Improving IPOG's Vertical Growth Based on a Graph Coloring Scheme

Feng Duan, Yu Lei
University of Texas at Arlington
Arlington, TX 76019, USA
feng.duan@mavs.uta.edu,
ylei@cse.uta.edu

Linbin Yu
Facebook Inc.
Menlo Park, CA 94025, USA
linbin@fb.com

Raghu N. Kacker, D. Richard Kuhn
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{raghu.kacker, kuhn}@nist.gov

***Abstract*—We show that the vertical growth phase of IPOG is optimal for *t*-way test generation when *t* = 2, but it is no longer optimal when *t* is greater than 2. We present an improvement that reduces the number of tests generated during vertical growth. The vertical growth problem is modeled as a classical NP-hard problem called "Minimum Vertex Coloring". We adopted a greedy coloring algorithm to determine the order in which missing tuples are covered during vertical growth. We implemented a revised IPOG algorithm incorporating this improvement. The experimental results show that compared with the original IPOG algorithm, which uses an arbitrary order to cover missing tuples during vertical growth, the revised IPOG algorithm reduces the number of tests for many real-life systems.**

Keywords*—Combinatorial testing; Multi-way test generation; ACTS; Minimum vertex coloring; Tuple ordering;

I. Introduction

In our earlier work, we developed a *t*-way test generation algorithm called In-Parameter-Order-General (IPOG) [3-5]. The IPOG algorithm is a generalization of a pairwise or 2-way test generation algorithm called In-Parameter-Order (IPO) [1, 2]. For a system with *t* or more parameters, the IPOG algorithm builds a *t*-way test set for the first *t* parameters, extends the test set to build a *t*-way test set for the first *t+1* parameters, and then continues to extend the test set until it builds a *t*-way test set for all the parameters.

The extension of an existing *t*-way test set for an additional parameter is done in two phases:

- Horizontal growth, which extends each existing test by adding one value for the new parameter;
- Vertical growth, which adds new tests, if needed, to cover the remaining tuples that have not been covered yet.

A careful examination of the IPOG algorithm reveals that the vertical growth phase is optimal for *t*-way test generation when $t = 2$, but it is no longer optimal when $t > 2$. This is partly because the IPO algorithm was originally designed for 2-way testing. When the IPO algorithm was generalized, no effort was made to optimize the vertical growth phase, and a straightforward extension was adopted.

In this paper, we present an improvement that reduces the number of tests generated during vertical growth. The vertical growth problem is modeled as a classical NP-hard problem called "Minimum Vertex Coloring" [11, 12]. We adopt a greedy coloring algorithm to determine the order in which missing tuples are covered during vertical growth. This is in contrast with the current algorithm where tuples are covered in an arbitrary order, i.e., as they are encountered, during vertical growth. In this paper we focus on *t*-way test generation without constraints, leaving constraint handling as part of our future work.

We implemented a revised IPOG algorithm that incorporates the above improvement in ACTS [6, 7]. We conducted experiments on a set of real-life systems that have been used to evaluate the effectiveness of *t*-way test generation algorithms. The experimental results show that the revised algorithm performed better than the original IPOG algorithm implemented in ACTS, and better than PICT [8, 9], for a set of real-life systems.

The remainder of the paper is organized as follows. Section II briefly reviews the IPOG algorithm for *t*-way testing. Section III describes two motivating examples to show why the vertical growth phase of IPOG for *t*-way test generation when $t > 2$ is not optimized, while it is optimal for 2-way test generation. Section IV presents the improvement based on a graph coloring scheme. Section V reports the design and the results of the experiments. Section VI discusses related work improving IPOG. Section VII provides concluding remarks and our plan for future work.

II. The IPOG Algorithm

In this section, we present the major steps of IPOG algorithm, as shown in Fig. 1. Refer to [3] for more details.

Assume that we already covered the first *k* parameters. To cover the $(k+1)$-th parameter, say *p*, it is sufficient to cover all the *t*-way combinations (also known as tuples) involving parameter *p* and any group of $(t-1)$ parameters among the first *k* parameters. These combinations are covered in two steps, horizontal growth and vertical growth. Horizontal growth adds a value of *p* to each existing test. Each value is chosen such that it covers the most uncovered combinations. During vertical growth, the remaining combinations are covered one at a time, either by changing an existing test or by adding a new test. When we add a new test to cover a combination, parameters that are not involved in the combination are given a special

