International Journal of Software Engineering and Knowledge Engineering

PRIORITIZATION OF COMBINATORIAL TEST CASES BY INCREMENTAL INTERACTION COVERAGE

RUBING HUANG[1,*], XIAODONG XIE[2], DAVE TOWEY[3], TSONG YUEH CHEN[4], YANSHENG LU[1] and JINFU CHEN[5]

[1]*School of Computer Science and Technology, Huazhong University of Science and Technology Wuhan, Hubei 430074, P.R. China*

[2]*School of Computer Science and Technology, Huaqiao University Xiamen, Fujian 362021, P.R. China*

[3]*BNU-HKBU: United International College, Zhuhai, Guangdong 519085, P.R. China*

[4]*Faculty of Information and Communication Technologies, Swinburne University of Technology Hawthorn, Victoria 3122, Australia*

[5]*School of Computer Science and Telecommunication Engineering, Jiangsu University Zhenjiang, Jiangsu 212013, P.R. China*

Combinatorial testing is a well-recognized testing method, and has been widely applied in practice. To facilitate analysis, a common approach is to assume that all test cases in a combinatorial test suite have the same fault detection capability. However, when testing resources are limited, the order of executing the test cases is critical. To improve testing cost-effectiveness, prioritization of combinatorial test cases is employed. The most popular approach is based on interaction coverage, which prioritizes combinatorial test cases by repeatedly choosing an unexecuted test case that covers the largest number of uncovered parameter value combinations of a given strength (level of interaction among parameters). However, this approach suffers from some drawbacks. Based on previous observations that the majority of faults in practical systems can usually be triggered with parameter interactions of small strengths, we propose a new strategy of prioritizing combinatorial test cases by incrementally adjusting the strength values. Experimental results show that our method performs better than the random prioritization technique and the technique of prioritizing combinatorial test suites according to test case generation order, and has better performance than the interaction-coverage-based test prioritization technique in most cases.

Keywords: Software testing; combinatorial testing; test case prioritization; interaction coverage; incremental interaction coverage; algorithm.

1. Introduction

Suppose that a system under test (SUT) is affected by its k parameters (or factors), and each of these parameters may have many possible values (or levels). Ideally, to ensure system quality, we should test all combinations of parameter values. However, it is practically infeasible to do this due to the large amount of resources and effort

*Corresponding Author: School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. Email: rbhuang@hust.edu.cn.

required, especially for complex systems with a large number of parameters and values.

Combinatorial testing (or combinatorial interaction testing), a black-box testing technique, aims at generating an effective test suite in order to detect failures triggered by the interactions among parameters of the SUT. It is widely applied in various applications, especially for highly-configurable systems [1–5]. Combinatorial testing provides a tradeoff between testing effectiveness and efficiency, as it uses a smaller test suite that covers certain key combinations of parameter values for sampling the entire combination space. For example, 2-wise combinatorial testing (or pairwise testing where the level of interaction among parameters, the *strength*, is 2) only requires the generated test suite to cover all possible 2-tuples of parameter values (referred to as 2-wise parameter value combinations).

In the fault model of combinatorial testing, it is assumed that failures are caused by parameter interactions. Previous studies have shown that faults can normally be identified by testing interactions among a small number of parameters [1, 6, 7]. A failure-causing interaction is called a *faulty interaction*, and the size of a faulty interaction (that is, the number of parameters required to detect a failure) is referred to as the *failure-triggering fault interaction* (or FTFI) number [1, 6].

Traditionally, combinatorial testing treats all test cases equally in a test suite. However, the order of executing the test cases may be critical in practice, for example in regression testing with limited test resources. Therefore, the potentially failure-revealing test cases should be executed as early as possible. In other words, a well-designed test case execution order may be able to identify failures earlier, and thus enable earlier fault characterization, diagnosis and revision [7]. To improve testing efficiency, test case prioritization [8], which means to prioritize test cases according to some strategy, has been introduced. In test case prioritization, a prioritized test suite is generally referred to as a test sequence.

Test case prioritization of combinatorial test suites has also been well studied [4, 9–15]. Many techniques have been proposed to guide the prioritization of combinatorial test cases, such as random prioritization [9] and branch-coverage-based prioritization [13]. The most well-studied approach of prioritizing combinatorial test suite is based on interaction coverage (called interaction-coverage-based prioritization), which prioritizes test cases by repeatedly selecting an unexecuted element such that it covers the largest number of uncovered parameter value combinations of a given strength [4, 9–15].

However, the interaction-coverage-based prioritization technique has two challenges. Firstly, given a combinatorial test suite T of strength t, the prioritization method by interaction coverage only takes account of parameter value combinations of strength t for ordering T, which means that a test sequence prioritized by interaction coverage may only favor parameter value combinations of strength t. In other words, this test sequence may not be effective for τ-wise ($1 \leq \tau < t$) combinations of parametric values. A second challenge is that testers need to specify the strength.

Kuhn and his colleagues [1, 6] investigated interaction failures by analyzing the

fault reports of several software projects. They concluded that over 50% of faults can be triggered by one-wise interactions; more than 70% of faults can be detected by testing two-wise interactions; and approximately 90% of the faults can be discovered with three-wise interactions. In other words, the majority of faults in the SUT are generally caused by interactions of small strengths. Therefore, it is reasonable and practical to prioritize combinatorial test cases by covering all parameter value combinations at small strengths as early as possible. /*** Dave's comment [1]: perhaps delete the next sentence ***/ We would like to emphasize this category of failures in this paper.

Motivated by these facts, we propose a novel technique of prioritizing combinatorial test cases based on incremental interaction coverage, which orders combinatorial test cases by reusing already selected test cases and incrementally adjusting the strength values. Given a combinatorial test suite T of strength t, our strategy aims to prioritize T into a test sequence such that all possible parameter value combinations of each strength lower than t would be covered as early as possible. Therefore, our method has at least two advantages over the interaction-coverage-based prioritization technique: (1) no selection of strength is required in advance; and (2) different strengths are considered. Compared with the interaction-coverage-based prioritization technique, our method provides a priority of strengths lower than t over the strength t. In other words, our prioritized test suites cover all t-wise combinations of parameter values with lower priorities – not just all parameter value combinations at strengths lower than t. In terms of covering parameter value combinations and fault detection, experimental results show that our method has better performance than the random prioritization approach and the method of prioritizing combinatorial test suite according to the test case generation order, and also performs better than the interaction-coverage-based prioritization technique in most cases.

This paper is organized as follows. Section 2 introduces some preliminaries about combinatorial testing, and test case prioritization. Section 3 introduces a new prioritization strategy based on incremental interaction coverage, and analyzes its time complexity. Section 4 presents results of the simulations and empirical studies. Section 5 summarizes some related work, and Section 6 describes the conclusions and potential future work.

2. Preliminaries

In this section, some preliminaries of combinatorial testing and test case prioritization are presented.

2.1. *Combinatorial testing*

Combinatorial testing is widely used in the combinatorial test space to generate an effective test suite for detecting interaction faults that are triggered by interactions among parameters in the SUT.

Suppose that the SUT has k parameters $P_1, P_2, \cdots, P_k$, which may represent user inputs or configuration parameters, and each parameter P_i has discrete valid values from the finite set V_i. Let C be the set of constraints on parameter value combinations, and R be the set of interaction relations among parameters. In the remainder of this paper, we will refer to a *combination of parameters* as a *parameter combination*, and a *combination of parameter values* or a *parameter value combination* as a *value combination*.

Definition 1. A test profile, denoted as $TP(k, |V_1||V_2|\cdots|V_k|, C)$, is about the information on a combinatorial test space of the SUT, including k parameters, $|V_i|(i = 1, 2, \cdots, k)$ values for the i-th parameter, and constraints C on value combinations.

For example, Table 1 gives the configurations of a component-based system, in which there are four configuration parameters, each of which has three values. Therefore, its test profile can be written as $TP(4, 3^4, \emptyset)$.

Definition 2. Given a test profile denoted by $TP(k, |V_1||V_2|\cdots|V_k|, C)$, a k-tuple $(v_1, v_2, \cdots, v_k)$ is a test case for SUT, where $v_i \in V_i(i = 1, 2, \cdots, k)$.

For example, a 4-tuple $tc = (\text{Windows}, \text{IE}, \text{LAN}, \text{Access})$ is a test case for the SUT shown in Table 1.

Definition 3. Given a $TP(k, |V_1||V_2|\cdots|V_k|, C)$, an $N \times k$ matrix is a t-wise $(1 \leq t \leq k)$ covering array denoted as $CA(N; t, k, |V_1||V_2|\cdots|V_k|)$, which satisfies the following properties: (1) each column $i(i = 1, 2, \cdots, k)$ contains only elements from the set V_i; and (2) the rows of each $N \times t$ sub-matrix cover all t-tuples of parametric values from the t columns at least once.

When $|V_1| = |V_2| = \cdots = |V_k| = v$, the covering array can also be written as

Table 1. Configurations of a component-based system.

Operating system	Browser	Network connection	Database
Windows	IE	LAN	DB/2
Linux	Firefox	VPN	Access
Solaris	Netscape	ISND	Oracle

Table 2. A combinatorial test suite for pairwise testing.

Test No.	Operating system	Browser	Network connection	Database
1	Windows	IE	LAN	DB/2
2	Windows	Firefox	VPN	Oracle
3	Windows	Netscape	ISND	Access
4	Linux	IE	ISND	Oracle
5	Linux	Firefox	LAN	Access
6	Linux	Netscape	VPN	DB/2
7	Solaris	IE	VPN	Access
8	Solaris	Firefox	ISND	DB/2
9	Solaris	Netscape	LAN	Oracle

$CA(N;t,k,v)$. Obviously, the interaction relation set R has elements of the same size for $CA(N;t,k,|V_1||V_2|\cdots|V_k|)$, that is, $R=\{\{P_{j_1},P_{j_2},\cdots,P_{j_t}\}|1\leq j_1<j_2<\cdots<j_t\leq k,\ t \text{ is fixed}\}$ and $|R|=C_k^t$.

On the other hand, a covering array T denoted as $CA(N;t,k,|V_1||V_2|\cdots|V_k|)$ is also a covering array of strength $\tau(1\leq\tau<t)$. In other words, T can also be written as $CA(N;\tau,k,|V_1||V_2|\cdots|V_k|)$ where $1\leq\tau<t$. Thus, there exists a subset $T'\subseteq T$ such that T' is a covering array of strength τ, that is, $CA(|T'|,\tau,k,|V_1||V_2|\cdots|V_k|)$.

For example, to achieve exhaustive testing of all possible value combinations for the system shown in Table 1, we should require $3^4=81$ test cases. However, as shown in Table 2, 2-wise combinatorial testing requires only a set of 9 test cases (denoted as $CA(9;2,4,3^4)$ or $CA(9;2,4,3)$) for covering all pairs of parameter values.

