國立交通大學

資訊科學與工程研究所

碩士論文

一個有效率在成對測試上產生測試資料的演算法

An Efficient Algorithm to Case Generation for Pairwise Testing

研究生：林宗蔚

指導教授：王豐堅 教授

中華民國九十六年八月

一個有效率在成對測試上產生測試資料的演算法

An Efficient Algorithm to Case Generation for Pairwise Testing

研究生：林宗蔚　　Student：Chung-Way Lin

指導教授：王豐堅　　Advisor：Feng-Jian Wang

國立交通大學
資訊科學與工程研究所
碩士論文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年八月

一個有效率在成對測試上產生測試資料的演算法

研究生: 林宗蔚　　　　　　指導教授: 王豐堅 博士

國立交通大學

資訊科學與工程研究所

新竹市大學路 1001 號

摘要

軟體的測試是耗費金錢與時間，而且常常因爲預算有限而受到限制。當系統需要測試不同的 parameters，且每個 parameters 有不同的參數，測試所有可能的組合需要花費大量的時間和金錢。Pairwise testing 就是對任意兩個 parameters，其所有參數的組合都必須再 test case set 中至少出現一次。本篇論文提出了一個 testing generation strategy 並與其他策略比較實驗的結果。除此之外,這篇論文也同時提出了一個演算法，在某種前提條件下，此演算法能夠快速的擴展 test cases。

Keywords:軟體測試, pairwise testing, testing generation, test case set.

An Efficient Algorithm to Case Generation for Pairwise Testing

Student: Chung-Way Lin Advisor: Dr. Feng-Jian Wang
Institute of Computer Science and Engineering
Nation Chiao Tung University
1001 Ta Hsueh Road, Hsinchu, Taiwan, ROC

Abstract

Software testing is expensive, time consuming and is often restricted by budgets. Given different input parameters with distinct values, exhaustive testing which tests all possible combination needs lots of money and time. Pairwise testing requires that, for each pair of input parameters of a system, every combination of valid values of any two parameters must be covered at least once in a test case set. In this paper, we present a testing generation strategy for pairwise testing. The algorithms are constructed by improving the testing strategy *IPO* and the simulation is then made for comparison with *IPO*. Besides, a study for quick extension of test cases is presented. Under some constraints, the existent test cases can be extended quickly by using the extension method.

Keywords: software testing, pairwise testing, testing generation, test case set.

誌謝

本篇論文的完成，首先要感謝我的指導教授王豐堅博士兩年來不斷的指導與鼓勵，讓我在軟體工程的測試技術上，得到很多豐富的知識與經驗。另外，也非常感謝我的畢業口試評審委員梅興博士與留忠賢博士，提供許多寶貴的意見，補足我論文裡不足的部分。

其次，我要感謝實驗室的學長姐們，有博士班懷中學長的指導與照顧，讓我學到許多做研究的技巧，得以順利完成論文。

最後，我要感謝我的家人，由於你們的支持，讓我能心無旁騖地讀書，專心做研究。由衷地感謝你們大家一路下來陪著我走過這段研究生歲月。

Table of Contents

List of Figures and Tables

Chapter 1. Introduction

Software testing becomes more and more important due to complexity and increment size of software systems. However, software testing is an expensive and time consuming process often restricted by budgets. Given different input parameters and each parameter with different values, exhaustive testing to all possible combinations costs lots of time and money. To balance the budget and efficiency, pairwise testing is frequently adopted for different types in software testing [3][4].

Given numbers of input parameters with different values to the system, pairwise testing tests every combination of valid values of any two parameters in a test case set. For example, a function with three three-valued parameters *A*, *B*, *C*, is awaited to be tested

	Parameter *A*	Parameter *B*	Parameter *C*
Value 1	a_1	b_1	c_1
Value 2	a_2	b_2	c_2
Value 3	a_3	b_3	c_3

It needs 27 (3*3*3) test cases to exhaust the testing. However, with pairwise testing technique, only nine test cases are required to cover all of the pairwise combinations, as shown in Table 1.

Test cases	Parameter A	Parameter B	Parameter C
1	a_1	b_1	c_1
2	a_1	b_2	c_2

3	a_1	b_3	c_3
4	a_2	b_1	c_2
5	a_2	b_2	c_3
6	a_2	b_3	c_1
7	a_3	b_1	c_3
8	a_3	b_2	c_1
9	a_3	b_3	c_2

Table 1. Test cases for three three-valued parameters

Many test generation strategies for pairwise testing are published. The strategy proposed in [3][13][14] starts with an empty set and test cases are added one by one for testing. A number of candidate test cases are produced by a greedy algorithm, and the one covering the most uncovered pairs is chosen. Adopting greedy algorithm makes the strategy time and space consuming.

Another pairwise testing strategy is called “Orthogonal Latin squares” [7][11]. If all parameters have the same number of values, Orthogonal Latin squares can be used to generate optimal test case set.

In [2], an approach called *IPO* to generate pairwise test cases is raised. The *IPO* strategy consists of three parts, a set storing all uncovered pairs, *IPO_H* (Horizontal), and *IPO_V* (Vertical). *IPO_H* extends original test cases when adding parameters, and *IPO_V* increases the number of test cases.

However, *IPO* may generate unnecessary test cases during execution. In this thesis, we propose two test generation strategies, called in-values-order (*IVO*). During the experiment, a critical disadvantage of *IVO* is found and a modified *IVO* is presented to overcome the disadvantage of *IVO*. The reminder of the paper is organized as follows. Section 2 describes an overview of testing and some papers about pairwise testing published. Section 3 shows a new strategy for pairwise testing called *IVO* and a modified *IVO*. Section 4 describes the duplicate algorithm. Conclusion and future work is in section 5.

Chapter 2 Background

Section 2.1 Overview Of Testing Techniques

In a program, there might exist several kinds of errors, e.g., control flow errors, data-declaration errors, and, … etc. Testing is the process of executing a program with the intent of finding errors. [5]

A strategy for software testing may be viewed as the set of testing with the spiral shown in Figure 1.

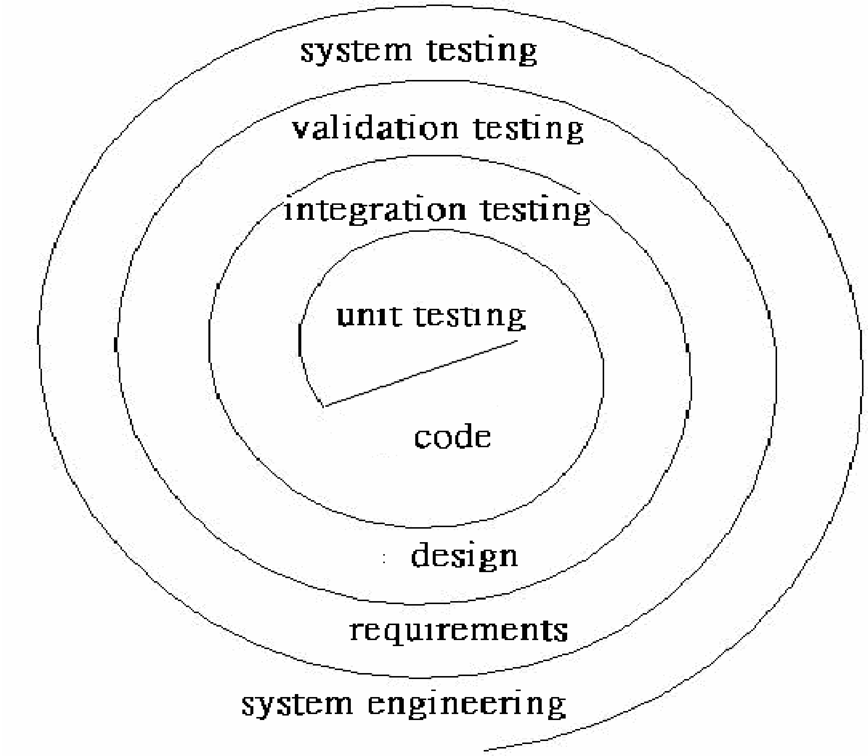


Figure 1 Testing strategy [6]

Testing within software engineering is implemented sequentially in three steps, which are unit test, integration test and high-order tests, as shown in Figure 2.

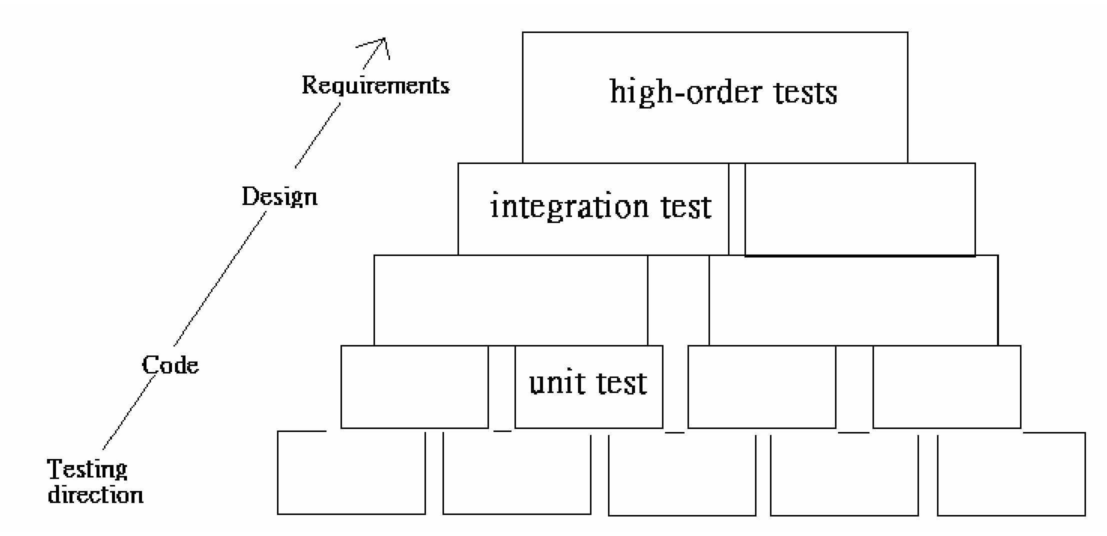


Figure 2 Software testing steps [6]

Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration. The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. [6]

Resources for testing such as resource time, budget, and computing time are limited. To achieve complete testing for a program is impossible. The key issue of testing becomes "To find the subset among all possible test cases has the highest probability of detecting the most errors" [6]. The most important part in program testing is to design effective test cases.

Another issue for testing is the test case prioritization [8][9][10][12]. Test case prioritization is to permute the execution of test cases according to some criterion. The benefit for test case prioritization is to help in early detection of faults during regression testing.

Given different input parameters with distinct values, exhaustive testing needs lots of money and time. Pairwise testing is effective for different types in software testing. Pairwise testing requires that the input parameters have their own values to the system, and every combination of valid values of any two parameters must be covered at least once in a test case set.

