwisegen/

Appendix

Input: VSCA configuration: $(t, k, (v_1, v_2, \ldots, v_k)$, C), VSCA size: N, Population Size: P_{size}, Maximum Number of Generations: NOG, Number of Reproductions: NOR, Number of Test Cases for Crossover: NTC
Output: Optimal VSCA

Algorithm 1. VSCA-GA

procedure VSCA-GA
  Initialize $G := 0$ ▷ generation number
  ▷ Generate initial population pop_1
  for each VSCA$_f$ in pop_1 **do** ▷ $1 \leq f \leq P_{size}$
    Create an interaction list L of all t-way and t_i-way interactions between all components C_m ▷ $1 \leq m \leq k$
    for $m = 1$ to k **do**
      for $j = 1$ to v_m **do**
        Store the number of uncovered interactions of value val$_{mj}$ of component C_m in $N_{\text{uncovered}}(\text{val}_{mj})$
      end for
    end for
    for each component C_m **do**
      Assign each value val$_{mj}$ an equal probability of selection P (val$_{mj}$)
    end for
    ▷ Generate first test case
    Create the first test case tc_{f1} by selecting a value for each component C_m randomly
    Let val$_{ms}$ is the selected value of component C_m
    ▷ Generate remaining $(N-1)$ test cases
    Initialize $i := 2$
    for $i = 2$ to N **do**
      for $m = 1$ to k **do**
        Update interaction list L by eliminating the interactions covered by val$_{ms}$ in test case $tc_{f(i-1)}$
        Store the number of remaining interactions of val$_{ms}$ in $N'_{\text{uncovered}}(\text{val}_{ms})$
        Decrease probability of selection of value val$_{ms}$ of C_m selected in test case $tc_{f(i-1)}$ according to equation 3
        Update probability of remaining values of C_m according to equation 5.
      end for
      for each component C_m **do**
        Generate a random number between 1 to 100
        Create test case tc_{fi} by selecting a value of C_m based on the interval in which the random number falls
      end for
    end for
  end for
  Calculate fitness of each VSCA$_f$ in pop_1 using equation 1
  $G \leftarrow G + 1$
  while solution not found **and** $G \leq NOG$ **do**
    ▷ Perform Crossover
    Initialize *counter* := 1
    while *counter* $\leq NOR$ **do**
      Select two parent VSCA from population pop_{G-1} using Roulette Wheel Selection
      Let $parent_1$ is the parent VSCA having higher fitness among the two parents
      Calculate the number of distinct pairs covered by each test case of $parent_1$
      if $NTC <$ number of test cases covering least number of distinct interactions **then**
        Calculate the number of interactions covered by respective test cases of $parent_2$
        Select NTC test cases of $parent_2$ that cover maximum number of interactions not covered by $parent_1$
      else if $NTC >$ number of test cases covering least number of distinct interactions **then**
        Select NTC test cases in $parent_1$ when sorted in ascending order by the number of distinct interactions covered by them

        else
          Select NTC test cases that covers least number of distinct interactions in $parent_1$
        end if
        Perform crossover between $parent_1$ and $parent_2$ by exchanging selected test cases to generate offsprings os_1, os_2
        ▷ Perform Mutation
        Apply greedy mutation on os_1 and os_2 as discussed in Section 5.3
        Replace weaker VSCA in pop_{G-1} by os_1, os_2 to form new population pop_G
        $counter \leftarrow counter + 1$
      end while
      Calculate fitness of each $VSCA_f$ in the pop_G using equation 1
      if solution found **then**
        break
      else
        $G \leftarrow G + 1$
      end if
    end while
    if generations $> NOG$ **then**
      return (solution not found)
    else
      return $VSCA_f$ ▷ VSCA with 100% fitness
    end if
end procedure