Definition 4. Given a $TP(k,|V_1||V_2|\cdots|V_k|,C)$, a variable-strength covering array, denoted as $VCA(N;t,k,|V_1||V_2|\cdots|V_k|,Q)$, is an $N\times k$ covering array of strength t containing Q, which is a set of CAs, every element of which is of strength $>t$ and is defined on a subset of k parameters.

Intuitively speaking, a $VCA(N;t,k,|V_1||V_2|\cdots|V_k|,R)$ can also be considered a $CA(N;t,k,|V_1||V_2|\cdots|V_k|)$, with the interaction relation set R of the VCA containing elements of different sizes, that is, the VCA contains other CAs.

Each row of a covering array or variable-strength covering array stands for a test case while each column represents a parameter of the SUT. Testing with a t-wise covering array is called t-wise combinatorial testing, while testing with a variable-strength covering array is called variable-strength combinatorial testing.

In combinatorial testing, the *uncovered t-wise value combinations distance* (UVCD) is a distance measure often used to evaluate test cases when constructing a covering array or variable-strength covering array [16].

Definition 5. Given a combinatorial test suite T, strength t, and a test case tc, uncovered t-wise value combinations distance (UVCD) of tc is defined as:

$$UVCD_t(tc,T)=|CombSet_t(tc)\setminus CombSet_t(T)|, \tag{1}$$

where $CombSet_t(tc)$ is defined as the set of t-wise value combinations covered by test case tc, while $CombSet_t(T)$ is the set of t-wise value combinations covered by test suite T. More specifically, let $tc=(v_1,v_2,\cdots,v_k)$ where $v_i\in V_i(i=1,2,\cdots,k)$, $CombSet_t(tc)$ and $CombSet_t(T)$ can be respectively written as follows:

$$CombSet_t(tc)=\{(v_{j_1},v_{j_2},\cdots,v_{j_t})|v_{j_1}\in V_{j_1},v_{j_2}\in V_{j_2},\cdots,v_{j_t}\in V_{j_t},\ 1\leq j_1<j_2<\cdots<j_t\leq k\}, \tag{2}$$

$$CombSet_t(T)=\bigcup_{tc\in T}CombSet_t(tc). \tag{3}$$

To reduce the cost of combinatorial testing, many researchers have focused on algorithms to generate the optimal combinatorial test suite with the minimal number of test cases. Unfortunately, it has been proven that the problem of constructing covering arrays or variable-strength covering arrays is NP-Complete [17]. Nevertheless, many strategies and tools for building combinatorial test suites have been developed in recent years. Some major approaches to combinatorial test suite construction involve greedy algorithms, heuristic search algorithms, recursive algorithms, and algebraic methods (see [7] for more details).

2.2. *Test case prioritization*

To illustrate our work clearly, let us initially define a few terms. /*** Dave's comment [2]: Earlier, we used tc to refer to test cases, should we continue that here? ***/ Suppose $T = \{t_1, t_2, \cdots, t_N\}$ is a test suite of size N, and $S = \langle s_1, s_2, \cdots, s_N \rangle$ is an ordered set suite (we call it a test sequence) where $s_i \in T$ and $s_i \neq s_j (i, j = 1, 2, \cdots, N; i \neq j)$. If two test sequences are $S_1 = \langle s_1, s_2, \cdots, s_m \rangle$ and $S_2 = \langle q_1, q_2, \cdots, q_n \rangle$, we define $S_1 \overline{\wedge} S_2$ as $\langle s_1, s_2, \cdots, s_m, q_1, q_2, \cdots, q_n \rangle$. By definition, $T \setminus S$ is the maximal subset of T whose elements are not in S.

Test case prioritization is done to obtain a schedule of test cases, so that, according to some criteria (such as the cost of test case execution or statement coverage), test cases with higher priority are executed earlier in testing. A well-prioritized test sequence may improve the likelihood of detecting faults early. The problem of test case prioritization is defined as follows, from [8].

Definition 5. Given (T, Ω, f), where T is a test suite, Ω is the set of all possible test sequences obtained by ordering test cases of T, and f is a function from Ω to the set of real numbers, the problem of test case prioritization is to find an $S \in \Omega$ such that:

$$(\forall S')\ (S' \in \Omega)\ (S' \neq S)\ [f(S) \geq f(S')]. \tag{4}$$

In Equation 4, f is a function which evaluates a test sequence S by returning an award value. /*** Dave's comment [3]: please confirm the following sentence is correct ***/ The most well-known function is *a weighted average of the percentage of faults detected* (APFD) [18], which is a measure of how quickly a test sequence can detect faults during the execution. Let T be a test suite of size n, and let F be a set of m faults revealed by T. Let SF_i be the first test case in test sequence S of T which detects fault i. The APFD for test sequence S is given by the following equation from [18]:

$$APFD = 1 - \frac{SF_1 + SF_2 + \cdots + SF_m}{n \times m} + \frac{1}{2n}. \tag{5}$$

To date, many techniques of test case prioritization have been proposed according to different criteria, such as time-aware prioritization [19], search-based

prioritization [20], risk-exposure-based prioritization [21], source-code-based prioritization [8,22], fault-severity-based prioritization [23], and history-based prioritization [24]. Most test case prioritization strategies can be categorized into two classes: greedy methods and meta-heuristic search methods [15].

3. Incremental-Interaction-Coverage-Based Test Prioritization

In this section, we present a method of prioritizing combinatorial test cases based on incremental interaction coverage (denoted IICBP), a heuristic algorithm implementing this method, and a complexity analysis of the algorithm.

3.1. *Method*

The IICBP technique divides a $CA(N; t, k|V_1||V_2|\cdots|V_k|)$ into t independent subsets $(A_1, A_2, \cdots, A_t)$ such that:

$$A_i \bigcap A_j = \emptyset, i, j = 1, 2, \cdots, t, i \neq j; \tag{6}$$

$$\bigcup_{i=1}^{j} A_i = CA(\sum_{i=1}^{j} |A_i|; j, k, |V_1||V_2|\cdots|V_k|), j = 1, 2, \cdots, t, \tag{7}$$

where $A_i(i = 1, 2, \cdots, t)$ is a test sequence prioritized by interaction-coverage-based strategy [4, 11–13, 15] of strength i. Each subset A_j $(j = 2, 3, \cdots, t)$ is prioritized by ICBP using the seeding set $\bigcup_{l=1}^{j-1} A_l$. However, the processes of covering array partition and prioritization for each sub-partition are inter-related in such a way that once a subpartition is completed, test case prioritization of this sub-partition is also completed. In other words, each test case in $A_i(i = 1, 2, \cdots, t)$ is selected by using strength i and previously chosen test cases as seeds. Once all i-wise value combinations are covered by the selected test cases (that is, A_i has been successfully constructed), strength i is incremented by 1. The criterion is to choose the element e' from test suite T as the next test element in test sequence S such that:

$$(\forall e)\ (e \in T)\ (e \neq e')\ [UVCD_i(e', S) \geq UVCD_i(e, S)]. \tag{8}$$

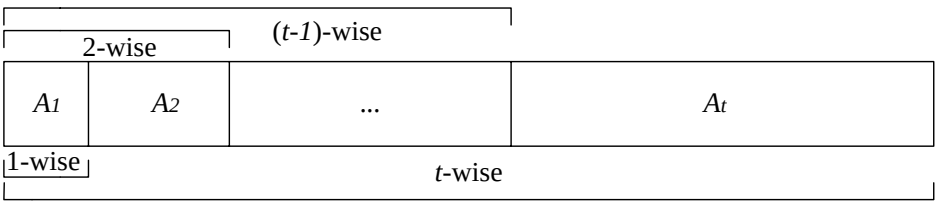


Fig. 1. Illustration of prioritizing combinatorial test cases by incremental-interaction-coverage.

Algorithm 1 Select the best test element (BTES)

Input: Already prioritized test set S, candidate test suite T, and strength t
Output: Best test element $best\_data \in T$

```
1: Set best_distance = −1;
2: for (each element e ∈ T)
3:    Calculate UVCD of e, that is, UVCD_t(e, S);
4:    if (UVCD_t(e, S) ≥ best_distance)
5:       best_distance = UVCD_t(e, S);
6:       best_data = e;
7:    end if
8: end for
9: return best_data.
```

Algorithm 2 Interaction-coverage-based prioritization of combinatorial test cases (ICBP)

Input: Covering array $CA(N; t, k, |V_1||V_2| \cdots |V_k|)$, denoted as T
Output: Test sequence S

```
1: Initialize S = ⟨⟩;
2: while (|S| ≠ N)
3:    best_data = BTES(S, T, t); //Generate the best test element.
4:    T = T \ {best_data};
5:    S = S^⟨best_data⟩;
6: end while
7: return S.
```

Algorithm 3 Incremental-interaction-coverage-based prioritization of combinatorial test cases (IICBP)

Input: Covering array $CA(N; t, k, |V_1||V_2| \cdots |V_k|)$, denoted as T
Output: Test sequence S

```
1: Initialize S = ⟨⟩, τ = 1, T' = T;
2: while (|S| ≠ N)
3:    if (|CombSet_τ(S)| == |CombSet_τ(T)|)
4:       τ = τ + 1;
5:    end if
6:    best_data = BTES(S, T', τ); //Generate the best test element.
7:    T' = T' \ {best_data};
8:    S = S^⟨best_data⟩;
9: end while
10: return S.
```

The process is repeated until all $A_i(i = 1, 2, \cdots, t)$ are prioritized according to i-wise interaction coverage. Fig. 1 gives a schematic diagram for the relationship between A_i and the relevant i-wise interaction coverage.

Since the element selection criterion (see Equation 8) is widely used in the prioritization of combinatorial test cases, we present the algorithm implementing this criterion (Algorithm 1). However, there may exist more than one best test element, indicating that they have the same maximal UVCD value. In such a tie case, we randomly select one best element. The test case prioritization technique by interaction coverage (denoted as ICBP) [4, 11–13, 15] is also given in Algorithm 2, and Algorithm 3 presents the detailed IICBP processes.

In this paper, we assume that a combinatorial test suite is equivalent to a covering array, and that all parameters are independent. In other words, the variable-strength covering array is not considered in this paper. Also, constraints on value combinations are ignored. Therefore, the test profile can be abbreviated as $TP(k, |V_1||V_2| \cdots |V_k|)$.

3.2. *Complexity analysis*

In this subsection, we briefly analyze the time complexity for the IICBP algorithm (Algorithm 3). Given a $CA(N; t, k, |V_1||V_2| \cdots |V_k|)$, denoted as T, we define $a = \max_{1 \le i \le k}\{|V_i|\}$. /*** Dave's comment [4]: is there any reason for choosing 'a'? Can we give an explanation? ***/

We first analyze the time complexity of selecting the i-th $(i = 1, 2, \cdots, N)$ test case, which depends on two factors: (1) the number of candidates required for the calculation of UVCD; and (2) the time complexity of calculating UCVD of strength l $(1 \le l \le t)$ for each candidate during the process of constructing A_l.