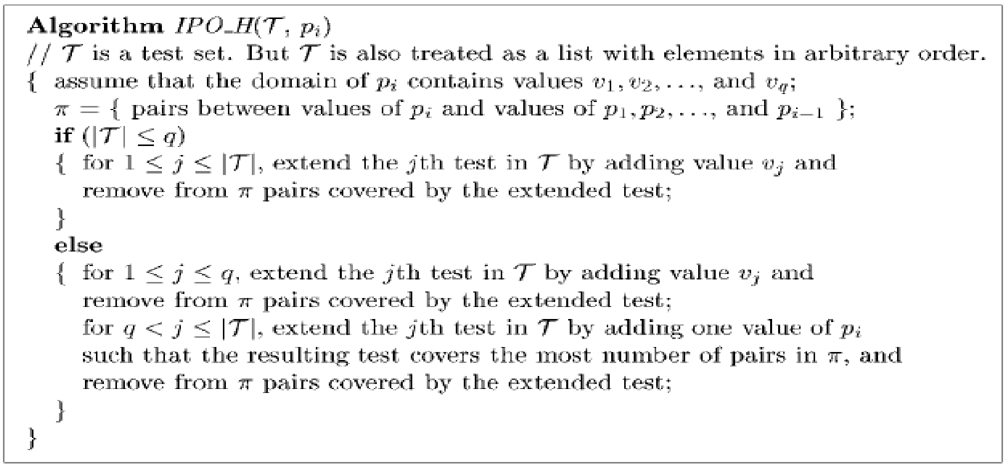
The problem of generating a minimum set of pairwise test cases is proved NP-complete [1]. Finding strategies which generate pairwise test set is necessary. Many test generation strategies for pairwise testing are published [2][3][7]. The strategy proposed in [3][13][14] starts with an empty set and test cases are added one by one for testing. A number of candidate test cases are produced by a greedy algorithm, and the one covering the most uncovered pairs is chosen. Adopting greedy algorithm makes the strategy time and space consuming.

Section 2.2 The *IPO* Strategy

In [2], the approach called *IPO* to generate pairwise test cases is raised. In order to explain the *IPO* clearly, some terms are defined as follows. Parameters are denoted with capital letters such as *A*, *B,* … , etc. Values for each parameter are denoted as corresponding lowercase letters with foot marks. For examples, values of parameter *A* are a_1, a_2, …, etc. A test case denoted as [a_1, b_1, …] is a combination of values for each parameter. A pair is denoted as (a, b) which is the combination of values for two different parameters. Uncovered pairs are the pairs which are not found in existent test cases. In a test case, * is used to represent any possible values of some designated parameters.

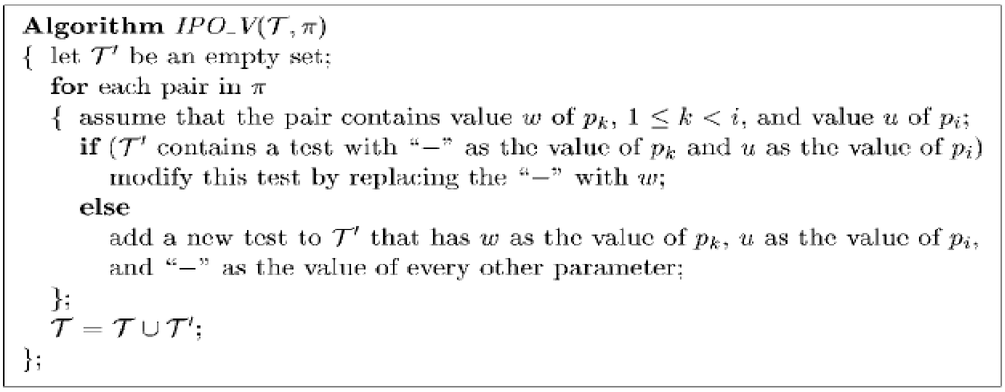
The *IPO* strategy consists of three parts, a set π stores all uncovered pairs, *IPO_H* (Horizontal), and *IPO_V* (Vertical) shown in Figure 3 and Figure 4. *IPO_H* extends original test cases when adding parameters, and *IPO_V* increases the number of test cases.

In *IPO_H*, the initial pairwise test set is generated for the first two parameters, and each possible value of the new parameter is added to each existent test cases one by one as extension. Therefore, there're unextended test cases if number of possible values is less than number of original test cases. For each unextended test case, the value covering the most uncovered pairs with it is added to accomplish the extension. If there are uncovered pairs in *Π*, these pairs are merged to generate test cases until all pairs are covered, and these new generated test cases are added to the test case set.



```
Algorithm IPO_H($\mathcal{T}$, $p_i$)
// $\mathcal{T}$ is a test set. But $\mathcal{T}$ is also treated as a list with elements in arbitrary order.
{ assume that the domain of $p_i$ contains values $v_1, v_2, \ldots,$ and $v_q$;
  $\pi$ = { pairs between values of $p_i$ and values of $p_1, p_2, \ldots,$ and $p_{i-1}$ };
  if ($|\mathcal{T}| \leq q$)
  { for $1 \leq j \leq |\mathcal{T}|$, extend the $j$th test in $\mathcal{T}$ by adding value $v_j$ and
    remove from $\pi$ pairs covered by the extended test;
  }
  else
  { for $1 \leq j \leq q$, extend the $j$th test in $\mathcal{T}$ by adding value $v_j$ and
    remove from $\pi$ pairs covered by the extended test;
    for $q < j \leq |\mathcal{T}|$, extend the $j$th test in $\mathcal{T}$ by adding one value of $p_i$
    such that the resulting test covers the most number of pairs in $\pi$, and
    remove from $\pi$ pairs covered by the extended test;
  }
}
```

Figure 3 Algorithm *IPO_H* [2]



```
Algorithm IPO_V($\mathcal{T}$, $\pi$)
{ let $\mathcal{T}'$ be an empty set;
  for each pair in $\pi$
  { assume that the pair contains value $w$ of $p_k$, $1 \leq k < i$, and value $u$ of $p_i$;
    if ($\mathcal{T}'$ contains a test with "−" as the value of $p_k$ and $u$ as the value of $p_i$)
      modify this test by replacing the "−" with $w$;
    else
      add a new test to $\mathcal{T}'$ that has $w$ as the value of $p_k$, $u$ as the value of $p_i$,
      and "−" as the value of every other parameter;
  };
  $\mathcal{T} = \mathcal{T} \cup \mathcal{T}'$;
};
```

Figure 4 Algorithm *IPO_V* [2]

	Parameter A	Parameter B	Parameter C	Parameter D
Values 1	a_1	b_1	c_1	d_1
Values 2	a_2	b_2	c_2	d_2
Values 3			c_3	d_3

Table 2 Example for *IPO* strategy

Take Table 2 as an example. There are four parameters *A*, *B*, *C*, and *D*. *A* and *B* are two-valued; *C* and *D* are three-valued. Initially, *IPO_H* works for the first two parameters *A* and *B*; four test cases [a_1, b_1], [a_1, b_2], [a_2, b_1], [a_2, b_2] are generated and Π is empty. Then parameter *C* is added to generate twelve uncovered pairs (a_1, c_1), (a_1, c_2), (a_1, c_3), (a_2, c_1), (a_2, c_2), (a_2, c_3), (b_1, c_1), (b_1, c_2), (b_1, c_3), (b_2, c_1), (b_2, c_2) and (b_2, c_3). Because there are three values in parameter *C*, c_1, c_2, c_3 are added to [a_1, b_1], [a_1, b_2], [a_2, b_1] respectively, and the extended test cases are [a_1, b_1, c_1], [a_1, b_2, c_2], [a_2, b_1, c_3]. Then these pairs (a_1, c_1), (b_1, c_1), (a_1, c_2), (b_2, c_2), (a_2, c_3), (b_1, c_3) are removed from Π. To extend [a_2, b_2], the three possible extensions are shown as follows.

- [a_2, b_2, c_1] which covers two uncovered pairs (a_2, c_1), (b_2, c_1).
- [a_2, b_2, c_2] which covers one uncovered pair (a_2, c_2).
- [a_2, b_2, c_3] which covers one uncovered pair (b_2, c_3).

Since [a_2, b_2, c_1] covers the most uncovered pairs. [a_2, b_2, c_1] is chosen as extension [a_2, b_2], and (a_2, c_1) and (b_2, c_1) are removed from Π.

After *IPO_H*, there are still four pairs (a_1, c_3), (a_2, c_2), (b_1, c_2) and (b_2, c_3) in Π. The test case [a_1, *, c_3] is generated to cover the pair (a_1, c_3). Because [a_1, *, c_3] and (a_2, c_2) are not the same values, (a_2, c_2) can't be merged in to [a_1, *, c_3] and the test case [a_2,*, c_2] can be generated to cover (a_2, c_2). The position of parameter *B* in [a_2,*, c_2] is *, and the value of parameter *C* in (b_1, c_2) is c_2, so the * is changed for b_1. The test case [a_2,*, c_2] is changed to [a_2, b_1, c_2]. For (b_2, c_3), the execution process is the same with above, and the test case [a_1, *, c_3] is changed to [a_1, b_2, c_3]. The two new test cases are added to the test case set, and there are six test cases [a_1, b_1, c_1], [a_1, b_2, c_2], [a_2, b_1, c_3], [a_2, b_2, c_1], [a_2, b_1, c_2], [a_1, b_2, c_3] in the test case set, and no pairs are in Π.

Then parameter *D* is added, the execution process is the same with the above *IPO_H*. Finally in *IPO_H*, the six extended test cases are [a_1, b_1, c_1, d_1], [a_1, b_2, c_2, d_2], [a_2, b_1, c_3, d_3], [a_2, b_2, c_1, d_1], [a_1, b_2, c_3, d_3] and [a_2, b_1, c_2, d_2], and there are still six pairs (c_1, d_2), (c_1, d_3), (c_2, d_1), (c_2, d_3), (c_3, d_1), (c_3, d_2) in Π. Because these pairs can not merged together, the six test cases [*,*, c_1, d_2], [*,*, c_1, d_3], [*,*, c_2, d_1], [*,*, c_2, d_3], [*,*, c_3, d_1], [*,*, c_3, d_2] are generated and * can be assigned any values of the designated parameter. Finally, the twelve test cases [a_1, b_1, c_1, d_1], [a_1, b_2, c_2, d_2], [a_2, b_1, c_3, d_3], [a_2, b_2, c_1, d_1], [a_1, b_2, c_3, d_3], [a_2, b_1, c_2, d_2], [*,*, c_1, d_2], [*,*, c_1, d_3], [*,*, c_2, d_1], [*,*, c_2, d_3], [*,*, c_3, d_1]and [*,*, c_3, d_2] are generated by the *IPO*, and * can be assigned any values of corresponding parameters.

But in *IPO*, because each * must be assigned a value of corresponding parameter after each *IPO_V*, assignment of values to *'s may result that more test case are required to cover all pairs.