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
4th International Workshop on Combinatorial Testing (IWCT 2015)
978-1-4799-1885-0/15/$31.00 ©2015 IEEE

value called "don't care". These "don't care" values can be later changed to cover other combinations.

```
Algorithm IPOG(int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set
2. sort the parameters in set ps in a non-increasing order of their
      domain sizes, and denote them as $P_1, P_2, ..., $ and $P_n$
3. add into test set ts a test for each combination of values of the first
      t parameters
4. for (int $i = t + 1$; $i \le n$; $i$ ++){
5.    let $\pi$ be the set of all t-way combinations of values involving
         parameter $Pi$, and any group of $(t - 1)$ parameters among the
         first $i - 1$ parameters
6.    // horizontal extension for parameter $P_i$
7.    for (each test $\tau = (v_1, v_2, ..., v_{i-1})$ in test set ts) {
8.       choose a value $vi$ of $Pi$ and replace $\tau$ with $\tau' = (v_1, v_2, ..., v_{i-1},$
            $v_i)$ so that $\tau'$ covers the most number of combinations of
            values in $\pi$
9.       remove from $\pi$ the combinations of values covered by $\tau'$
10.   } // end for at line 7
11.   // vertical extension for parameter $P_i$
12.   for (each combination $\sigma$ in set $\pi$){
13.      if (there exists a test $\tau$ in test set ts that can be changed to a test
            $\tau'$ that covers both $\tau$ and $\sigma$ , i.e., $\tau$ is compatible with $\sigma$) {
14.         replace test $\tau$ with $\tau'$ in ts
15.         remove from $\pi$ the combinations of values covered by $\tau'$
16.      } else {
17.         add a new test $\tau$ only contains $\sigma$ into ts
18.         remove from $\pi$ the combinations of values covered by $\tau$
19.      } // end if at line 13
20.   } // end for at line 12
21. } // end for at line 4
22. return ts;
}
```

Fig. 1. The original version of the IPOG algorithm

When we use a test to cover a combination, only "don't care" values can be changed. A "don't care" value is a value that can be replaced by any value without affecting the coverage of a test set. If no existing test can be changed to cover a combination σ, a new test needs to be added in which the parameters involved in σ are assigned the same value as in σ and the other parameters are assigned "don't care" values.

The new parameter, i.e. the parameter to be covered by the current extension of the existing test set, is assigned the "don't care" value in a test when choosing any possible value of this new parameter does not cover any new combination in the test. This only happens during horizontal growth. The old parameters, i.e. the parameters that have already been covered by the existing test set, are assigned the "don't care" value during vertical growth when adding a new test to cover a combination. Some of these "don't care" values will be changed to a specific value at a later point of vertical growth for the same parameter whereas others may survive to the extension for the next parameter to be covered.

Thus, there may exist three types of test with "don't care" values in the existing test set after horizontal growth:

1. *Tests only have "don't care" value for the new parameter:* No more (uncovered) combinations can be covered, since horizontal growth has tried all possible extensions for this test using all possible values of the new parameter.
2. *Tests only have "don't care" values for the old parameters:* Though the value of the new parameter has been determined, there may still exist some possible extensions, since horizontal growth does not consider all possible values of the old parameters.
3. *Tests have "don't care" values for the new parameter and the old parameters:* Choosing any value of the new parameter would not cover any more uncovered combinations. But there may exist some possible extensions for the "don't care" values of the old parameters and the new parameter combined together during vertical growth.

Types 2 and 3 indicate opportunities where uncovered combinations can be covered by existing tests, i.e., without adding new tests. The challenge is that, for a general system, if there is a systematic strategy for changing "don't care" values to reduce the number of tests as much as possible.

III. MOTIVATING EXAMPLES

Assume that the horizontal growth phase of IPOG has been finished for a given system, and the vertical growth phase is about to begin. Since the new parameter is involved in every missing tuple, we can divide all missing tuples into different groups such that all the tuples in the same group involve the same value of the new parameter. Doing so allows us to divide the vertical growth problem into multiple independent sub-problems, each of which tries to generate tests to cover one group of missing tuples. Note that, missing tuples in different groups must be covered with different tests. In this respect, there exists no interaction between missing tuples in different groups.