For (1), it requires $(N-i)+1$ test cases to compute UVCD. For (2), according to C_k^l l-wise parameter combinations, we divide all possible l-wise value combinations that are derived from a $TP(k, |V_1||V_2| \cdots |V_k|)$ into C_k^l sets that form

$$\Pi_l = \{\pi_l | \pi_l = \{(v_{i_1}, v_{i_2}, \cdots, v_{i_l}) | v_{i_j} \in V_{i_j}, j = 1, 2, \cdots, l\}, \\ 1 \le i_1 < i_2 < \cdots < i_l \le k\}. \tag{9}$$

As a consequence, when using a binary search, the order of time complexity of (2) is $O(\sum_{\pi_l \in \Pi_l} \log(|\pi_l \setminus CombSet_l(T)|))$, which equals $O(\sum_{\pi_l \in \Pi_l} \log(|\pi_l|))$. Let us define the following function:

$$f_l = \begin{cases} 0, & \text{if } l = 0; \\ \sum_{i=1}^{l} |A_l|, & \text{if } 1 \le l \le t. \end{cases} \tag{10}$$

From Equation 10, we have $f_t = \sum_{l=1}^{t} |A_l| = N$.

According to $A_l (1 \le l \le t)$, the order of time complexity of constructing A_l is $O((\sum_{i=f_{l-1}+1}^{f_l} (N - i + 1)) \times (\sum_{\pi_l \in \Pi_l} \log(|\pi_l|)))$. Since t subparts $A_1, A_2, \cdots, A_t$

are included in the algorithm IICBP execution, the order of time complexity can be described as follows:

$$O(IICBP) = O(\sum_{l=1}^{t}((\sum_{i=f_{l-1}+1}^{f_l}(N-i+1)) \times (\sum_{\pi_l \in \Pi_l} \log(|\pi_l|))))$$
$$< O(\sum_{l=1}^{t}((\sum_{i=f_{l-1}+1}^{f_l}(N-i+1)) \times (C_k^l \times \log(a^l))))(1 \le l \le t). \quad (11)$$

There exists an integer $\mu(1 \le \mu \le t)$ such that:

$$(\forall l)\ (1 \le l \le t)\ (\mu \ne l)\ [(C_k^\mu \times \log(a^\mu)) \ge (C_k^l \times \log(a^l))]. \quad (12)$$

As a consequence,

$$O(IICBP) < O(\sum_{l=1}^{t}((\sum_{i=f_{l-1}+1}^{f_l}(N-i+1)) \times (C_k^\mu \times \log(a^\mu))))$$
$$= O((\sum_{i=1}^{N}(N-i+1)) \times (C_k^\mu \times \log(a^\mu)))$$
$$= O(C_k^\mu \times \log(a^\mu) \times (N^2+N)/2). \quad (13)$$

Therefore, we can conclude that the order of time complexity of algorithm IICBP is $O(N^2 \times C_k^\mu \times \log(a^\mu))(1 \le \mu \le t)$.[a]

As discussed in [15], the order of time complexity of algorithm ICBP (Algorithm 2) is $O(N^2 \times C_k^t \times \log(a^t))$. /*** Dave's comment [5]: please confirm the next sentence ***/ According to Appendix A, if $1 \le t < \lceil \frac{k}{2} \rceil$, $\mu = t$, or if $\lceil \frac{k}{2} \rceil \le t \le k$, $\mu = \lceil \frac{k}{2} \rceil$, then the order of time complexity of algorithm IICBP is the same as that of algorithm ICBP.

4. Experimental Results

In this section, some experimental results from simulations and experiments with real programs are presented to analyze the effectiveness of the prioritization of combinatorial test cases by incremental interaction coverage. We evaluate test sequences prioritized by algorithm IICBP (denoted `IICBP`) by comparing with those ordered by three other strategies: (1) test sequence according to covering array generation sequence (denoted `Original`); (2) random test sequence whose ordering is randomly prioritized (denoted `Random`); and (3) test sequence prioritized by algorithm ICBP (denoted `ICBP`).

[a]If $1 \le t < \lceil \frac{k}{2} \rceil$, $\mu = t$; if $\lceil \frac{k}{2} \rceil \le t \le k$, $\mu = \lceil \frac{k}{2} \rceil$, see Appendix A for more details.

Table 3. Sizes of covering arrays for four test profiles.

Test profile	ACTS					PICT				
	2	3	4	5	6	2	3	4	5	6
$TP(6, 5^6)$	25	199	1058	4149	15625	37	215	1072	4295	15625
$TP(10, 2^3 3^3 4^3 5^1)$	23	103	426	1560	3590	23	109	411	1363	3934
$TP(8, 2^6 9^1 10^1)$	90	180	632	1080	2520	90	192	592	1237	2370
$TP(7, 2^4 3^1 6^1 16^1)$	96	289	578	1728	2304	96	293	744	1658	2655

4.1. *Simulations*

We initially designed some typical test profiles to construct covering arrays, then applied different test case prioritization techniques to prioritize them, evaluating each prioritization strategy Three simulations were involved. The first simulation was to evaluate the rate of value combinations covered by the different prioritization techniques. The second and third simulations aimed at assessing rates of fault detection for different test sequences when executing all, or some test cases, respectively.

4.1.1. *Simulation instrumentation*

We designed four test profiles as four system models with details shown in Table 3. The first two test profiles were $TP(6, 5^6)$ and $TP(10, 2^3 3^3 4^3 5^1)$, both of which have been used in previous studies [15]. The third and fourth test profiles (that is, $TP(8, 2^6 9^1 10^1)$ and $TP(7, 2^4 3^1 6^1 16^1)$) were from real-world applications: a real configuration model of GNUzip (gzip); and a module of a lexical analyzer system (flex).

The original covering arrays were generated by two different tools: *Advanced Combinatorial Testing System* (ACTS) [25, 26] and *Pairwise Independent Combinatorial Testing* (PICT) [27]. Both of these are supported by greedy algorithms, and implemented, respectively, by the *In-Parameter-Order* (IPO) method [25] and the *one-test-at-a-time* approach (generating one test case each time) [28]. We focused on covering arrays with strength $t = 2, 3, 4, 5, 6$; and the sizes of the covering arrays generated by ACTS or PICT are given in Table 3. Since randomization is used in some test case prioritization techniques, we ran each test profile 100 times and report the average of the results.

4.1.2. *Simulation One: Rate of covering value combinations*

In this simulation, we measured how quickly a test sequence could cover value combinations of different strengths. We only considered strength $t = 2, 3, 4$. Algorithm ICBP requires that the strength t be initialized in advance. However, because we sometimes may not know the strength of a covering array in practical testing applications, we also take account of test sequences ordered by algorithm ICBP when selecting lower strength $\tau (1 < \tau < t)$, that is, $\texttt{ICBP}_\tau$.

1. Metrics: *Average percentage of combinatorial coverage* (APCC) [15] is used as the metric to evaluate the rate of value combinations covered by a test sequence. The APCC values range from 0% to 100%; higher APCC values mean better rates of covering value combinations. Let a test sequence be $S = \langle s_1, s_2, \cdots, s_N \rangle$, obtained by prioritizing a $CA(N; t, k, |V_1||V_2| \cdots |V_k|)$, the formula for APCC at strength τ is given as follows:

$$APCC_\tau(S) = \frac{\sum_{i=1}^{N-1} |\bigcup_{j=1}^{i} CombSet_\tau(s_j)|}{N \times |CombSet_\tau(T_{all})|}, \tag{14}$$

where T_{all} is the set of all test cases from $TP(k, |V_1||V_2| \cdots |V_k|)$.

2. Results and analysis: For covering arrays of strength $t(2 \le t \le 4)$ on individual

Table 4. $APCC_\tau$ metric (%) for different prioritization techniques for $TP(6, 5^6)$.

	Method	$t=2$		$t=3$			$t=4$			
		τ=1	τ=2	τ=1	τ=2	τ=3	τ=1	τ=2	τ=3	τ=4
	Original	82.67	48.00	93.80	85.11	63.47	94.93	89.61	82.68	63.59
A	Random	82.75	48.00	97.54	80.63	58.62	99.53	97.69	89.20	59.99
C	$ICBP_t$	82.96	48.00	97.71	89.94	64.31	99.54	97.88	91.36	65.39
T	$ICBP_{t-1}$	NA	NA	98.16	92.17	59.40	99.57	98.40	92.63	60.80
S	$ICBP_{t-2}$	NA	NA	NA	NA	NA	99.66	98.62	89.43	59.98
	IICBP	85.87	48.00	98.45	92.03	63.61	99.71	98.60	92.49	64.90
	Original	90.63	60.27	98.16	92.11	64.40	99.59	97.83	91.41	64.56
P	Random	87.52	56.35	97.70	89.39	60.26	99.53	97.73	89.37	60.32
I	$ICBP_t$	89.95	60.27	97.91	91.79	64.58	99.55	97.90	91.53	65.28
C	$ICBP_{t-1}$	NA	NA	98.30	92.81	60.93	99.59	98.39	92.76	61.32
T	$ICBP_{t-2}$	NA	NA	NA	NA	NA	99.67	98.64	89.65	60.34
	IICBP	91.19	60.00	98.58	92.70	64.23	99.72	98.62	92.63	64.86

Table 5. $APCC_\tau$ metric (%) for different prioritization techniques for $TP(10, 2^3 3^3 4^3 5^1)$.

	Method	$t=2$		$t=3$			$t=4$			
		τ=1	τ=2	τ=1	τ=2	τ=3	τ=1	τ=2	τ=3	τ=4
	Original	86.14	66.55	92.72	85.33	72.17	97.06	88.66	82.99	73.82
A	Random	86.15	62.75	96.69	89.52	70.98	99.19	97.28	91.45	76.11
C	$ICBP_t$	88.60	67.32	97.56	91.85	74.99	99.36	98.03	93.51	79.98
T	$ICBP_{t-1}$	NA	NA	97.67	92.23	72.48	99.42	98.09	93.80	77.97
S	$ICBP_{t-2}$	NA	NA	NA	NA	NA	99.45	98.18	92.14	76.35
	IICBP	89.31	66.90	97.73	92.06	74.15	99.47	98.15	93.61	79.38
	Original	88.18	66.51	97.56	92.21	76.23	99.08	97.44	92.55	78.56
P	Random	86.16	63.23	96.90	90.05	72.10	99.15	97.18	91.22	75.45
I	$ICBP_t$	88.64	66.82	97.90	92.36	76.10	99.34	97.95	93.26	79.17
C	$ICBP_{t-1}$	NA	NA	97.80	92.67	73.70	99.40	98.02	93.56	77.24
T	$ICBP_{t-2}$	NA	NA	NA	NA	NA	99.43	98.11	91.89	75.71
	IICBP	89.12	66.43	97.80	92.51	75.51	99.45	98.08	93.37	78.52

Table 6. $APCC_\tau$ metric (%) for different prioritization techniques for $TP(8, 2^6 9^1 10^1)$.