For example, there are five two-valued parameters $A = \{a_1, a_2\}$, $B = \{b_1, b_2\}$, $C = \{c_1, c_2\}$, $D = \{d_1, d_2\}$, $E = \{e_1, e_2\}$. For the first two parameters *A* and *B*, four test cases are generated [a_1, b_1], [a_1, b_2], [a_2, b_1], [a_2, b_2]. After execution *IPO_H* for adding parameter *C*, the extended test cases are [a_1, b_1, c_1], [a_1, b_2, c_2], [a_2, b_1, c_2], [a_2, b_2, c_1], and no uncovered pairs. Then parameter *D* is added, after execution *IPO_H*, the four extended test cases are [a_1, b_1, c_1,

d_1], [a_1, b_2, c_2, d_2], [a_2, b_1, c_2, d_1], [a_2, b_2, c_1, d_2], and there are two uncovered pairs (b_1, d_2) and (b_2, d_1). Because (b_1, d_2) and (b_2, d_1) can not be merged together, the two test cases [*, b_1, *, d_2], [*, b_2, *, d_1] are generated in *IPO_V.* Different assignment of values to *'s in [*, b_2, *, d_1] leads to different results when parameter *E* is added:

- If [*, b_2, *, d_1] is assigned to become [a_2, b_2, c_1, d_1] and [*, b_1, *, d_2] is assigned to becomes [a_1, b_1, c_1, d_2]. Then parameter *E* is added, after *IPO_H*, the six extended test cases are [a_1, b_1, c_1, d_1, e_1], [a_1, b_2, c_2, d_2, e_2], [a_2, b_1, c_2, d_1, e_2], [a_2, b_2, c_1, d_2, e_1], [a_1, b_1, c_1, d_2, e_2] and [a_2, b_2, c_1, d_1, e_1], and there is one uncovered pair (c_2, e_1), therefore, the extra test case [*,*, c_2, *, e_1] is need to cover pair (c_2, e_1).

- If [*, b_2, *, d_1] is assigned to become [a_2, b_2, c_2, d_1] and [*, b_1, *, d_2] is assigned to become [a_1, b_1, c_1, d_2]. Then parameter *E* is added, after *IPO_H*, the six extended test cases are [a_1, b_1, c_1, d_1, e_1], [a_1, b_2, c_2, d_2, e_2], [a_2, b_1, c_2, d_1, e_2], [a_2, b_2, c_1, d_2, e_1], [a_1, b_1, c_1, d_2, e_2] and [a_2, b_2, c_2, d_1, e_1], and no uncovered pairs.

In above example, different assignment of values to *'s in test cases for former processes to cover all pairs lead different number of test cases for five parameters.

Section 2.3 Orthogonal Latin Squares

Another pairwise testing strategy is called "Orthogonal Latin squares" [7][11]. If all parameters have the same number of values, Orthogonal Latin squares can be used to generate optimal test case set. Optimal test case set is the minimum set of test cases which cover all pairs. A Latin square is usually represented by a square matrix as follows:

$$1 \quad 2 \quad 3$$

$$\begin{array}{ccc} 2 & 3 & 1 \\ 3 & 1 & 2 \end{array}$$

Values in the column 1 are the values of parameter 1, so are the values in column 2 and column 3, correspondingly. The Latin square has the property that each value in a column or row is distinct.

Assume that there are two matrixes $[A_{ij}]$ and $[B_{ij}]$, if the combined matrix is $[C_{ij}] = (A_{ij}, B_{ij})$. For C_{ij}, C_{xy}, $i \neq x$ and $j \neq y$, if $C_{ij} \neq C_{xy}$, $[A_{ij}]$ and $[B_{ij}]$ are orthogonal. If there are *k* parameters, the methodology needs *k*-2 orthogonal Latin squares. For example, there are four parameters and three values for each parameter, two Orthogonal Latin squares are required and shown as follows:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{array} \qquad \begin{array}{ccc} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{array}$$

The following matrix is obtained through superimposing the above matrix.

$$\begin{array}{ccc} (1, 1) & (2, 2) & (3, 3) \\ (3, 2) & (1, 3) & (2, 1) \\ (2, 3) & (3, 1) & (1, 2) \end{array}$$

The methodology represents the configuration (2, 1, 3, 2): row 2, column 1, entry (3, 2) and the complete set of test configurations is shown as follows:

Configuration number	Parameter1	Parameter2	Parameter3	Parameter4
1	1	1	1	1

2	1	2	2	2
3	1	3	3	3
4	2	1	3	2
5	2	2	1	3
6	2	3	2	1
7	3	1	2	3
8	3	2	3	1
9	3	3	1	2

Table 3 Test configurations of orthogonal latin squares

For each row in Table 3, each configuration number means a test case, and the numbers from column 2 to column 5 mean the values' combination of each test case. Therefore, for four three-valued parameters $A = \{a_1, a_2, a_3\}$, $B = \{b_1, b_2, b_3\}$, $C = \{c_1, c_2, c_3\}$ and $D = \{d_1, d_2, d_3\}$. According to Table 3, the optimal nine test cases are $[a_1, b_1, c_1, d_1]$, $[a_1, b_2, c_2, d_2]$, $[a_1, b_3, c_3, d_3]$, $[a_2, b_1, c_3, d_2]$, $[a_2, b_2, c_1, d_3]$, $[a_2, b_3, c_2, d_1]$, $[a_3, b_1, c_2, d_3]$, $[a_3, b_2, c_3, d_1]$ and $[a_3, b_3, c_1, d_2]$.

However, Orthogonal Latin squares have the following limitation:

1. Orthogonal arrays might not exist.
2. Every parameter must be with the same number of values.
3. $|N|$ means the number of parameters, and $|V|$ means the number of values for each parameter. Orthogonal Latin squares require that $|N|$ isless than $|V|+1$, and $|V|$ is a power of a prime.
4. For $|V| = 6$, Orthogonal Latin squares do not exist.

Chapter 3 The *IVO* Strategy

To solve the disadvantage of *IPO* (In Parameters Order), in this chapter, a test generation algorithm for pairwise testing called In Values Order (*IVO*) is described. *IVO* is improved from the *IPO* strategy [2] with the execution order of *IPO_H* and *IPO_V*. Chapter 3 is organized as follows. Section 3.1 describes the *IVO* strategy, the simulation results and companion between *IVO* and *IPO*. During the simulation, a critical disadvantage of *IVO* is found, and *modified IVO* (*MIVO*) is proposed in section 3.2. Section 3.3 compares with *MIVO* and *IPO*.

3.1 The *IVO* Strategy

The execution order of *IPO* is different from that of *IVO*. For *IPO*, *IPO_H* and *IPO_V* are sequentially operated for each parameter. In *IVO*, it is assumed that all the input parameters are known and ordered according to the number of values. *IVO* extends the test cases until all parameters are added, then all uncovered pairs are merged to test cases until all pairs are covered. To explain *IVO* clearly, the notations are defined as follows.

- $P = \{P_1, P_2, \dots, P_n\}$ is a set of parameters.
- V_i is the value set of P_i, and $|V_i| \geq |V_{i+1}|$.
- v_{ij} is the jth element of V_i.
- PS is the set of uncovered pairs $PS = \{(a, b) \mid a \in V_i, b \in V_j, i \neq j$, and (a, b) is uncovered pair$\}$.
- A test case $T = \{[a_1, a_2, \dots, a_n] \mid a_i \in V_i\}$.
- $TS = \{T_1, T_2, \dots, T_m\}$ is the set of test cases.

```
Algorithm: In-Value-Order
Begin:
    for the first two parameters, $P_1$ and $P_2$
    $TS := \{[v_1, v_2] \mid v_1 \in V_1, v_2 \in V_2\}$
    if $n = 2$ then stop;
    $\forall$ parameter $P_i$, $i = 3, \ldots, n$;
     {
        $\forall \gamma \in V_1 \cup V_2 \cup \ldots \cup V_{i-1}, \delta \in V_i$
            add $(\gamma, \delta)$ to $PS$
        for $1 \le j \le |V_i|$ that $T_j = [a_1, a_2, \ldots, a_{i-1}]$
          {
            extend $T_j$ with $v_{ij}$ such that $T_j = [a_1, a_2, \ldots, a_{i-1}, v_{ij}]$
            and remove $(a_1, v_{ij}), (a_2, v_{ij}), \ldots, (a_{i-1}, v_{ij})$ from $PS$
          }
        for $|V_i| < j \le |TS|$
         {
            can_count = 0
            $T_j = [a_1, a_2, \ldots, a_{i-1}]$
            for $1 \le k \le |V_i|$
              {
                count = $|\{(a_z, v_{ik}) \mid z = 1,2,\ldots i-1, (a_z, v_{ik}) \in PS\}|$
                if(count > can_count)
                  {
```

```
22                          can_count = count
23                          $c = k$
24                        }
25                    }
26              extend $T_j$ with $v_{ik}$, such that $T_j$=[ $a_1, a_2, \ldots, a_{i-1}, v_{ik}$ ]and remove
27              $(a_1, v_{ik}), (a_2, v_{ik}), \ldots, (a_{i-1}, v_{ik})$ from $PS$
28          }
29      }
30      let $TS$' is a test case set, $TS' = \varnothing$
31        $\forall (\alpha, \beta) \in PS, \alpha \in V_i, \beta \in V_j$
32          {
33              if ( $\exists$ some test cases $T = [a_1, a_2, \ldots, a_n]$, $T \in TS'$ that ($a_i = \alpha, a_j$
34               $= \beta$) or ($a_i = *, a_j = \beta$) or ($a_i = a_j = *$))
35                   set $a_i = \alpha, a_j = \beta$ and remove $(\alpha, \beta)$ from $PS$
36              else
37                   generate a new test case $T = [a_1, a_2, \ldots, a_n]$ in which
38                   $\forall z = 1 \ldots n, z \neq i, j, a_z = *, a_i = \alpha, a_j = \beta$, add $T$ to $TS$'
39                   remove $(\alpha, \beta)$ from $PS$
40              }
41      $TS = TS \cup TS'$
42  end
```

In *IVO* algorithm, test cases are generated by the first two parameters in line 02. When parameter is introduced in line 06, new generated uncovered pairs are added to *PS* in line 07. Because parameters are sorted, |*TS*| must be large than *j* in line 13. Existing test cases are extended until all parameters are added from line 08 to line 28. Finally, the uncovered pairs are merged to test cases from line 30 to line 41.

For example, there are four parameters $P_1 = \{v_{11}, v_{12}, v_{13}\}$, $P_2 = \{v_{21}, v_{22}, v_{23}\}$, $P_3 = \{v_{31}, v_{32}\}$ and $P_4 = \{v_{41}, v_{42}\}$. Initially in *IVO*, for the first two parameters P_1 and P_2, nine test cases $[v_{11}, v_{21}]$, $[v_{11}, v_{22}]$, $[v_{11}, v_{23}]$, $[v_{12}, v_{21}]$, $[v_{12}, v_{22}]$, $[v_{12}, v_{23}]$, $[v_{13}, v_{21}]$, $[v_{13}, v_{22}]$ and $[v_{13}, v_{23}]$ are generated to *TS*. Then parameter P_3 is added, the twelve uncovered pairs (v_{11}, v_{31}), (v_{11}, v_{32}), (v_{12}, v_{31}), (v_{12}, v_{32}), (v_{13}, v_{31}), (v_{13}, v_{32}), (v_{21}, v_{31}), (v_{21}, v_{32}), (v_{22}, v_{31}), (v_{22}, v_{32}), (v_{23}, v_{31}) and (v_{23}, v_{32}) are added to *PS*. Because there are two values in parameter P_3, v_{31} andv_{32} are added to $[v_{11}, v_{21}]$ and $[v_{11}, v_{22}]$ respectively. The extended test cases are $[v_{11}, v_{21}, v_{31}]$, $[v_{11}, v_{22}, v_{32}]$, and the pairs (v_{11}, v_{31}), (v_{21}, v_{31}), (v_{11}, v_{32}) and (v_{22}, v_{32}) are removed from *PS*. To extend $[v_{11}, v_{23}]$, two possible extensions, are shown as follows, are introduced.