A. Example for 3-way test generation

We present an example to show why the vertical growth of IPOG needs to be improved for the *t*-way generation when *t* is greater than two: Assume that we are in the vertical growth phase of a 3-way test generation process. Let *d* be the new parameter being covered. Assume that the missing tuples involving a value of *d* denoted as *d*.1 are t1 = {*a*.2, *b*.0, *d*.1}, t2 = {*b*.1, *c*.0, *d*.1}, t3 = {*a*.2, *c*.0, *d*.1}, t4 = {*b*.0, *c*.1, *d*.1}. Fig. 2 shows the conflict graph of this example; vertices are missing tuples and edges are conflicts. Two tuples have a conflict if and only if they cannot be covered by the same test. This happens when there exists at least one parameter that appears in both tuples but have different values in these two tuples.

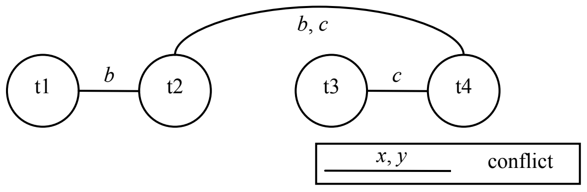

Fig. 2. Conflict graph for the example 3-way test generation process

If we try to cover these missing tuples in a default order (t1, t2, t3, t4), we get 3 tests as shown in TABLE I. ("*" represents "don't care" value.)

TABLE I. A LESS OPTIMAL TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**			
	a	***b***	***c***	***d***
t1, t3	2	0	0	1
t2	*	1	0	1
t4	*	0	1	1

However, if the tuples are covered in the following order (t2, t4, t1, t3), only 2 tests are needed to cover all of them, as shown in TABLE II.

TABLE II. AN OPTIMAL TEST SET FOR THE EXAMPLE 3-WAY TEST GENERATION PROCESS

Covered Tuples	**Test**			
	a	***b***	***c***	***d***
t2, t3	2	1	0	1
t4, t1	2	0	1	1

The problem of how to use the minimum number of tests to cover a set of missing tuples can be modeled as the "Minimum Vertex Coloring (MVC)" problem in the conflict graph of these tuples.

A vertex coloring is an assignment of labels or colors to each vertex of a graph such that no edge connects two identically colored vertices. The MVC problem seeks to minimize the number of colors for a given graph. Such a coloring is referred to as a minimum vertex coloring, and the minimum number of colors with which the vertices of a graph G may be colored is called the chromatic number [10]. The MVC problem is a classical NP-hard optimization problem in computer science, and is typically solved using a greedy algorithm, e.g., greedy coloring [13, 14].

A greedy coloring colors of the vertices of a graph in a greedy manner. Specifically, it considers the vertices of the graph in sequence and assigns each vertex its first available color. Greedy colorings may not always result in the minimum number of colors. In particular, the order in which the vertices are covered has a significant impact on the result.

A commonly used ordering for greedy coloring is to choose a vertex of minimum degree, order the remaining vertices, and then place this vertex last in the ordering, which is equivalent to a non-increasing order of the degree of the vertices. If every sub-graph of a graph G contains a vertex of degree at most *d*, then the greedy coloring using this ordering will use at most *d+1* colors [15, 16].

B. Example for 2-way test generation

We use an example to show why the vertical growth of IPOG is optimal for pairwise: Assume that we are in the vertical growth phase of a 2-way test generation process. Let *d* be the new parameter being covered. Assume that the missing tuples involving a value of *d* denoted as *d*.1 are t1 = {*a*.0, *d*.1}, t2 = {*a*.1, *d*.1}, t3 = {*a*.2, *d*.1}, t4 = {*b*.0, *d*.1}, t5 = {*b*.1, *d*.1}, t6 = {*c*.1, *d*.1}. Fig. 3 shows the conflict graph of this example that consists of three connected components, each of which is a complete graph with conflicts involving only one old parameter.

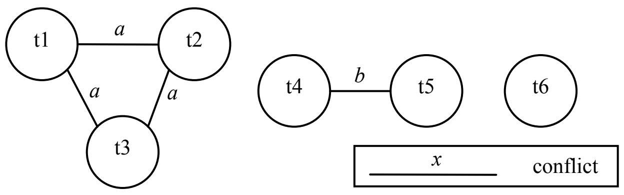

Fig. 3. Conflict graph for the example 2-way test generation process

Any arbitrary order is optimal in the 2-way test generation, such as (t1, t2, t3, t4, t5, t6) or (t1, t4, t6, t2, t5, t3) whose test set size is 3. The reason can be presented as follows: For the largest complete sub-graph with degree *d*, it needs at least *d+1* colors to separately color its *d+1* vertices. While *d+1* colors are enough to color any complete sub-graph, these sub-graphs in the conflict graph for 2-way test generation are all unconnected, *d+1* colors are enough to color the whole conflict graph. In this case, there is only one result for greedy coloring with any order, *d+1* colors, which is optimal. This explains why the original IPOG algorithm is optimal for 2-way test generation, even though it uses an arbitrary order to cover missing tuples during vertical growth.