	Method	$t=2$		$t=3$			$t=4$			
		τ=1	τ=2	τ=1	τ=2	τ=3	τ=1	τ=2	τ=3	τ=4
	`Original`	82.87	69.62	83.53	71.77	62.21	90.80	84.17	79.60	72.11
A	`Random`	93.26	76.03	96.38	85.75	64.66	98.96	95.14	86.82	71.32
C	$\texttt{ICBP}_\texttt{t}$	95.58	79.94	97.32	84.49	69.40	99.26	96.50	90.60	76.65
T	$\texttt{ICBP}_\texttt{t-1}$	NA	NA	97.63	89.79	65.85	99.36	97.06	91.14	74.47
S	$\texttt{ICBP}_\texttt{t-2}$	NA	NA	NA	NA	NA	99.39	97.14	88.37	71.98
	`IICBP`	95.73	79.79	97.89	89.58	66.49	99.40	97.12	90.55	75.33
	`Original`	93.94	78.62	97.08	88.32	71.00	98.91	95.37	88.47	74.28
P	`Random`	93.17	75.82	96.71	86.45	66.87	98.90	94.84	86.18	70.87
I	$\texttt{ICBP}_\texttt{t}$	95.51	79.94	97.58	88.88	72.20	99.21	96.40	89.94	75.43
C	$\texttt{ICBP}_\texttt{t-1}$	NA	NA	97.93	89.99	70.06	99.33	96.89	90.53	73.62
T	$\texttt{ICBP}_\texttt{t-2}$	NA	NA	NA	NA	NA	99.35	96.97	87.69	71.50
	`IICBP`	95.76	79.59	98.02	89.85	70.74	99.36	96.94	89.95	74.50

Table 7. $APCC_\tau$ metric (%) for different prioritization techniques for $TP(7, 2^4 3^1 6^1 16^1)$.

	Method	$t=2$		$t=3$			$t=4$			
		τ=1	τ=2	τ=1	τ=2	τ=3	τ=1	τ=2	τ=3	τ=4
	`Original`	75.54	63.40	76.46	65.40	58.65	76.65	65.68	59.50	55.18
A	`Random`	91.07	69.82	96.85	87.64	68.32	98.38	93.26	81.98	61.31
C	$\texttt{ICBP}_\texttt{t}$	93.98	75.77	97.79	90.91	73.82	98.66	94.52	84.12	64.78
T	$\texttt{ICBP}_\texttt{t-1}$	NA	NA	98.08	92.11	70.08	98.96	95.73	86.39	62.70
S	$\texttt{ICBP}_\texttt{t-2}$	NA	NA	NA	NA	NA	99.04	96.11	83.45	61.67
	`IICBP`	94.47	75.01	98.16	91.86	72.42	99.08	96.02	85.63	62.62
	`Original`	92.58	74.52	97.25	88.47	72.28	98.70	94.74	86.57	70.84
P	`Random`	91.12	71.04	96.89	87.81	69.80	98.72	94.62	85.05	67.62
I	$\texttt{ICBP}_\texttt{t}$	94.17	76.27	97.90	91.36	75.02	99.05	96.24	88.87	72.79
C	$\texttt{ICBP}_\texttt{t-1}$	NA	NA	98.14	92.19	71.59	99.19	96.80	89.89	70.36
T	$\texttt{ICBP}_\texttt{t-2}$	NA	NA	NA	NA	NA	99.27	97.00	86.33	68.02
	`IICBP`	94.47	75.66	98.18	91.87	73.74	99.28	96.87	89.12	71.22

test profiles, we have the following observations based on the results reported in Tables 4–6. Each table corresponds to a particular test profile.

(a) Combinatorial test sequences prioritized by `IICBP` strategy have greater $\text{APCC}_\tau(1 \leq \tau \leq t)$ values than `Original` test sequences and `Random` test sequences. Therefore, the `IICBP` technique outperforms `Original` and `Random`.
(b) Given a covering array of strength t, the $\texttt{ICBP}_\tau$ has the highest APCC_τ when $1 < \tau \leq t$; but the `IICBP` has the highest $\text{APCC}_{\tau'}$ when $1 \leq \tau' \neq \tau \leq t$.
(c) The ACTS `Original` test sequences often have lower APCC values than `Random` test sequences; the PICT `Original` test sequences always outperform `Random` test sequences, and occasionally outperform $\texttt{ICBP}_\texttt{t-1}$ and $\texttt{ICBP}_\texttt{t-2}$ test sequences.

Observation (a) is easily explained, hence, we just explain the second and the third observations here. For observation (b), since ICBP prioritizes combinatorial test cases by using strength τ, therefore its APCC value is the highest at strength τ. However, IICBP comprehensively considers different strengths for prioritizing test cases, and hence it has the highest APCC values at other strength values.

For observation (c), the difference in performance is due to the different mechanisms involved in implementing ACTS and PICT. For example, without loss of generality, suppose we have a $TP(k, |V_1||V_2|\cdots|V_k|)$ with $|V_1| \geq |V_2| \geq \cdots \geq |V_k|$. The ACTS algorithm first uses *horizontal growth* [25,26] to build a t-wise ($2 \leq t \leq k$) test set for the first t parameters. This implies that it needs at least $1+(|V_1|-1)\prod_{i=2}^{t}|V_i|$ test cases to cover all 1-wise value combinations. On the other hand, PICT selects a next test case such that it covers the largest number of t-wise value combinations that have not been covered – a mechanism similar to that of ICBP.

In summary, given a covering array of strength t, IICBP strategy performs better than Original and Random strategies with respect to $\text{APCC}_\tau(1 \leq \tau \leq t)$, and performs better than ICBP technique of strength τ in for strengths that are not equal to τ.

4.1.3. *Simulation Two: Rate of fault detection when executing all test cases*

In the second simulation, we modeled four systems with a number of failures by using the same four test profiles as in Section 4.1.1 to analyze the fault detection rate of each prioritization technique when executing all test cases in a covering array.

With regard to the distribution of failures, we assigned some failures at lower strengths according to results reported in [1,6]. For example, in [1], several software projects were studied and the interaction faults were reported to have 29% to 82% faults as 1-wise faults (that is, the FTFI number is 1); 6% to 47% of faults as 2-wise faults; 2% to 19% as 3-wise; 1% to 7% of faults as 4-wise; and even fewer faults beyond 4-wise interactions. Therefore, in our simulation we only considered simulated interaction faults of the FTFI number $= 1, 2, 3, 4$. As a result, the fault distribution simulated for each test profile was designed as following: thirty 1-wise interaction faults; forty 2-wise interaction faults; twenty 3-wise interaction faults; and five 4-wise interaction faults. Each injected fault was randomly generated with replacement in individual test profiles. Since the simulated interaction fault was randomly chosen and some prioritization strategies involved some randomization, we ran each algorithm 100 times for each test profile, and report the average of the results.

1. Metrics: The APFD metric [18] is often used to evaluate fault detection rates of different prioritization techniques. However, this metric does have a requirement that all faults should be detected by a given test sequence. In other words, if a fault cannot be detected, the APFD metric fails. The *normalized APFD* metric

(NAPFD) [13] has been proposed as an enhancement of APFD. It includes information about both fault finding and time of detection. The higher the NAPFD value, the higher the fault detection rate. Similar to the definition of APFD given in Equation 5, the formula for NAPFD is presented as follows:

$$NAPFD = p - \frac{SF_1 + SF_2 + \cdots + SF_m}{n \times m} + \frac{p}{2n}, \tag{15}$$

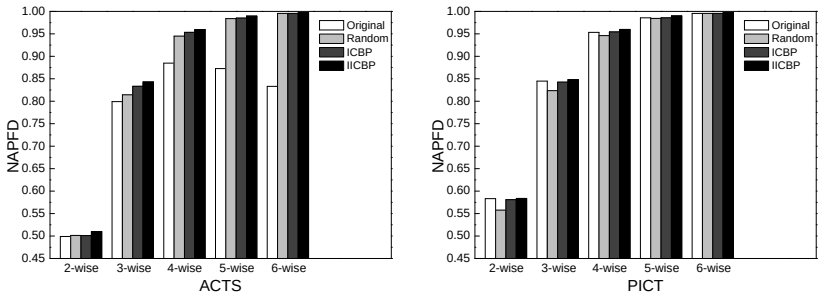
where m, n, and $SF_i (i = 1, 2, \cdots, m)$ have the same interpretations as in APFD, and p represents the ratio of the number of faults identified by selected test cases relative to the number of faults detected by the full test suite. If a fault, f_i, is never found, we set $SF_i = 0$. Obviously, if all faults can be detected, NAPFD and APFD are identical due to $p = 1.0$.

2. Results and analysis: Fig. 2 presents the simulation results in terms of the NAPFD metric values for different prioritization techniques, based on which we have the following observations.
   (a) The `IICBP` technique has significantly better fault detection rates than the ACTS `Original` method, but only slightly better performance than the PICT `Original` method.
   (b) The `IICBP` test sequences have higher NAPFD values than the `Random` test sequences.
   (c) Compared to `ICBP`, `IICBP` has similar NAPFD values. More specifically, when prioritizing the covering arrays of strength $t = 2$, the `IICBP` failure-detection rate is sometimes slightly less than that of `ICBP`; when ordering the covering arrays of strength $t > 2$, `IICBP` performs slightly better than `ICBP`.

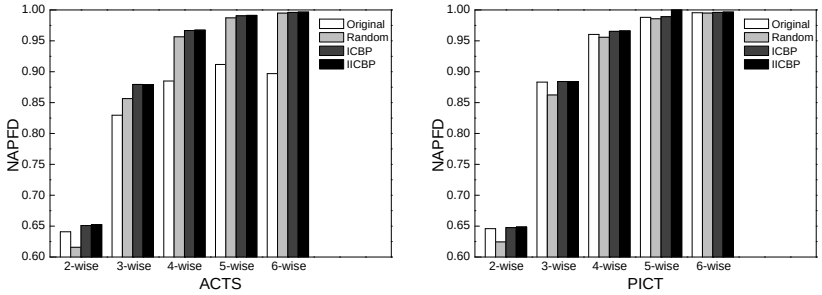
The first and second observations are consistent with those reported for Simulation One. For Observation (c), we take a covering array of strength $t = 5$ generated by PICT on $TP(10, 2^3 3^3 4^3 5^1)$ as an example. Table 8 shows the average number of test cases required to find all faults at different FTFI numbers. We can observe that for any FTFI number, `IICBP` performs better than other methods. However, since the size of the original test suite (1363) is much larger than any value shown in Table 8, the difference among NAPFD values obtained by different methods is

Table 8. The average number of test cases in different sequences required to detect all faults for different FTFI numbers.