- $[v_{11}, v_{23}, v_{31}]$ which covers uncovered pair (v_{23}, v_{31}) in *PS*
- $[v_{11}, v_{23}, v_{32}]$ which covers uncovered pair (v_{23}, v_{32}) in *PS*

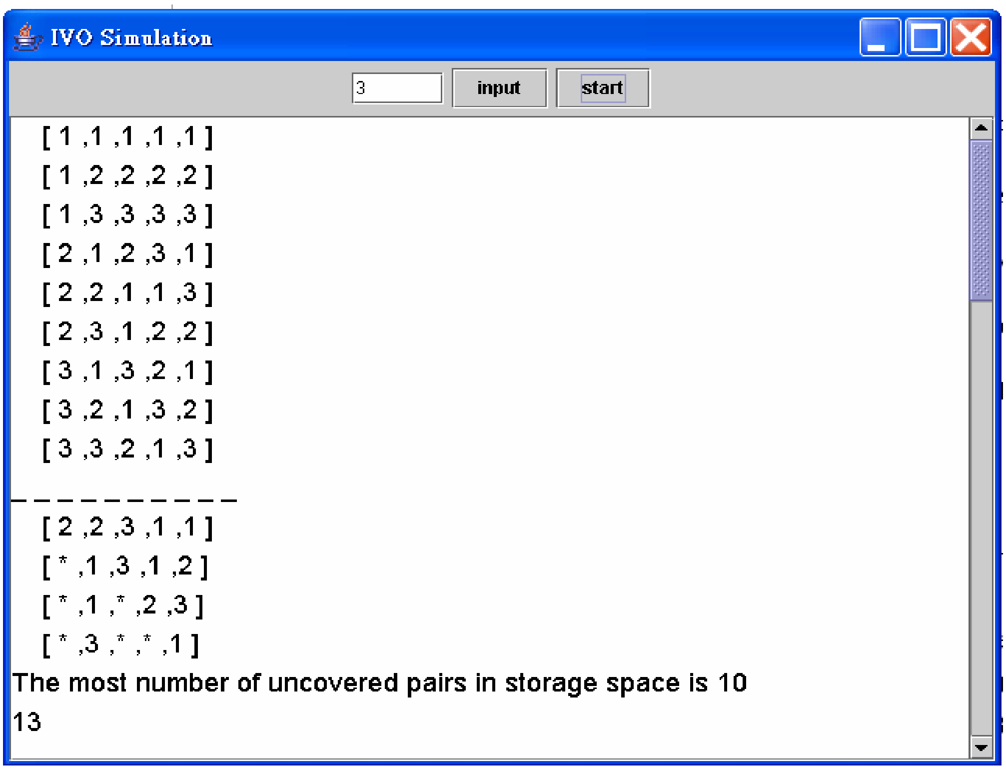
Both test cases cover only one uncovered pair, $[v_{11}, v_{23}, v_{31}]$ is chosen, and (v_{23}, v_{31}) is removed from *PS*. For the rest test cases, the extension process is the same as above. Finally, the rest six extended test case are $[v_{12}, v_{21}, v_{32}]$, $[v_{12}, v_{22}, v_{31}]$, $[v_{12}, v_{23}, v_{32}]$, $[v_{13}, v_{21}, v_{31}]$, $[v_{13}, v_{22}, v_{32}]$ and $[v_{13}, v_{23}, v_{31}]$ and no pairs are left in *PS*.

When parameter P_4 is added, the execution process is the same as above. Nine test cases, $[v_{11}, v_{21}, v_{31}, v_{41}]$, $[v_{11}, v_{22}, v_{32}, v_{42}]$, $[v_{11}, v_{23}, v_{31}, v_{42}]$, $[v_{12}, v_{21}, v_{32}, v_{41}]$, $[v_{12}, v_{22}, v_{31}, v_{41}]$, $[v_{12}, v_{3}, v_{32}, v_{41}]$, $[v_{13}, v_{21}, v_{31}, v_{42}]$, $[v_{13}, v_{22}, v_{32}, v_{41}]$ and $[v_{13}, v_{23}, v_{31}, v_{41}]$ are extended. However, there is still one pair (v_{12}, v_{42}) left in *PS*, and test case $[v_{12}, *, *, v_{42}]$ is generated to *TS* in merging process, and * can be assigned any value of corresponding parameter. The ten test cases are generated by *IVO* for pairwise testing.

We have implemented a program for simulation of *IVO* and *IPO*. The program was written in j2sdk 1.4.2_06 and the computer hardware and software are listed as follows: operation system is windows XP, the cpu is AMD Athon(tm)64 processor(1.81GHz) and the memory is DDR 512MB.

In the program, the input is a set containing numbers of values for each parameter, and the output is all the test cases. Each number in test cases represents the value of the corresponding parameter, i.e., according to the input order, if the first parameter is $P_1 = \{\nu_{11,} \nu_{12,} \nu_{13}\}$, ν_{11} is represented as “1”, ν_{12} is represented as “2”, and ν_{13} is represented as “3” in the first number of each test case, so are the other parameters. The output is composed of three parts, the first part is the extended test cases which are generated by the first two parameters, the second part is the test cases which are generated by merging in the last step and the third is a statement describing the number of uncovered pairs before merging. In the program, the first part and the other parts are separated by a dotted line.

The simulation result for five three-valued parameters is shown in figure 5. All the combination pairs which are generated by two parameters are covered by the thirteen test cases, and * in test cases can be assigned any values of corresponding parameter. Before merging, there are ten uncovered pairs in *PS*.



```
  [ 1 ,1 ,1 ,1 ,1 ]
  [ 1 ,2 ,2 ,2 ,2 ]
  [ 1 ,3 ,3 ,3 ,3 ]
  [ 2 ,1 ,2 ,3 ,1 ]
  [ 2 ,2 ,1 ,1 ,3 ]
  [ 2 ,3 ,1 ,2 ,2 ]
  [ 3 ,1 ,3 ,2 ,1 ]
  [ 3 ,2 ,1 ,3 ,2 ]
  [ 3 ,3 ,2 ,1 ,3 ]

- - - - - - - - - - -
  [ 2 ,2 ,3 ,1 ,1 ]
  [ * ,1 ,3 ,1 ,2 ]
  [ * ,1 ,* ,2 ,3 ]
  [ * ,3 ,* ,* ,1 ]
The most number of uncovered pairs in storage space is 10
13
```

Figure 5 The user interface for simulation

Table 4 and Table 5 show the simulation result of *IVO* and *IPO* for five different inputs, and the details of order of the parameters for the five examples are shown in appendix A. In Table 4, the number of uncovered pairs is the sum of all the numbers of uncovered pairs before merging.. Obviously, test cases generated by *IVO* are less than *IPO* for the first four examples.

Experiment cases:

E_1: five parameters (5 3-valued).

E_2: seven parameters (1 6-valued, 2 5-valued, 1 4-valued, 2 3-valued, 1 2-valued).
E_3: ten parameters (2 7-valued, 2 6-valued, 3 5-valued, 3 4-valued).
E_4: twenty-five parameters (5 7-valued, 4 5-valued, 4 4-valued, 6 3-valued, 6 2-valued).
E_5: forty parameters (8 7-valued, 8 5-valued, 8 4-valued, 8 3-valued, 8-2valued).

IVO	E_1	E_2	E_3	E_4	E_5
# of test cases	13	33	55	78	131
# of uncovered pairs	10	6	18	132	613

Table 4 Result of the five examples for *IVO*

IPO	E_1	E_2	E_3	E_4	E_5
# of test cases	14	37	61	86	107
# of uncovered pairs	7	10	45	141	150

Table 5 Result of the five examples for *IPO*

However, in the fifth example, the number of test cases generated by *IVO* is more than that by *IPO*. To clarify such a condition, two experiments, E_6 and E_7 are designed. E_6 is the case for two-valued of 10, 20, 30 and 40 parameters corresponding. E_7 is three-valued instead. The result for E_6 and E_7 are show in Figure 6 and Finger 7 respectively. From the results of E_6 and E_7, a critical disadvantage for *IVO* found. If all input parameters are given the same number of values, the simulation result of *IVO* is worse than *IPO*.

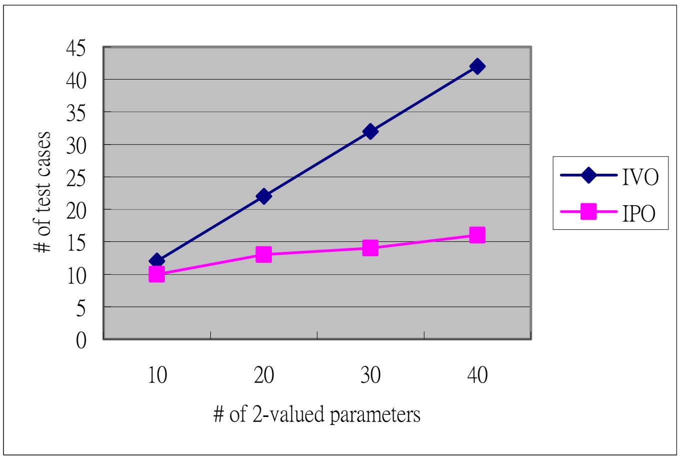


Figure 6 Simulation results of *IVO* and *IPO* for E_6

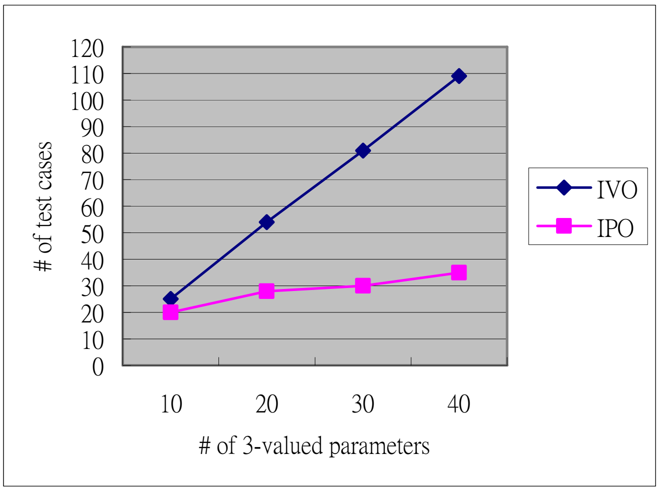


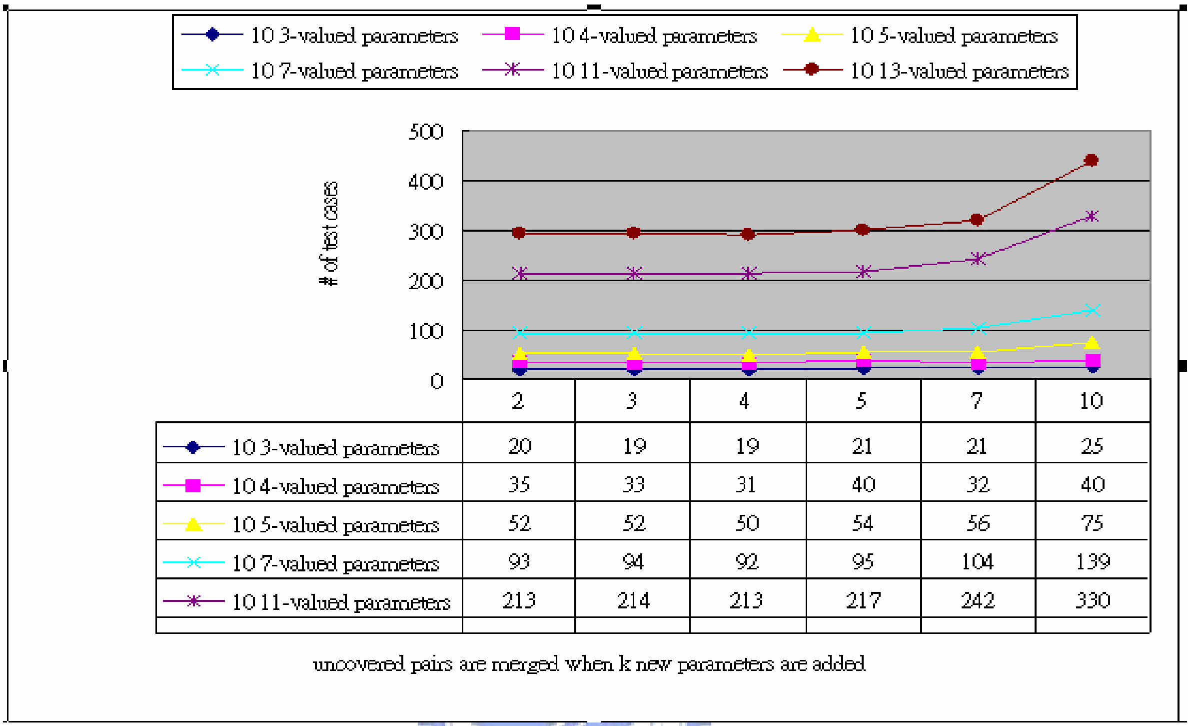
Figure 7 Simulation results of *IVO* and *IPO* for E_7

In Figure 6, the input is 40 two-valued parameters and the difference in test case size of *IVO* and *IPO* is 26. Furthermore, in Figure 7, the inputs are 40 three-valued parameters, the difference in test case size between *IVO* and *IPO* is 74. With the inputs are *n* *k*-valued parameters, larger *n* and *k* are, larger the difference in test case size becomes.

The reason is described as follows. Because *IVO* merges all the uncovered pairs in the last step, the number of test cases can be increased heavily during in this step. If all input parameters are given the same number of values, more uncovered pairs (v_{ij}, v_{kq}) are generated when later parameters P_k are added, $1\leq i\leq k$, $1 \leq j\leq |V_i|$, $1 \leq q\leq |V_k|$. The size of test cases in *IVO* is larger than that with *IPO* after uncovered pairs are merged.

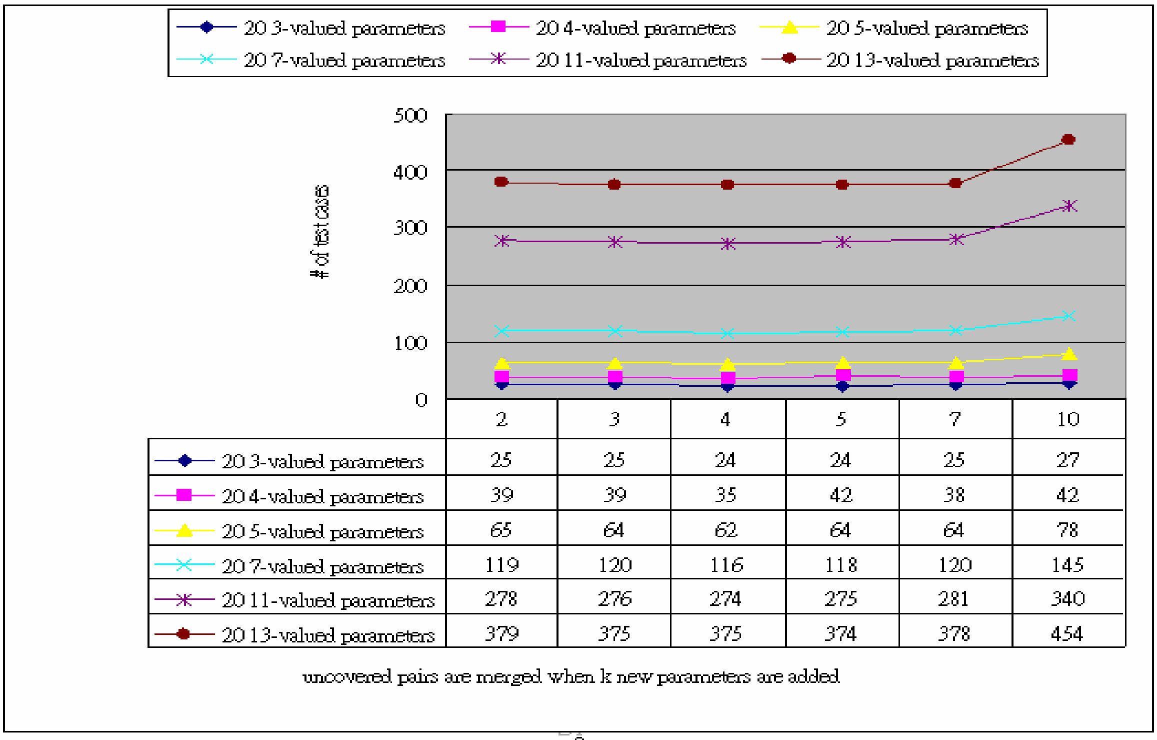
Section 3.2 The *Modified IVO (MIVO)*

To overcome the disadvantage of *IVO* discussed in above section. We propose another algorithm, called *modified IVO* (*MIVO*). *MIVO* executes mergence once when *k* new parameters are added. If the number of parameters left is less than *k* and *PS* is not empty, *MIVO* execute the final mergence in the last step. According to the experiment shown in Figure 8 and Figure 9, the *MIVO* gets better result when *k* is assigned 4.



	2	3	4	5	7	10
◆ 10 3-valued parameters	20	19	19	21	21	25
■ 10 4-valued parameters	35	33	31	40	32	40
▲ 10 5-valued parameters	52	52	50	54	56	75
× 10 7-valued parameters	93	94	92	95	104	139
✳ 10 11-valued parameters	213	214	213	217	242	330

Figure 8 Simulation result for different *k*'s with 10 *n*-valued parameters



	2	3	4	5	7	10
◆ 20 3-valued parameters	25	25	24	24	25	27
■ 20 4-valued parameters	39	39	35	42	38	42
▲ 20 5-valued parameters	65	64	62	64	64	78
× 20 7-valued parameters	119	120	116	118	120	145
✳ 20 11-valued parameters	278	276	274	275	281	340
● 20 13-valued parameters	379	375	375	374	378	454

Figure 9 Simulation result for different *k*'s with 20 *n*-valued parameters

We analyze above experiments of figure 8 and 9, and find that when k is assigned 2, 3 or 4, the situation of the disadvantage of *IVO* is not happened, and it gets better result when k is assigned larger number. But when k is assigned 5, the situation of the disadvantage of *IVO* appears, and the situation is the main reason that the number of test cases for k = 5is larger than that k =4.

MIVO algorithm shows as follows:

Algorithm: Modified In-Value-Order (*MIVO*)

```
Begin:
    for the first two parameters, $P_1$ and $P_2$
    $TS: = \{[v_1, v_2] \mid v_1 \in V_1, v_2 \in V_2\}$
    if $n = 2$ then stop;
    $\forall$ parameter $P_i$, $i = 3, \ldots, n$;
    {
        $\forall \gamma \in V_1 \cup V_2 \cup \ldots \cup V_{i-1}, \delta \in V_i$
            add $(\gamma, \delta)$ to $PS$
        for $1 \le j \le |V_i|$ that $T_j=[a_1, a_2, \ldots, a_{i-1}]$
        {
            extend $T_j$ with $v_{ij}$ such that $T_j=[a_1, a_2, \ldots, a_{i-1}, v_{ij}]$
            and remove $(a_1, v_{ij}), (a_2, v_{ij}), \ldots, (a_{i-1}, v_{ij})$ from $PS$
        }
        for $|V_i| < j \le |TS|$
        {
            can_count = 0
            $T_j=[a_1, a_2, \ldots, a_{i-1}]$
```

```
      for $1 \le k \le |V_i|$
       {
        count = $|\{(a_z, v_{ik}) | z = 1,2,\ldots i-1, (a_z, v_{ik}) \in PS\}|$
        if(count > can_count)
          {
            can_count = count
            $c = k$
          }
       }
      extend $T_j$ with $v_{ik}$ such that $T_j = [a_1, a_2, \ldots, a_{i-1}, v_{ik}]$
      and remove $(a_1, v_{ik}), (a_2, v_{ik}), \ldots, (a_{i-1}, v_{ik})$ from $PS$
  }
if ($i$ mod 4 = 0)
  {
    let $TS'$ is a test case set, $TS' = \varnothing$
       $\forall (\alpha, \beta) \in PS, \alpha \in V_i, \beta \in V_j$
          {
          if ($\exists$ some test cases $T = [a_1, a_2, \ldots, a_n]$, $T \in TS'$
          that($a_i = \alpha, a_j = \beta$) or ($a_i = *, a_j = \beta$) or($a_i = a_j = *$))
             set $a_i = \alpha, a_j = \beta$ and remove $(\alpha, \beta)$ from $PS$
          else
             generate a new test case $T = [a_1, a_2, \ldots, a_n]$ in
             which $\forall z = 1\ldots n, z \neq i, j, a_z = *, a_i = \alpha, a_j =$
             $\beta$, add $T$ to $TS$, remove $(\alpha, \beta)$ from $PS$
```

```
41                              }
42                          if($PS$ is empty)
43                          * is assigned any values of corresponding
44                           parameter
45                      $TS = TS \cup TS'$
46              }
47          }
48      $TS' = \varnothing$
49          $\forall (\alpha, \beta) \in PS, \alpha \in V_i, \beta \in V_j$
50            {
51              if ( $\exists$ some test cases $T = [a_1, a_2, \ldots, a_n]$, $T \in TS'$ that ($a_i =$
52              $\alpha, a_j = \beta$) or ($a_i = *, a_j = \beta$) or ($a_i = a_j = *$))
53                  set $a_i = \alpha, a_j = \beta$ and remove $(\alpha, \beta)$ from $PS$
54              else
55                  generate a new test case $T = [a_1, a_2, \ldots, a_n]$ in which
56                  $\forall z = 1 \ldots n, z \neq i, j, a_z = *, a_i = \alpha, a_j = \beta$, add $T$ to $TS'$
57                  remove $(\alpha, \beta)$ from $PS$
58            }
59      $TS = TS \cup TS'$
60  end
```

In *MIVO* algorithm, test cases are generated by the first two parameters in line 2. When parameter is introduced in line 6, the generated uncovered pairs are put in *PS* in line 7. The extension for existing test cases is similar to previous algorithm, except that each mergence is done for adding every 4 new parameters from line 8 to line 46. Finally, the rest uncovered pairs are merged to test cases from line 48 to line59.

The extension results of *MIVO* for E_6 and E_7 are shown in Figure 10 and Figure 11. Obviously, *MIVO* have great improvement in the number of test cases of *MIVO* for parameters with the same number of values.