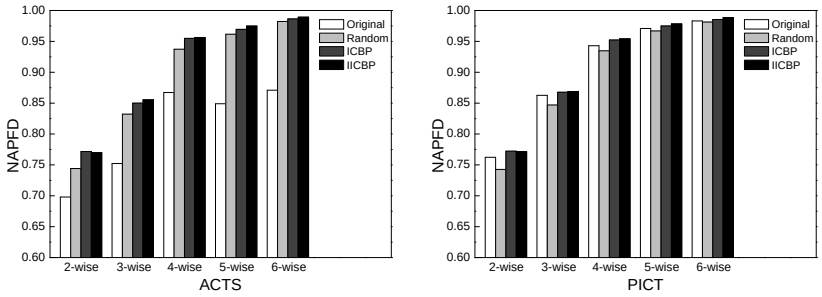
Method	FTFI number			
	1	2	3	4
`Original`	8.97	49.02	120.78	235.43
`Random`	10.39	52.75	155.64	278.14
`ICBP`	8.33	33.62	104.09	216.60
`IICBP`	4.65	22.81	81.89	201.50



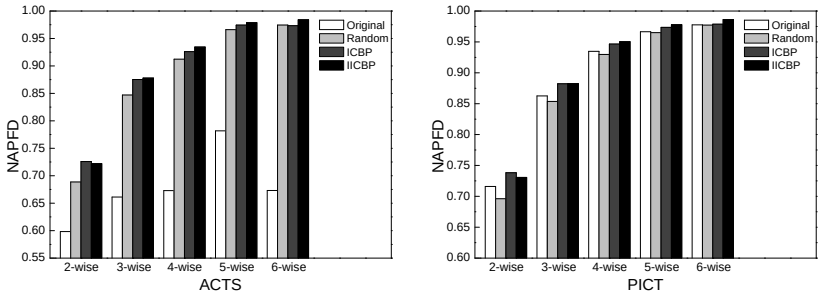
(a) $TP(6, 5^6)$



(b) $TP(10, 2^3 3^3 4^3 5^1)$



(c) $TP(8, 2^6 9^1 10^1)$



(d) $TP(7, 2^4 3^1 6^1 16^1)$

Fig. 2. NAPFD metric for different prioritization techniques when executing all test cases.

smaller. Therefore, IICBP may have similar NAPFD values to ICBP, and sometimes is similar to Random and Original, when executing all test cases. /*** Dave's comment [6]: Please check the last paragraph ***/

4.1.4. *Simulation Three: Rate of fault detection when executing part of the test suite*

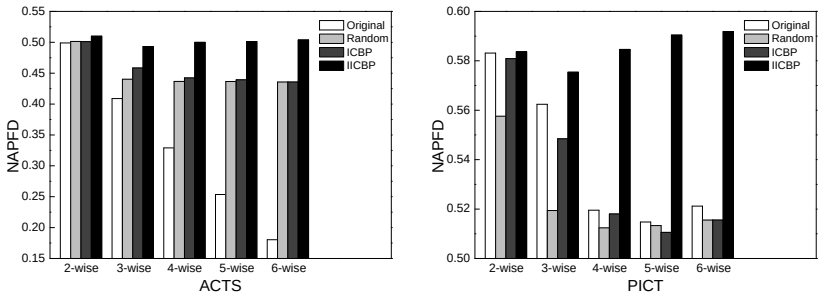
Since resources are limited, in practice it is often the case that not all test cases in a test suite (or test sequence) are executed. In this simulation, we focused on the fault detection rates of different test case prioritization techniques when running only part of a given test sequence.

The simulation design was consistent with that of Simulation Two, as explained in Section 4.1.3, including fault distribution and fault generation. With regard to the portion of the test sequence to be executed, we followed the practice adopted in previous prioritization studies [13] of fixing the number of test cases that would be executed to be the size of a covering array at strength $t = 2$. For instance, consider $TP(6, 5^6)$ in Table 3: for any strength, the 25 ACTS test cases and 37 PICT test cases were chosen to be executed in each test sequence generated by each method.

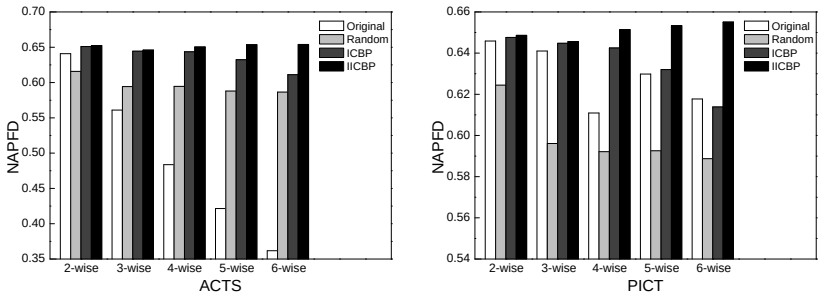
1. Metrics: Similar to Simulation Two, the NAPFD metric (Equation 15) was also used to evaluate fault detection rates of different prioritization strategies when executing part of test suite. Here, it is should be noted that n in Equation 15 is the number of executed test cases rather than the number of all test cases in a given test sequence.
2. Results and analysis: The NAPFD values for the different prioritization methods are summarized in Fig. 3, from which the following observations can be made.
   (a) The NAPFD values for the Random test sequences were higher than those for ACTS Original, but lower than those for the PICT Original test sequences. This observation is consistent with those reported for the other simulations.
   (b) IICBP outperforms Original, Random, and ICBP in most cases.
   (c) With the increase of strength, the improvement of IICBP over Original, Random, and ICBP increases significantly. In other words, when the strength is larger, IICBP is more suitable for prioritizing combinatorial test suites than Original, Random, or ICBP.

In summary, according to the APCC and NAPFD metrics, the IICBP technique performs better than the Original and Random techniques. Compared with ICBP, IICBP performs better at low strengths in terms of APCC metric values. However, IICBP may produce test sequences with similar NAPFD metric values to those of ICBP when executing all test cases, but with better NAPFD metric values when running only part of the test suite.

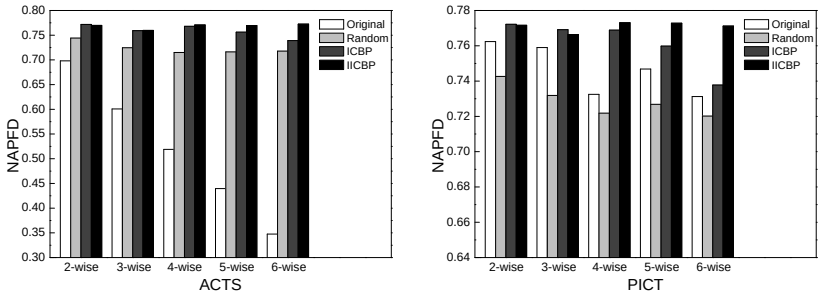
Obviously, two faults with the same faulty interaction may have different properties. For example, given a $TP(k, |V_1||V_2|\cdots|V_k|)$, faults f_1 and f_2 can be both



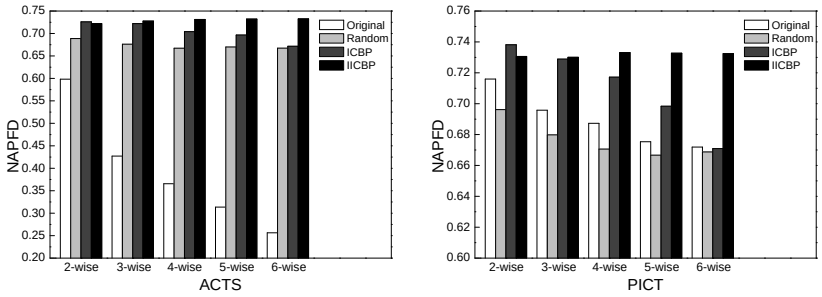
(a) $TP(6, 5^6)$



(b) $TP(10, 2^3 3^3 4^3 5^1)$



(c) $TP(8, 2^6 9^1 10^1)$



(d) $TP(7, 2^4 3^1 6^1 16^1)$

Fig. 3. NAPFD metric for different prioritization techniques when executing only part of the test suite.

identified by a 2-wise faulty interaction $\{P_1, P_2\}$. Fault f_1 may be triggered when "$(P_1 = v_1)\&\&(P_2 = v_2)$" where $v_1 \in V_1$ and $v_2 \in V_2$; while fault f_2 may be triggered by "$(P_1 \neq v_1)\&\&(P_2 \neq v_2)$". Consider a test case, its probability of revealing fault f_1 (or the failure rate of f_1 – the number of failure-causing test cases revealing f_1 as a proportion of all possible tests) is $\frac{1}{|V_1|\times|V_2|}$, and the probability of revealing fault f_2 is $\frac{(|V_1|-1)\times(|V_2|-1)}{|V_1|\times|V_2|}$. When parameters P_1 and P_2 both have a large number of possible values, the probabilities of detecting f_1 and f_2 could be very different. In Simulation Two and Simulation Three, the faulty interaction of each simulated fault was consistent with that of fault f_1, that is, each fault could only be detected by a special value combination rather than different value combinations. /*** Dave's comment [7]: please confirm the next sentence ***/ As for faults that differ from fault f_1, the effectiveness of our method will be investigated later by studying some real-life programs.

4.2. *An empirical study*

4.2.1. *Experiment instrumentation*

/*** Dave's comment [8]: should we remove all mention of cmdline? ***/

We used six C programs (count, series, tokens, ntree, nametbl and cmdline), downloaded from `http://www.maultech.com/chrislott/work/exp/`, as subject programs [29]. These programs were originally created to support research on comparison of defect revealing mechanisms [29], evaluation of different combination strategies for test case selection [30], and fault diagnosis [31, 32]. Each program contains some faults. To determine the correctness of an executing test case, i.e. an oracle, we created a fault-free version of each program by analyzing the corresponding fault description.

Table 9 describes these subject programs. The third column (LOC) stands for the number of lines of executable code in these programs; while the fifth column (No. of detectable faults) represents the number of faults detected by some test cases derived from the accompanying test profiles, which are not guaranteed to be able to detect all faults. By analyzing the detectable faults, as shown in Table 9, we summarize them according to the FTFI number of each fault. Similar to [30], due to the incomplete specifications of cmdline, it was not included in this study.

Table 9. Subject programs.