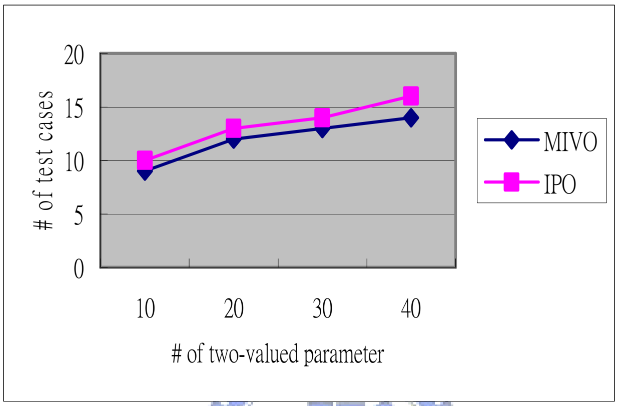


Figure 10 Simulation results of *MIVO* and *IPO* for *n* two-valued parameters

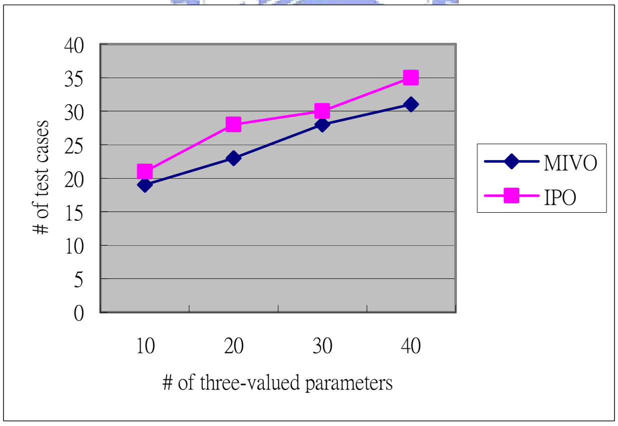


Figure 11 Simulation results of *MIVO* and *IPO* for *n* three-valued parameters

Table 6, 7 and 8 show the simulation results of *MIVO, IPO,* and *IVO* for the examples E_4, E_5, E_6, E_7, E_8 and E_9, the parameters input order is recorded in Appendix A.

Experiment cases:

E_8: twenty parameters (20 4-valued).

E_9: twenty-five parameters (6 13-valued, 5 11-valued, 7 9-valued, 7 7-valued).

E_{10}: thirty parameters (10 15-valued, 10 13-valued, 10 10-valued).

E_{11}: forty-three parameters (5 10-valued, 6 8-valued, 8 6-valued, 4 5-valued, 6 4-valued, 6 3-valued, 8 2-valued).

E_{12}: fifty-four parameters (6 10-valued, 6 9-valued, 6 8-valued, 6 7-valued, 6 6-valued, 6 5-valued, 6 4-valued, 6 3-valued, 6 2-valued).

MIVO	E_4	E_5	E_8	E_9	E_{10}	E_{11}	E_{12}
# of test cases	70	89	35	262	420	152	178
# of uncovered pairs	46	108	76	218	602	110	200
time in seconds	0.016	0.031	0.016	0.032	0.062	0.047	0.063

Table 6 Simulation result of *MIVO* for six examples.

IPO	E_4	E_5	E_8	E_9	E_{10}	E_{11}	E_{12}
# of test cases	86	107	43	298	483	184	216
# of uncovered pairs	141	150	28	424	828	395	312
time in seconds	0.016	0.048	0.015	0.047	0.094	0.063	0.109

Table 7 Simulation result of *IPO* for six examples.

IVO	E_4	E_5	E_8	E_9	E_{10}	E_{11}	E_{13}
# of test cases	78	131	64	419	1040	221	385
# of uncovered pairs	132	613	432	1770	8673	639	3058
time in seconds	0.016	0.031	0.015	0.016	0.157	0.031	0.046

Table 8 Simulation result of *IVO* for six examples.

Chapter 4 Extending A Test Case Set Based On Duplication

In the chapter, we discuss an algorithm to duplicate the data pairs. In the algorithm, test cases can be quickly extended under some constraints. Section 4.1 introduces the duplicate algorithm. Section 4.2 introduces a pairwise graph model. Section 4.3 proves the correctness of the duplicate algorithm and discusses influence when adding a *k*-valued parameter to a test case set which covers pairwise perfectly for *n*.

Section 4.1 An Extension Algorithm With Duplicate Technique

Definition 4.1. (pairwise perfectly coverage): A test case set *TS* is claimed to **cover pairwise perfectly for** *n*, when there are $n+1$ parameters, each of which has *n* values and all pairs appears distinctly in *TS*.

For example, given three three-valued parameters $P_1 = \{v_{11}, v_{12}, v_{13}\}$, $P_2 = \{v_{21}, v_{22}, v_{23}\}$ and $P_3 = \{v_{31}, v_{32}, v_{33}\}$, and the test case set $TS = \{[v_{11}, v_{21}, v_{31}][v_{11}, v_{22}, v_{32}][v_{11}, v_{23}, v_{33}][v_{12}, v_{21}, v_{32}][v_{12}, v_{22}, v_{33}][v_{12}, v_{23}, v_{31}][v_{13}, v_{21}, v_{33}][v_{13}, v_{22}, v_{31}][v_{13}, v_{23}, v_{32}]\}$ which cover all pairs without repetition. Thus, *TS* covers pairwise perfectly.

Consider the system which has $n+1$ parameters, and each parameter contains *n* distinct values, $n \geq 1$. The testing data need $n^2(n) + n^2(n-1) + \ldots + n^2{*}1 = n^2\,[(n) + (n-1) + \ldots +1] = n^3(n+1)/2$ pairs at most. Let each test case have $n+1$ values and each pair be generated by choosing any two values of them. Each of these test cases covers $n(n+1)/2$ pairs at most and the least number of test cases is $[n^3\,(n+1)/2]/\,n(n+1)/2 = n^2$.

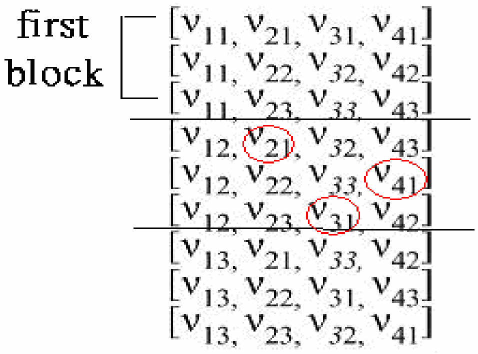
Definition 4.2. (block): Let a test case set *TS* cover pairwise perfectly for n. A ***block*** is a set containing n test cases which have the same value of parameter P_1. A block B = $\{T_i \mid 1 \leq i \leq n,\ T_i$ is a test case and T_i has the same value of $P_1\}$.

Proposition 4.1: Let a test case set *TS* cover pairwise perfectly for n. Each test case can be assigned to some block and the *TS* can be divided into n different blocks

Proof: Because *TS* covers pairwise perfectly for n, all combination pairs are in *TS*. Consider v_{11} and v_{2i}, $1 \leq i \leq n$. The n pairs (v_{11}, v_{21}), (v_{11}, v_{22}), …, (v_{11}, v_{2n}) are in *TS*. $\forall v_{1j}$, $1 \leq j \leq n$, there are n test cases whose first value is v_{1j}, thus *TS* can be divided into n different blocks.□

Let a test case set *TS* cover pairwise perfectly for n. According to proposition 4.1, there exist a block in which all test cases for P_1 have the same value, and these test cases can be formed as $[v_{1i}, v_{2j}, v_{3j}, \ldots, v_{(n+1)j}]$, $1 \leq i, j \leq n$. Such a block is called the first block, and W.L.O.G., in following sections, the first block is used to prove or clarify the property for the duplicate algorithm.

Figure 12 is an example that the test case set *TS* covers pairwise perfectly for 3 and test cases are divided into three blocks.



$[v_{11}, v_{21}, v_{31}, v_{41}]$
$[v_{11}, v_{22}, v_{32}, v_{42}]$
$[v_{11}, v_{23}, v_{33}, v_{43}]$
$[v_{12}, v_{21}, v_{32}, v_{43}]$
$[v_{12}, v_{22}, v_{33}, v_{41}]$
$[v_{12}, v_{23}, v_{31}, v_{42}]$
$[v_{13}, v_{21}, v_{33}, v_{42}]$
$[v_{13}, v_{22}, v_{31}, v_{43}]$
$[v_{13}, v_{23}, v_{32}, v_{41}]$

Figure 12 Pairwise perfectly for 3.

There're three properties for a test case set *TS* which covers pairwise perfectly for n.

Property 1. $\forall$test cases $T_a, T_b \in B_c$, $1\leq c\leq n$, if $v_{ij} \in T_a$, $v_{ij} \notin T_b$, $1<i\leq n+1$, $1\leq j\leq n$.

Property 1 holds because for any two test cases T_a and T_b in the same block, if $v_{ij} \in T_a$, $v_{ij}\in T_b$, $1<i\leq n+1$ and $1\leq j\leq n$, the pair (v_{1k}, v_{ij}), $1\leq k\leq n$, appears repeatedly,

Property 2. $\forall$test case $T_a \in B_c$, $c\neq 1$, if $v_{ij} \in T_a$,$v_{kj} \notin T_a$, $i\neq k$, $1< i, k\leq n+1$, $1\leq j\leq n$.

Property 2 holds because $\forall$test case $T_a \in B_c$, $c\neq 1$, if $v_{ij} \in T_a$,$v_{kj} \in T_a$, the pair (v_{ij}, v_{kj}) appears repeatedly with test cases in the first block.

```
The Duplicate Algorithm:
Input: A test case set TS covers pairwise perfectly for $n$, and $m$ $n$-valued parameters
waiting to be extended, where $1\leq m\leq n+1$.
Begin:
    $\forall$parameter $P_{(n+1)+i}$, $1\leq i\leq m$.
        $\forall$ test case $T_a$, a=1,2,…,$n^2$
            $\forall v_{ij}$ is the $j^{th}$ value of parameter $P_i$, $1\leq j\leq n$.
                if ($v_{ij}\in T_a$) extend $T_a$ by adding $v_{(n+i+1)j}$
End
```

The extended test case set loses $n(n-1)$ pairs for each new added parameter. This is proved in section 4.3. Take figure 13 as an example, there are four three-valued parameter $P_1 = \{v_{11}, v_{12}, v_{13}\}$, $P_2 = \{v_{21}, v_{22}, v_{23}\}$, $P_3 = \{v_{31}, v_{32}, v_{33}\}$ and $P_4 = \{v_{41}, v_{42}, v_{43}\}$ and the test case set covers pairwise perfectly for 3. For four three-valued parameter $P_5 = \{v_{51}, v_{52}, v_{53}\}$, $P_6 =$

$\{v_{61}, v_{62}, v_{63}\}$, $P_7 = \{v_{71}, v_{72}, v_{73}\}$ and $P_8 = \{v_{81}, v_{82}, v_{83}\}$ being added, each test cases are extended according to the duplicate algorithm and the extension result is shown in figure 13.