Subject	Test profile	LOC	No. of faults	No. of detectable faults	FTFI number				
					0	1	2	3	4
count	$TP(6, 2^1 3^5)$	42	15	12	0	4	4	4	0
series	$TP(3, 5^2 7^1)$	288	23	22	1	3	4	14	NA
tokens	$TP(8, 2^4 3^4)$	192	15	11	1	4	5	1	0
ntree	$TP(4, 4^4)$	307	32	24	0	5	11	7	1
nametbl	$TP(5, 2^1 3^2 5^2)$	329	51	44	1	17	24	2	0

Table 10. Sizes of original test sequences for each subject program.

Subject program	ACTS					PICT				
	2	3	4	5	6	2	3	4	5	6
count	15	41	108	243	486	14	43	116	259	486
series	35	175	NA	NA	NA	39	175	NA	NA	NA
tokens	12	37	93	212	486	12	39	103	228	482
ntree	20	64	256	NA	NA	19	75	256	NA	NA
nametbl	25	82	225	450	NA	25	78	226	450	NA

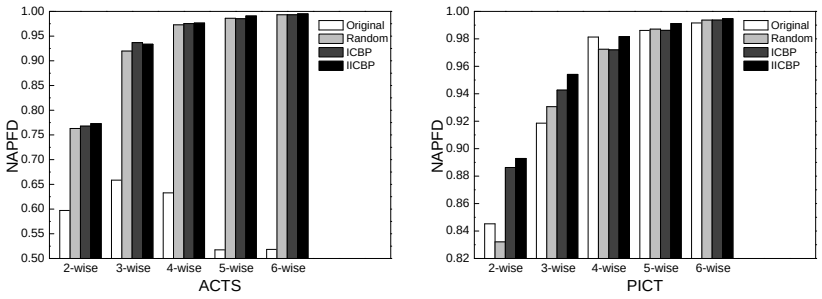
Similar to the simulations described above, we also used ACTS and PICT to generate original test sequences for each subject program. Moreover, we focused on covering arrays with strength $t = 2, 3, 4, 5, 6$. Table 10 shows the sizes of the original test sequences obtained by ACTS and PICT. For the effectiveness metrics, we used NAPFD for respectively executing all test cases and a subset of the entire test suite such that the size of the subset was equal to each covering array of strength $t = 2$. Due to randomization in some prioritization techniques, we ran the experiment 100 times for each subject program and report the average.

4.2.2. *Results and analysis*

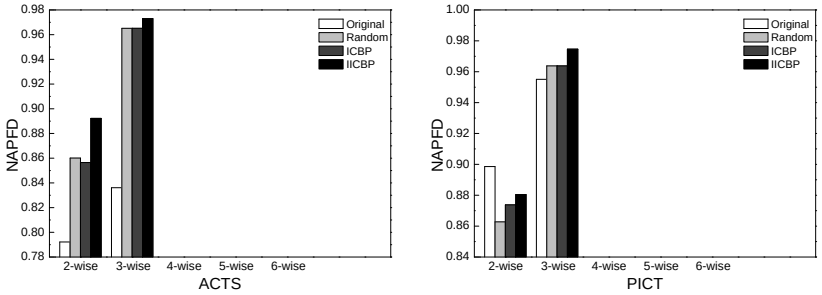
The experimental results from running all prioritization techniques to test count, series, tokens, ntree, and nametbl, are summarized in Figs. 4 and 5.

1. When executing all test cases in the test suite, as shown in Figs. 4(a)–4(e), we have the following observations: (a) for all test suites at strength $t = 3, 4, 5, 6$, `IICBP` performs significantly better than `Original` using ACTS, and has slightly better performance than `Random` and `ICBP`, regardless of whether using ACTS or PICT; and (b) for strength $t = 2$ test suites, no conclusive observations could be obtained.
2. When executing part of the test suite, as illustrated in Figs. 5(a)–5(e), it can be observed that for four programs (count, series, ntree, and nametbl), the performance of the various prioritization strategies was very similar: (1) in most cases, `IICBP` had higher NAPFD metric values than `Original`, `Random`, and `ICBP`; (2) with the increase of strength, the improvement of `IICBP` over `Original`, `Random`, or `ICBP` generally increased; (3) the `Original` ACTS test sequences performed worst in terms of fault detection rate, while the `Original` PICT test sequences sometimes have the largest NAPFD values, such as for 2-wise series and 3-wise ntree; (4) for covering arrays of strength $t = 2$ on nametbl, `ICBP` has the best performance in terms of the rate of fault detection. These observations are basically consistent with those for the simulations.

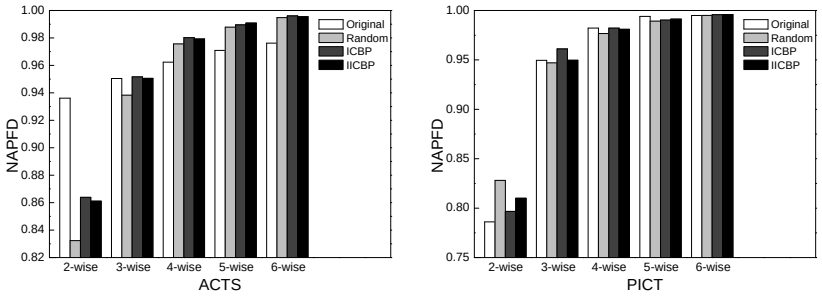
   For the remaining program (tokens), no conclusive remarks could be drawn. As observed, each prioritization method may sometimes perform best, and may sometimes perform worst.



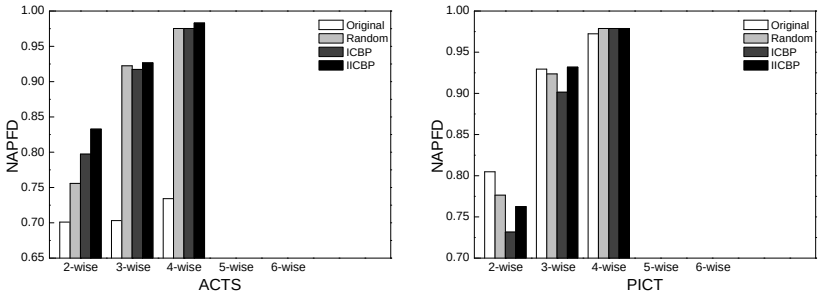
(a) Program count



(b) Program series

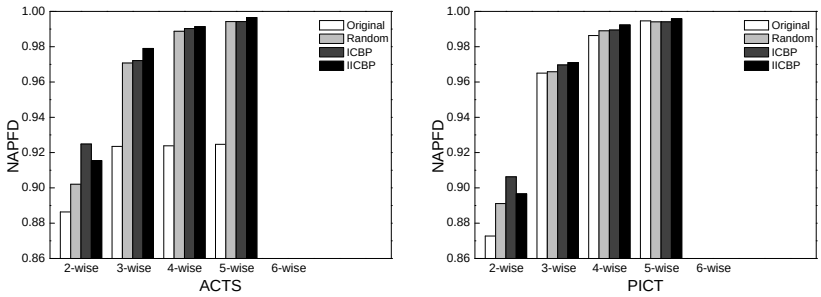


(c) Program tokens



(d) Program ntree

Fig. 4. NAPFD metric for different prioritization techniques for five real programs when executing all test cases



(e) Program nametbl

Fig. 4. (*Continued*).

In summary, the experimental study using real programs shows similar results to the simulations in terms of the rate of fault detection, that is, when executing all test cases in the combinatorial test suite, `IICBP` had similar performance to `Original`, `Random`, and `ICBP` generally; while `IICBP` performed better than others in most cases when executing only part of the test suite.

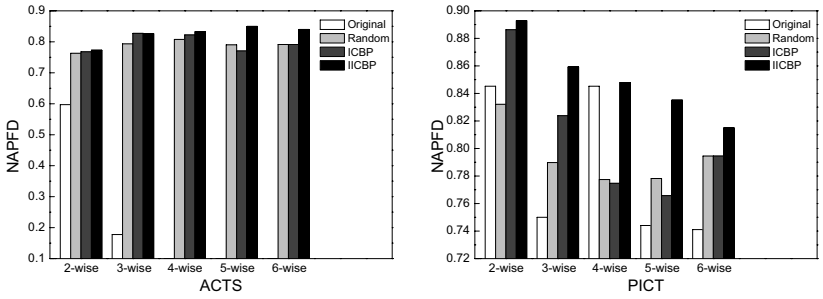
4.3. *Threats to validity*

Despite our best efforts, our experiments may face some threats to validity. In this section, we present the most significant of these, which are classified into three categories: (1) threats to external validity; (2) threats to internal validity; and (3) threats to construct validity.

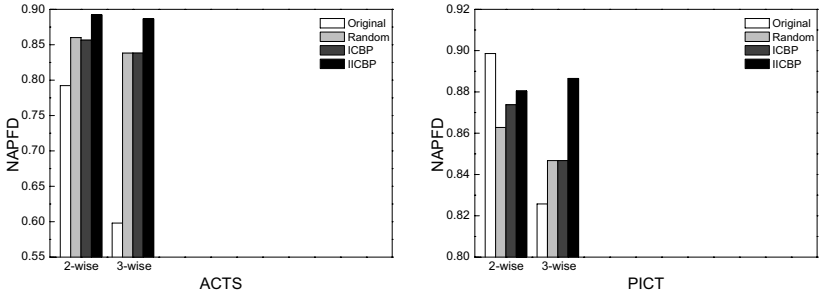
External validity refers specifically to what extent our experimental results can be generalized. We mainly outline three threats to external validity: (1) Test profile representativeness – in our study, four widely used, but limited test profiles were employed; (2) Subject program representativeness – we have examined only five subject programs, written in the C language, all of which are of relatively small size; and (3) Covering array generation representativeness – in our experiment, we used ACTS and PICT for generating different covering arrays, however, both of these belong to the category of greedy algorithm [7]. To address these potential threats, additional studies using a greater range of test profiles, a greater number of subject programs, and different algorithms for covering array construction will be conducted in the future.

Internal validity refers to whether or not there were mistakes in the experiments. We have tried to manually cross-validate our analyzed programs on small examples, and we are confident of the correctness of the experimental and simulation setups.