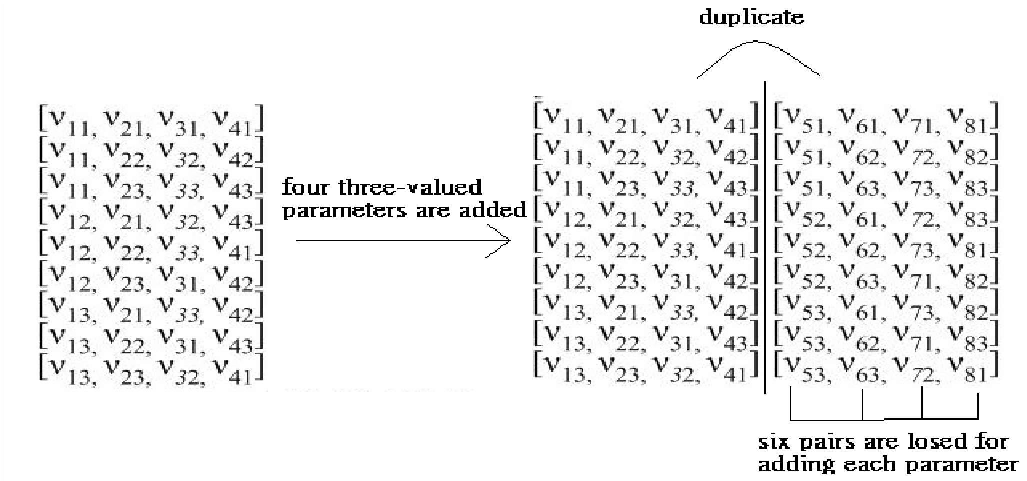


Figure 13　Example of the duplicate algorithm

After executing the duplicate algorithm, there are some uncovered pairs, and the form of uncovered pairs must be $(v_{i1}, v_{(n+i+1)2})$, $(v_{i1}, v_{(n+i+1)3})$,…, $(v_{i1}, v_{(n+i+1)n})$,…,$(v_{in}, v_{(n+i+1)1})$, $(v_{in}, v_{(n+i+1)(n-1)})$, $1 \leq i \leq n+1$. Because $n*(n-1)$ pairs are lost for adding each new parameter, $n*(n-1)$ test cases are need to cover these uncovered pairs. The test cases generation algorithm is as follows:

```
Incremental -Test case - generation algorithm (ITG)
Input: The input is a TS which is generated by executing the duplicate algorithm, and the input to the duplicate algorithm is a test case set which covers pairwise perfect for n, and m n-valued parameters are waited to be extended, 1≤ m≤ n+1.
Begin
    ∀i=1,…n
      {
        ∀j=1,…n and i≠j
          {
```

```
                $v_{ij}$ is the $j^{th}$ value of parameter $P_i$, $1 \le j \le |V_i|$.
                generate a test case $[v_{1i}, v_{2i}, \ldots, v_{(n+1)i}, v_{(n+2)j}, v_{(n+3)j}, \ldots, v_{(n+m+1)j}]$ to TS
            }
        }
    End
```

Section 4.2 The Pairwise Graph Model.

To assist the proof of the duplicate algorithm, the pairwise graph model is introduced first.

Definition 4.3. (pairwise graph model): A pairwise graph is an undirected graph $G = (N, A)$. N is a set of nodes, of which a node is named as n_{ij}, $1 \le i,j \le n$, and n_{ij} represents the j^{th} value of P_i. For any arc $e \in A$, e connects n_{ij} and $n_{(i+1)k}$, $i \ge 1$, $1 \le j \le |V_i|$, $1 \le k \le |V_{i+1}|$.

Definition 4.4. (complete path): Given n parameters, a **complete path** is defined as a path in a pairwise graph that starts from n_{1i} and ends in n_{nj}, $1 \le i \le |V_1|$, $1 \le j \le |V_n|$. In a pairwise graph, loops are not allowed.

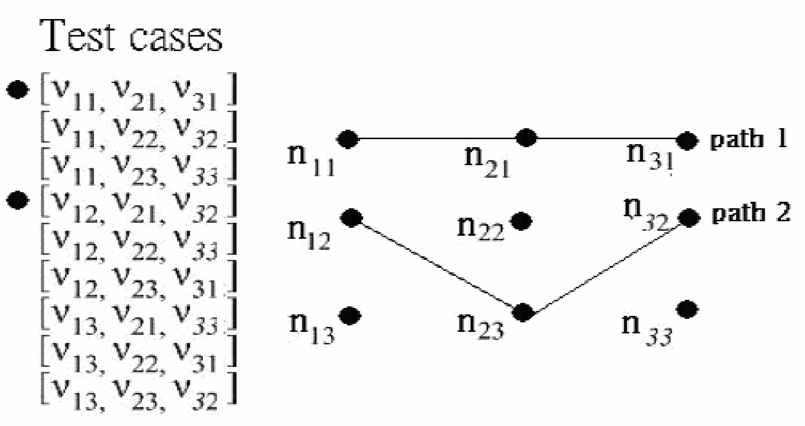


Figure 14. Pairwise graph

Consider the example shown above, where three three-valued parameters are $P_1 = \{v_{11}, v_{12}, v_{13}\}$, $P_2 = \{v_{21}, v_{22}, v_{23}\}$, $P_3 = \{v_{31}, v_{32}, v_{33}\}$. In the pairwise graph, $N = \{n_{11}, n_{12}, n_{13}, n_{21}, n_{22}, n_{23}, n_{31}, n_{32}, n_{33}\}$ and path 1 consists of $[v_{11}, v_{21}, v_{31}]$ and path 2 consists of $[v_{12}, v_{21}, v_{32}]$, and both paths are complete paths.

If a test case set covers pairwise perfectly for *n*, its corresponding pairwise graph has the following properties:

Property 1. For all complete paths, each arc is walked through once.

Property 1 holds because for each arc, if the arc is walked through twice, the corresponding pair appears twice in the test case set.

Property 2. For any paths, the pairs of the start and end nodes for any path are different to other paths that of another path.

Property 2 holds because for all paths, if the pair of the start and end nodes for any path is the same with another path, the corresponding pair appears twice in the test case set.

Take figure 15 as an example, there are three three-valued parameter, $P_1 = \{v_{11}, v_{12}, v_{13}\}$, $P_2 = \{v_{21}, v_{22}, v_{23}\}$ and $P_3 = \{v_{31}, v_{32}, v_{33}\}$. Consider the paths for $[v_{13}, v_{21}, v_{33}]$, $[v_{13}, v_{21}, v_{31}]$ and $[v_{13}, v_{23}, v_{31}]$. The paths corresponding to $[v_{13}, v_{21}, v_{33}]$ and $[v_{13}, v_{21}, v_{31}]$ go through arc 1, and the property 1 is violated. For $[v_{13}, v_{21}, v_{31}]$ and $[v_{13}, v_{23}, v_{31}]$, the pairs of start and end node are the same. Therefore the property 2 is violated.

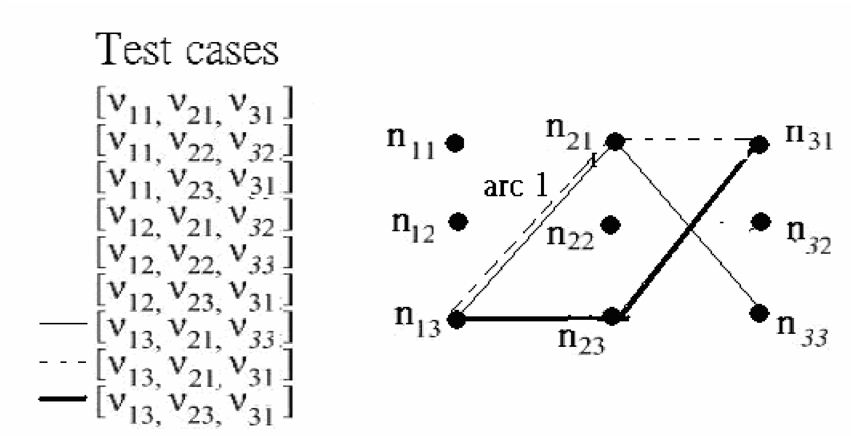


Figure 15. Example for violating two property of pairwise perfectly

Section 4.3 Proof Of The Duplicate Algorithm

To prove the correctness of the duplicate algorithm, some propositions are introduced firstly.

Definition 4.5. (value independent): If all the pairs generated for two parameters P_i and P_j are distinct in the test case set *TS*, P_i and P_j are claimed "**value independent**" in the *TS*.

For example, there are three parameters $P_1 = \{v_{11}, v_{12}, v_{13}\}$, $P_2 = \{v_{21}, v_{22}\}$ and $P_3 = \{v_{31}, v_{32}, v_{33}\}$ and the test case set $TS = \{[v_{11}, v_{21}, v_{31}], [v_{11}, v_{22}, v_{32}], [v_{12}, v_{21}, v_{33}], [v_{12}, v_{22}, v_{32}],$

[v_{13}, v_{21}, v_{31}], [v_{13}, v_{22}, v_{32}]}. Each pair (v_{1i}, v_{3j}) is distinct in the *TS*, $1\leq i, j\leq 3$, and parameter P_1 and P_3 are value independent in the *TS*. Because (v_{21}, v_{31}) and (v_{22}, v_{32}) appear twice, parameter P_2 and P_3 are not value independent in the *TS*.

To prove the following propositions, an Incremental-*TS*-extension (ITE) method is introduced. The ITE method is used to duplicate the permutation of pairs for a parameter in the test case set.

```
Incremental- TS -extension (ITE) method
Input: A TS covers all pairs which are generated by the k parameters. Let the parameter $P_i$ be one of the k parameters, and $P_{k+1}$ is a new parameter which is waited to be added to TS, $|V_i|\leq |V_{k+1}|$.
Begin
    $\forall$ test case $T_a \in TS$
    {
        $v_{ij}$ is the j^th value of parameter $P_i$, $1\leq j\leq |V_i|$.
        if($v_{ij} \in T_a$)
            extend $T_a$ by adding $v_{(k+1)j}$
    }
End
```

Proposition 4.2: Let two parameters P_i and P_j be value independent in the test case set *TS*, $1\leq i, j\leq k$. When a new parameter P_{k+1} is added, $|V_{k+1}|\geq |V_i|$. After executing the ITE method of the input parameters are P_i and P_{k+1}. The two parameters P_j and

P_{k+1} are also value independent in the extended test case set.

Proof: Assume that P_i and P_j are value independent in the extended test case set, but parameters P_j and P_{k+1} are not value independent. There are some pairs $(v_{jm}, v_{(k+1)n})$ where $1 \leq m \leq |V_j|$, $1 \leq n \leq |V_{k+1}|$, which appear repeatedly in the extended test case set. The pairs (v_{in}, v_{jm}) also appear repeatedly in the extended test case set, and parameters P_i and P_j are not value independent. That is a contradiction, and therefore parameters P_j and P_{k+1} are value independent.□

Proposition 4.3: Let the test case set *TS* cover pairwise perfectly for *n*. For any two test cases in different blocks, there is only one value in common.

Proof: $\forall$ test case $T_a \in B_i$, $T_b \in B_j$, $i \neq j$, if there are two values in common in T_a and T_b, the pair generated by the two values appears twice and *TS* doesn't cover pairwise perfectly for *n*. That is a contradiction.