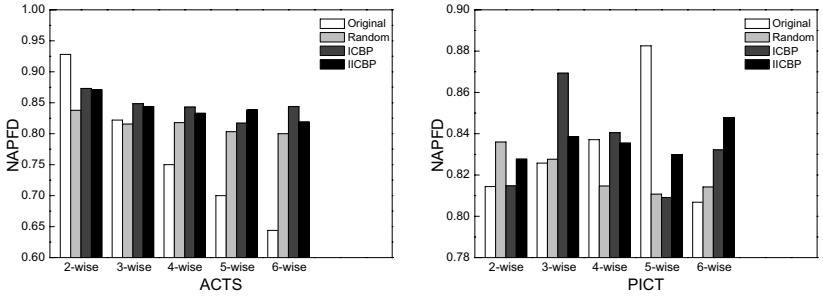
Finally, construct validity refers to whether or not we have conducted the studies fairly. In this article, we focus on the rate of covered value combinations and the rate of fault detection, measured with the APCC and NAPFD (or APFD) metrics, respectively. The NAPFD and APFD metrics are commonly used in the study of



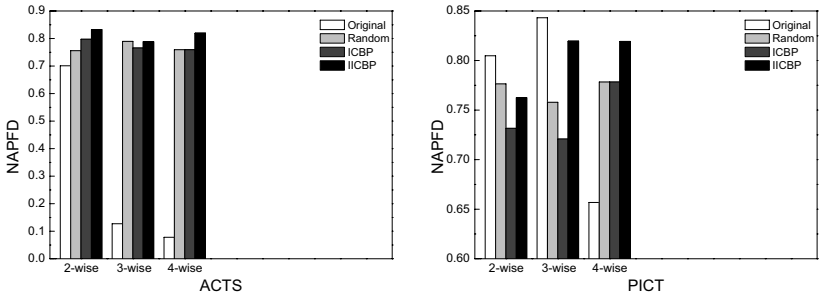
(a) Program count



(b) Program series

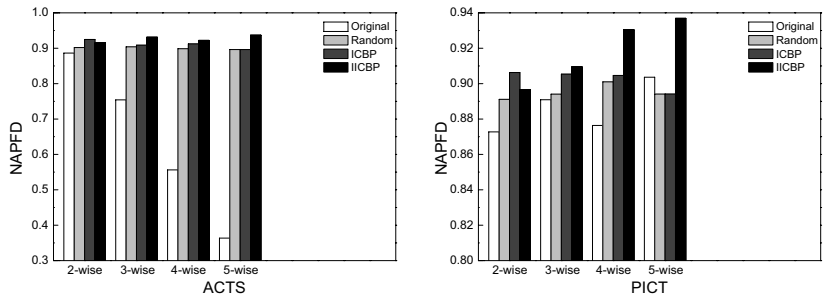


(c) Program tokens



(d) Program ntree

Fig. 5. NAPFD metric for different prioritization techniques for five real programs when executing only part of the test suite.



(e) Program nametbl

Fig. 5. (*Continued*).

Table 11. State of the art in combinatorial test case prioritization.

Strategies	Interaction coverage	Incremental interaction coverage
Pure prioritization	[4], [11], [12], [13], [15]	[33], Focus of this paper
Re-generation prioritization	[4], [9], [10], [13], [14]	[3], [5]

test case prioritization.

5. Related Work

Techniques for prioritizing combinatorial test cases have been well-researched in recent years, and can generally be divided into two categories: (1) pure prioritization: re-prioritizing test cases in the combinatorial test suite; and (2) re-generation prioritization: taking account of prioritization in the process of combinatorial test case generation [13]. The method proposed in this paper belongs to the first category.

From the perspective of interaction coverage, there are a large number of strategies supporting prioritization of combinatorial test cases. For example, Bryce and Colbourn [9, 10] proposed generating prioritized combinatorial test suites by assigning weights to each pairwise interaction of parameters, a technique in the re-generation prioritization category. Bryce and her colleagues [11, 12] introduced a technique of re-prioritizing combinatorial test cases based on interaction coverage, and applied this technique to event-driven software. Qu *et al.* [13] presented how to assign parameter combination weights that evaluate their importance, and also applied interaction-coverage-based prioritization strategies to configurable systems [4]. Chen *et al.* [14] used a re-generation prioritization strategy to construct combinatorial test sequences by applying the ant colony algorithm. Furthermore, Wang *et al.* [15] proposed a series of metrics for evaluating combinatorial test sequences by considering different factors such as test case cost and weight, and also introduced two heuristic algorithms in the pure prioritization category.

On the other hand, fewer studies have been conducted on the prioritization of combinatorial test cases from the perspective of incremental interaction coverage. Fouché *et al.* [3, 5] have recently proposed a technique named *incremental covering array failure characterization* (ICAFC), where incremental interaction coverage is used to generate incremental adaptive covering arrays. ICAFC starts at a low strength for constructing a covering array, and gradually increases strength by reusing previous test cases until some conditions are satisfied. However, an incremental adaptive covering array of strength t generated by ICAFC may be considered a prioritized combinatorial test suite only from the viewpoint of strength. We will discuss this issue further in the next section. Furthermore, Wang [33] has developed the technique of inCTPri to generate the prioritized combinatorial test cases. However, his inCTPri assumes covering arrays as inputs, while our method is applicable on any combinatorial test suite including covering arrays. Additionally, our method begins at strength $t = 1$ while inCTPri starts at a small strength value greater than 1.

The state of the art in combinatorial test case prioritization is summarized in Table 11, from which it can be seen that the topic has been extensively researched from the perspective of interaction coverage, but has received far less attention from the perspective of incremental interaction coverage. Our investigation (highlighted in the table) attempts to fill this gap in the research.

6. Discussion and Conclusion

Combinatorial testing has been widely used in practice, and test case prioritization has also been well studied. Prioritization of combinatorial test cases is a popular research area. This paper proposes a new strategy of prioritizing combinatorial test cases based on the intuition of incremental interaction coverage, which is a balanced strategy compared with traditional interaction-coverage-based test prioritization. Experimental results show that our method outperforms the random prioritization approach and the technique of prioritizing combinatorial test suites according to test case generation order, and has better performance than the ICBP technique in most scenarios, with respect to the APCC and NAPFD metrics.

/*** Dave's comment [9]: can you clarify/rephrase the next sentence? ***/
There have been some studies of the application of information on incremental interaction coverage. As illustrated in Section 5, for example, ICAFC [3, 5] has recently been proposed to generate incremental adaptive covering arrays based on incremental interaction coverage. Although both their and our studies share the same goal – identifying failures caused by a small number of parameters, as early as possible – there are some fundamental differences between them. Firstly, ICAFC aims mainly at constructing each covering array schedule from the others; /*** Dave's comment [10]: what do you mean by "others"? ***/ IICBP, on the other hand, aims to prioritize combinatorial test cases. Secondly, IICBP belongs to the category of pure prioritization, whereas ICAFC is a re-generation prioritization strategy. Thirdly,

IICBP begins at strength $t = 1$ for ordering test cases, while ICAFC starts at a low strength t, which is not necessarily 1 (usually $t = 2$ [3, 5]). Even if ICAFC starts at $t = 1$, its generated covering arrays are only partially prioritized. For example, suppose that an ICAFC-generated covering array T includes t independent parts $A_1, A_2, \cdots, A_t$ having the same meaning as in Fig. 1, T is a prioritized combinatorial test suite from the perspective of strength (that is, $A_1 \rightarrow A_2 \rightarrow \cdots, \rightarrow A_t$), however, the order of test cases in each subset $A_i (i = 1, 2, \cdots, t)$ is not considered. Finally, ICAFC performs better than traditional methods of constructing covering arrays when multiple covering arrays must be used, the reason being that it can reduce duplication of testing, which means that when a single covering array is used, covering arrays generated by ICAFC may not be comparable in size to those generated by traditional methods [5]. However, IICBP can use good covering arrays with smaller sizes generated by some effective algorithms.

Similar to the ICBP technique, our technique is not limited to conventional software. For example, event-driven software is a widely used category of software that takes sequences of events as input, alters state, and outputs new event sequences [11, 12]. Further studies should be focused on applying our strategy of prioritizing test cases in different software for which information about interaction coverage is available. Furthermore, some factors, such as test case cost and weight, were not considered in guiding test case prioritization in this paper. In future, it will be desirable to apply these factors to IICBP for prioritizing combinatorial test cases. In addition, the APCC metric is a well-known effectiveness measure of the rate of value combinations covered by a test sequence, however, it can only assess a given test sequence at a single strength. Given a combinatorial test sequence T of strength t, the $\text{APCC}_\tau (1 \leq \tau \leq t)$ metric value of T gives the rate of value combinations covered by T at strength τ. In other words, the rate of value combinations of T at strength $\tau' (1 \leq \tau' \leq t, \tau' \neq \tau)$ is neglected. It would be useful, but challenging, to develop a new metric to evaluate the rate of value combinations covered by a test sequence by comprehensively taking into consideration all strengths from 1 to t.

Appendix A. Calculation of μ Illustrated in Section 3.2

The question is formalized as follows. Given an integer variable l, three constant parameters $a (a > 1)$, $k (k > 1)$, and $t (1 \leq t \leq k)$, and a function $f(l) = C_k^l \times \log(a^l)$ where $1 \leq l \leq t$, find an integer $\mu \in [1, t]$ such that $f(\mu) = \max\limits_{1 \leq l \leq t} f(l)$. Obviously,

$$f(l) = C_k^l \times \log(a^l) = C_k^l \times l \times \log(a) = k! \times \log(a) \times \frac{1}{(l-1)! \times (k-l)!}. \quad \text{(A.1)}$$

Since $k! \times \log(a)$ is a constant, the problem converts to finding the minimum of $(l-1)! \times (k-l)! (1 \leq l \leq t)$ (denoted as $g(l)$), that is, finding an integer $\mu \in [1, t]$ such that $g(\mu) = \min\limits_{1 \leq l \leq t} g(l)$.

We first analyze the minimum of $g(l)$ when $l \in [1, k]$. As we know, since l is a discrete variable, that is, $l = 1, 2, \cdots, k$, the minimal value of $g(l)$ certainly exists. On the other hand, $g(1) = g(k) = (k-1)! > g(2) = g(k-1) = (k-2)!$, therefore, $\mu \in [2, k-1]$. Suppose that when $l = \mu$, $g(\mu)$ is the minimum value of $g(l)$, so two inequalities can be easily obtained as follows:

$$
\begin{aligned}
& \begin{cases} g(\mu) \leq g(\mu-1) \\ g(\mu) \leq g(\mu+1) \end{cases} \\
\Rightarrow\ & \begin{cases} (\mu-1)! \times (k-\mu)! \leq (\mu-2)! \times (k-\mu+1)! \\ (\mu-1)! \times (k-\mu)! \leq \mu! \times (k-\mu-1)! \end{cases} \\
\Rightarrow\ & \begin{cases} (\mu-2)! \times (k-\mu)! \times (2\mu-k-2) \leq 0 \\ (\mu-1)! \times (k-\mu-1)! \times (k-2\mu) \leq 0 \end{cases} \\
\Rightarrow\ & \begin{cases} 2\mu-k-2 \leq 0 \\ k-2\mu \leq 0 \end{cases} \quad \begin{pmatrix} \text{because } (\mu-2)! > 0,\ (k-\mu)! > 0, \\ (\mu-1)! > 0, \text{ and } (k-\mu-1)! > 0 \end{pmatrix} \\
\Rightarrow\ & \frac{k}{2} \leq \mu \leq \frac{k}{2} + 1.
\end{aligned}
$$

Intuitively speaking, when k is an even number, μ is equal to $\frac{k}{2}$ or $\frac{k}{2}+1$ to achieve the minimum of $g(h)$ because $g(\frac{k}{2}) = g(\frac{k}{2}+1) = (\frac{k}{2})! \times (\frac{k}{2}-1)!$; when k is an odd number, μ equals $\frac{k+1}{2}$ as it is a unique integer among $[\frac{k}{2}, \frac{k}{2}+1]$. Overall, for any k, $\mu = \lceil \frac{k}{2} \rceil$ such that $g(\mu) = \min\limits_{1 \leq l \leq k} g(l)$.