Because test cases are divided into blocks according to the value of parameter P_1, the value of P_1 is different for any two test cases in different block. Consider the values of parameter P_2 to P_{n+1}, because *TS* cover pairwise perfectly for *n*, all values appear once in a block. $\forall$ each value v_{kl} in the test case $T_a \in B_i$ where $k \neq 1$ and $1 \leq i, l \leq n$, the test case $T_b \in B_j$ can be found that v_{kl} is in T_b, $i \neq j$. For any two test cases in different blocks, there is only one variable that has the same value.

Proposition 4.4: Let the test case set *TS* cover pairwise perfectly for *n*. If an *n*-valued parameter is added, the extended test case set lose $n(n-1)$ pairs at least without increasing the number of test cases.

Proof: W.L.O.G, assume that the first value is v_{11} for each test case in the first block and figure 16 is the corresponding pairwise graph. When parameter P_{n+2} is added, two cases are discussed as follows. In case 1, all test cases in the first block are extended by adding n values of $V_{n+2.}$ In case 2, test cases in the first block are extended by adding j values of V_{n+2}, $0 \le j \le n-1$.

Case1: The i^{th} test case in the first block is extended by adding $v_{(n+2)j,}$ $1 \le i, j \le n$. There are $n+1$ space for $n+2$ parameters and $n-1$ space for n values in pairwise graph. For each complete path except the first block, if $i = j$ and the last node is $n_{(n+2)k,}$ the node n_{ik} can be found in the same complete path and is shown in the case 1 of figure 17. The pair ($v_{ik,}$ $v_{(n+2)k}$) appears repeatedly with the pair in the first block. If $i \neq j$, the proof is the same with above and the repeated pairs become ($v_{ik,}$ $v_{(n+2)l}$), $k \neq l$, and the corresponding graph is the case2 of Figure 17. $\forall$ test case $T_a \in B_{k,}$ $k \neq 1$, each extended test case has least one repeated pairs and the extended test case set loses $n(n-1)$ pairs at least.

Case2: Test cases in the first block are extended by adding j values of V_{n+2}, $0 \le j \le n-1$. If all test cases in the first block are extended, there are $n-j$ repeated pairs in the first block. According to proposition 4.3, there is one same value for any two test cases in different blocks. For the remainder $n(n-1)$ test cases, only $n-j$ test cases have no repeated pairs, and each of the n^2-2n+j test cases has least one repeated pair, so the total lost pairs are $n^2-2*n+j+n-j = n\,(n-1)$.□

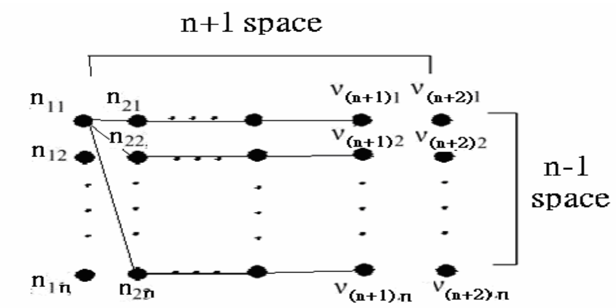


Figure 16 Pairwise graph for *n*+2 *n*-valued parameters

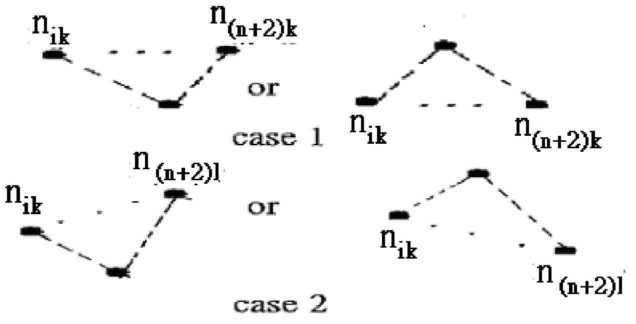


Figure 17 Form of the Repeated covered pairs

Proof of the correctness of the Duplicate Algorithm: A test case set *TS* covers pairwise perfectly for *n*. *m n*-valued new parameters are waited to be extended, $1 \leq m \leq n+1$. After extension by using duplicate algorithm, the test case set *TS* loses $n(n-1)$ pairs for adding a new parameter and the test cases set has the most pairwise coverage than other test case sets with the same size.

Proof: Because the test case set *TS* covers pairwise perfectly for *n*, any two parameters between P_1 and P_{n+1} are value independent in *TS*. According to proposition 4.2, the parameter P_{n+i+1} is value independent with former parameters except parameter P_i, $1 \leq i \leq m$, and the number of repeated pairs generated by P_i and P_{n+i+1} is $n\ (n-1)$. According to proposition 4.4, the least number of lost pairs is $n\ (n-1)$ when adding a new parameter.

Therefore, the *TS* extended by the duplicate algorithm still cover the most pairs without increasing number of test cases.□

The advantage for using the duplicate method is when new parameters are added, the test cases can be extended quickly. The duplicate method reduces large time in extension and retains high coverage rate.v_{1i},

Finally, some situations are discussed as follows. Let the test case set *TS* cover pairwise perfectly for n. There are $k*n(n+1)$ new pairs are generated when a k-valued parameter is added, $k\neq n$. Without increasing the number of test cases, how many pairs are lost at least after extension for $k< n$, $n< k\leq n^2$ and $k>n^2$?

$k< n$: the least number of lost pairs is zero. There are $k*n(n+1)$ new pairs are generated when adding a new parameter and the extended test cases can cover $n^2(n+1)$ new pairs at most, so the least number of lost pairs is zero.

$n< k\leq n^2$: the least number of lost pairs is $n(n+1)(k-n)+n^2-k$. If there are no repeated pairs in the extended test cases, the extended test cases cover $n^2(n+1)$ new generated pairs. Because $k\leq n^2$, repeated pairs are not found in only k extended test cases, and the remainder $n^2- k$ test cases have more than one repeated pair. The reason is same with the case 2 of proposition 4.4, so the number of lost pairs is $n(n+1)k-n^2(n+1) + n^2-k$.

$k>n^2$: the least number of lost pairs is $n(n+1)(k-n)$. If there are no repeated pairs in the extended test cases, the extended test cases cover most $n^2(n+1)$ new pairs, so the number of lost pairs is $n(n+1)k-n^2(n+1) = n(n+1)(k-n)$.

Chpater 5 Conclusion and Future Work

Pairwise testing is a special case of *n*-way testing. In this paper, *MIVO* test generation strategy for pairwise testing is presented and implemented. With our simulation, to reach the same coverage, *MIVO* generates fewer test cases than *IPO,* i.e., in test cases generation, *MIVO* is more effective.

The duplicate algorithm is also introduced and clearly proved in this paper. Using the duplicate algorithm, the extended test cases retain high pair's coverage. With the advantage of using the algorithm is that test cases can be easily extended and large amount of computing time for extension can be saved

Much works also remains to be done to validate the result of *MIVO*. In the future, we hope that *MIVO* testing strategy can be implemented to test real software systemes and more empirical results are gathered to confirm the usefulness of *MIVO*.

References

[1] Y.Lei and K.C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," Technical Report TR-2001-03, Dept. of Computer Science, North Carolina State Univ., Raleigh, North Carolina, May, 2001.

[2] Y.Lei and K.C. Tai. "A Test Generation Strategy for Pairwise Testing", IEEE Trans of Software Engineering, Vol 28, NO.1, Jan 2002.

[3] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The AETG System: An Approach to Testing Based on Combinational Design," IEEE Trans. Software Eng., vol. 23, no.7, pp.437-443, July 1997.

[4]D. M. Cohen, S. R. Dalal, M.L. Fredman, and G.C. Patton, "The combinatorial design approach to automatic test generation." IEEE Software, pages 83-87, September 1996.

[5]Glenford J. Myers, "The Art of Software Testing," Senior Staff Member, IBM Systems Research Institute, Lecturer in Computer Science, Polytechinic Institute of New York.

[6]Roger S. Pressman, Ph.D.,"Software Engineering: A Practitioner's Approach", Sixth Edition. 2005.

[7]R.Mandl, "Orthogonal Latin squares: An application of experiment design to compiler testing", Communications of the ACM, Vol. 28 NO. 10(October 1985) pp.1054-1058.

[8]Dennis. Jeffrey and Neelam Gupta," Test Case Prioritization Using Relevant Slices", Annual International Computer Software and Applications Conference (COMPSAC'06), IEEE, 2006.

[9] Renee C. Bryce and Charles J. Colbourn, "Test Prioritization for Pairwise Interaction Coverage", A-MOST'05, May 15-21,2005, St. Louis, Missouri, USA.

Copyright 2005 ACM.

[10] James A. Jones and Mary Jean Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/ Decision Coverage", IEEE Trans., Software Eng ., 2003.

[11]Soumen Maity and Amiya Nayak "Improved Test Generation Algorithms for Pair-wise Testing" 2005 IEEE, Software Reliability Engineering.

[12]S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing Test Cases for Regression Testing," Proc. ACM Software Testing and Analysis, pp. 102-112, Aug. 2000.

[13] D. M. Cohen, S. R. Dalal, A. Kajla, and G.C. Patton, "The automatic Efficient Test Generator (AETG) System", Proceedings of the 5th IEEE International Symposium on Software Reliability Engineering, pages 303-309, Nov. 1994

[14] D. M. Cohen, S. R. Dalal, J. Parelius, and G.C. Patton, "Combination Design Approach to Automatic Test Generation", IEEE Software, 13: pages 83-88,1996

Appendix A

The input order of the example for simulation

E_1: five parameters (5 3-valued).

E_2: seven parameters (5, 6, 3, 2, 3, 4, 5)

E_3: ten parameters (7, 4, 6, 5, 7, 5, 5, 6, 4, 4)

E_4: twenty-five parameters (3, 3, 4 ,7, 2, 5, 4, 2, 7, 3, 5, 2, 2, 2, 4, 3 ,3 ,7 ,2 ,4 ,3, 5, 5, 7, 7)

E_5: forty parameters (5, 4, 2, 3, 7, 2, 4, 4, 5, 3, 2, 2, 5, 7, 4,7, 3,3,4,5, 3,4,5,2, 2,7,2,3, 7,7,5,4,7,2,3,3,4,5,5,7)

E_8: twenty parameters (20 4-valued).

E_9: twenty-five parameters (11, 11, 13, 7, 9, 9, 13, 13, 11, 7, 7, 7, 13, 11, 9, 7, 9, 7, 13, 11, 7, 9, 9, 13, 7, 9)

E_{10}: thirty parameters(10, 13,10,15,13,13,10, 15,13,15,10,10,15,13,13,10, 15,15,10,13, 10,15, 13,10,15,10,13, 13,15,15)

E_{11}: forty-three parameters (6, 5, 5, 4, 3, 2, 6, 8, 4, 2, 6, 5, 3, 3, 10, 4, 2, 8, 2, 2, 6, 8, 3, 10, 2, 6, 6 , 5, 4, 4, 10, 2, 8, 8, 6, 4, 10, 2, 3, 3, 6, 8, 10).

E_{12}: fifty-four parameters(6, 9, 2, 10, 3, 5, 5, 7, 6, 3, 2, 4, 4, 9, 8, 2, 10, 7, 3, 9, 5, 5, 8, 4, 8, 10, 6, 3, 3, 8, 9, 5, 7, 6, 7, 5, 3, 10, 10, 2, 8, 6, 7, 4, 7, 6, 8, 4, 10, 2, 2, 9, 4, 9).