As a result, if $\lceil \frac{k}{2} \rceil \leq t \leq k$, $\mu = \lceil \frac{k}{2} \rceil$ such that $g(\mu) = \min\limits_{1 \leq l \leq t} g(l)$.

Next, we investigate the value of μ in the case of $1 \leq t < \lceil \frac{k}{2} \rceil$. Suppose two arbitrary integers m and n, such that $1 \leq m < n \leq t < \lceil \frac{k}{2} \rceil$, we can obtain:

$$
\begin{aligned}
g(m) - g(n) &= (m-1)! \times (k-m)! - (n-1)! \times (k-n)! \\
&= (m-1)! \times (k-n)! \times \Big(\prod_{i=m}^{n-1} (k-i) - \prod_{j=1}^{n-m} (n-j)\Big) \\
&= (m-1)! \times (k-n)! \times \Big(\prod_{i=k-n+1}^{k-m} i - \prod_{j=m}^{n-1} j\Big). \qquad \text{(A.2)}
\end{aligned}
$$

Due to $1 \leq m < n \leq t < \lceil \frac{k}{2} \rceil$, $m+n < 2 \times \lceil \frac{k}{2} \rceil$. If k is an even number, $2 \times \lceil \frac{k}{2} \rceil = k$; if k is an odd number, $2 \times \lceil \frac{k}{2} \rceil = k+1$. On the whole, $m+n < k+1$, that is, $k-n+1 > m$. Therefore, $\prod_{i=k-n+1}^{k-m} i > \prod_{j=m}^{n-1} j$. Thus, $g(m) > g(n)$. Consequently, $g(1) > g(2) > \cdots > g(t-1) > g(t)$. In other words, when $\mu = t$, $g(\mu) = \min\limits_{1 \leq l \leq t} g(l)$ for the case of $1 \leq t < \lceil \frac{k}{2} \rceil$.

As discussed above, we can conclude that if $1 \leq t < \lceil \frac{k}{2} \rceil$, $\mu = t$; if $\lceil \frac{k}{2} \rceil \leq t \leq k$, $\mu = \lceil \frac{k}{2} \rceil$, such that $g(\mu) = \min\limits_{1 \leq l \leq t} g(l)$, so that $f(\mu) = \max\limits_{1 \leq l \leq t} f(l)$.

Acknowledgements

/*** Dave's comment [11]: should we mention Dr. Wang? ***/ We would like to thank D. R. Kuhn for providing us the tool ACTS, and C. M. Lott for sending us the failure reports of the subject programs. This work is in part supported by the National Natural Science Foundation of China (Grant No. 61103053, and 61202110), and the Australian Research Council (Grant No. ARC LP100200208).

/*** Dave's comment [12]: the references seem a little strange: were they generated from a BibTeX file? ***/

References

[1] D. R. Kuhn, D. R. Wallace and A. M. Gallo, Software fault interactions and implications for software testing, *IEEE Transaction on Software Engineering* **30**(6) (2004) 418–421.

[2] C. Yilmaz, M. B. Cohen and A. A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, *IEEE Transactions on Software Engineering* **32**(1) (2006) 20–34.

[3] S. Fouché, M. B. Cohen and A. A. Porter, Towards incremental adaptive covering arrays, in *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC-FSE Companion'07*), Dubrovnik, Croatia, 2007, pp. 557–560.

[4] X. Qu, M. B. Cohen and G. Rothermel, Configuration-aware regression testing: An empirical study of sampling and prioritization, in *Proceedings of the ACM International Symposium on Software Testing and Analysis* (*ISSTA'08*), Seattle, Washington, USA, 2008, pp. 75–85.

[5] S. Fouché, M. B. Cohen and A. A. Porter, Incremental covering array failure characterization in large configuration spaces, in *Proceedings of the ACM International Symposium on Software Testing and Analysis* (*ISSTA'09*), Chicago, Illinois, USA, 2009, pp. 177–187.

[6] D. R. Kuhn and M. J. Reilly, An investigation of the applicability of design of experiments to software testing, in *Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop* (*SEW-27'02*), Greenbelt, Maryland, 2002, pp. 91–95.

[7] C. Nie and H. Leung, A survey of combinatorial testing, *ACM Computing Surveys* **43**(2) (2011) 11.1–11.29.

[8] G. Rothermel, R. H. Untch, C. Y. Chu and M. J. Harrold, Prioritizing test cases for regression testing, *IEEE Transactions on Software Engineering* **27**(10) (2001) 929–948.

[9] R. C. Bryce and C. J. Colbourn, Test prioritization for pairwise interaction coverage, in *Proceedings of the 1st International Workshop on Advances in Model-Based Software Testing* (*A-MOST'05*), Louis, Missouri, USA, 2005, pp. 1–7.

[10] R. C. Bryce and C. J. Colbourn, Prioritized interaction testing for pair-wise coverage with seeding and constraints, *Information and Software Technology* **48**(10) (2006) 960–970.

[11] R. C. Bryce and A. M. Memon, Test suite prioritization by interaction coverage, in *Proceedings of the Workshop on Domain-Specific Approaches to Software Test Automation* (*DoSTA'07*), Dubrovnik, Croatia, 2007, pp. 1–7.

[12] R. C. Bryce, S. Sampath and A. M. Memon, Developing a single model and test prioritization strategies for event-driven software, *IEEE Transaction on Software Engineering* **37**(1) (2011) 48–64.

[13] X. Qu, M. B. Cohen and K. M. Woolf, Combinatorial interaction regression testing: A study of test case generation and prioritization, in *Proceedings of the 23rd IEEE International Conference on Software Maintenance* (*ICSM'07*), Paris, France, 2007, pp. 255–264.

[14] X. Chen, Q. Gu, X. Zhang and D. Chen, Building prioritized pairwise interaction test suites with ant colony, in *Proceedings of the 9th Conference on Quality Software* (*QSIC'09*), Jeju, Korea, 2009, pp. 347–352.

[15] Z. Wang, L. Chen, B. Xu and Y. Huang, Cost-cognizant combinatorial test case prioritization, *International Journal of Software Engineering and Knowledge Engineering* **21**(6) (2011) 829–854.

[16] R. Huang, X. Xie, T. Y. Chen and Y. Lu, Adaptive random test case generation for combinatorial testing, in *Proceedings of the 36th Annual International Computer Software and Applications Conference* (*COMPSAC'12*), Izmir, Turkey, 2012, pp. 52–61.

[17] G. Seroussi and N. H. Bshouty, Vector sets for exhaustive testing of logic circuits, *IEEE Transaction on Software Engineering* **34**(3) (1988) 513–522.

[18] S. Elbaum, A. G. Malishevsky and G. Rothmermel, Test case prioritization: A family of empirical studies, *IEEE Transaction on Software Engineering* **28**(2) (2002) 159–182.

[19] S. A. Sahaaya Arul Mary and R. Krishnamoorthi, Time-aware and weighted fault severity based metrics for test case prioritization, *International Journal of Software Engineering and Knowledge Engineering* **21**(1) (2011) 129–142.

[20] Z. Li, M. Harman and R. M. Hierons, Search algorithms for regression test case prioritization, *IEEE Transaction on Software Engineering* **33**(4) (2007) 225–237.

[21] H. Yoon and B. Choi, A test case prioritization based on degree of risk exposure and its empirical study, *International Journal of Software Engineering and Knowledge Engineering* **21**(2) (2011) 191–209.

[22] W. E. Wong, J. R. Horgan, S. London and H. Agrawal, A study of effective regression testing in practice, in *Proceedings of the 8th IEEE International Symposium on Software Reliability Engineering* (*ISSRE'97*), Albuquerque, New Mexico, USA, 1997, pp. 264–274.

[23] S. Elbaum, A. G. Malishevsky and G. Rothmermel, Incorporating varying test costs and fault severities into test case prioritization, in *Proceedings of the 25th International Conference on Software Engineering* (*ICSE'03*), Portland, Oregon, USA, 2003, pp. 329–338.

[24] Y-C. Huang, K-L. Peng and C-Y. Huang, A history-based cost-cognizant test case prioritization technique in regression testing, *Journal of Systems and Software* **85**(3) (2012) 626–637.

[25] K. C. Tai and Y. Lei, A test generation strategy for pairwise testing, *IEEE Transaction on Software Engineering* **28**(1) (2002) 109–111.

[26] Y. Lei, R. Kacker, D. R. Kuhn and V. Okun, IPOG/IPOD: Efficient test generation for multi-way testing, *Software Testing, Verification and Reliability* **18**(3) (2007) 125–148.

[27] J. Czerwonka, Pairwise testing in real world: Practical extensions to test case generators, in *Proceedings of the 24th Pacific Northwest Software Quality Conference* (*PNSQC'06*), Portland, Oregon, USA, 2006, pp. 419–430.

[28] R. C. Bryce, C. J. Colbourn and M. B. Cohen, A framework of greedy methods of constructing interaction test suites, in *Proceedings of the 27th International Conference on Software Engineering* (*ICSE'05*), St. Louis, Missouri, USA, 2005, pp. 146–155.

[29] C. M. Lott and H. D. Rombach, Repeatable software engineering experiments for

comparing defect-detection techniques, *Empirical Software Engineering* **1**(3) (1996) 241–277.

[30] M. Grindal, B. Lindström, J. Offutt and S. F. Andler, An evaluation of combination strategies for test case selection, *Empirical Software Engineering* **11**(4) (2006) 583–611.

[31] Z. Zhang and J. Zhang, Characterizing failure-causing parameter interactions by adaptive testing, in *Proceedings of the ACM International Symposium on Software Testing and Analysis* (*ISSTA'11*), Toronto, Canada, 2011, pp. 331–341.

[32] L. S. G. Ghandehari, Y. Lei, T. Xie, D. R. Kuhn and R. Kacker, Identifiying failure-inducing combinations in a combinatorial test set, in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation* (*ICST'12*), Montreal, Canada, 2012, pp. 370–379.

[33] Z. Wang, Test case generation and prioritization for combinatorial testing, Ph.D. Thesis, School of Computer Science and Engineering, Southeast University, China, 2009.