IV. APPROACH FOR IPOG IMPROVEMENT

In Section III, we only discussed the case where new tests are created to cover missing tuples, ignoring the case where existing tests resulted from horizontal growth can also be changed to cover missing tuples. If all the existing tests with "don't care" values are of type 1 (see Section II) and thus cannot cover any more missing tuples, no additional action is needed; otherwise, it is important to properly represent the existing tests as colored vertices in the conflict graph with separate colors, and add edges to represent conflicts between missing tuples and existing tests. Note that in this case, the vertical growth phase of IPOG may no longer be optimal even for 2-way test generation, since the conflict graph including existing tests becomes more complex such that the arbitrary ordering of tuples does not necessarily produce the optimal result.

A. Conflict graph with existing tests

1) Graph model with type 2 existing tests

For an existing test in which only old parameters have the "don't care" value (called type 2 in Section II), the new parameter is assigned a specific value. Thus, it can only cover missing tuples that have the same value for the new parameter. We refer to these missing tuples as missing tuples relevant to this test. We represent the existing test as a colored vertex, and add edges to represent the conflicts between this test and missing tuples that are relevant to this test. Note that the colors of these colored vertices as existing tests are all different.

For example, let *f* be the new parameter being covered. Given two existing tests of type 2 as colored vertices T1 = {*a*.*, *b*.0, *c*.0, *d*.0, *e*.*, *f*.0}, T2 = {*a*.0, *b*.*, *c*.0, *d*.*, *e*.0, *f*.0}, the specific value of the new parameter is *f*.0. Assume that all the missing tuples involving *f*.0 are t1 = {*a*.0, *d*.0, *f*.0}, t2 = {*a*.0, *d*.1, *f*.0}, t3 = {*b*.0, *e*.0, *f*.0}, t4 = {*b*.0, *e*.1, *f*.0}, which are relevant to T1 and T2. Fig. 4 shows the conflict graph. (Dash lined circle means this vertex has already been colored as an existing test.)

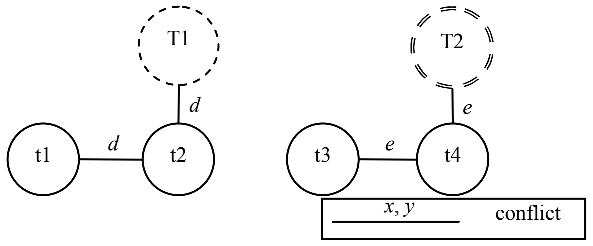

Fig. 4. Conflict graph with two type 2 existing tests

According to the conflict graph in Fig. 4, we can get an optimal tuple order (t2, t4, t1, t3), in the order of non-increasing degrees. The degree of a tuple is the number of the conflicts it involves. Following this order allows the tuples to be covered by the two existing tests, i.e., T1 and T2, without adding a new test, as shown in TABLE III.

TABLE III. AN OPTIMAL TEST SET GENERATED FROM THE CONFLICT GRAPH IN FIG. 4

Existing Test	Covered Tuples	Extended Test					
		a	***b***	***c***	***d***	***e***	***f***
T1	t4, t1	0	0	0	0	1	0
T2	t2, t3	0	0	0	1	0	0

If we do not capture the conflicts between missing tuples and existing tests, i.e., the two conflicts between T1 and t2, T2 and t4, an insufficient conflict graph would be obtained as shown in Fig. 5.

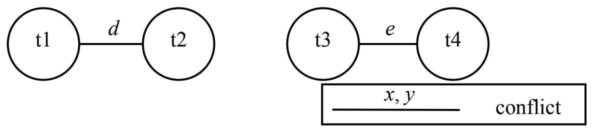

Fig. 5. An insufficient conflict graph with two type 2 existing tests

If this insufficient conflict graph is used, the tuple order based on the non-increasing order of degrees would be (t1, t2, t3, t4), which would generate a third test, in addition to existing tests T1 and T2, in order to cover all these tuples, as shown in TABLE IV.

TABLE IV. A LESS OPTIMAL TEST SET GENERATED FROM THE INSUFFICIENT CONFLICT GRAPH IN FIG. 5

Existing Test	Covered Tuples	Extended Test					
		a	***b***	***c***	***d***	***e***	***f***
T1	t1, t3	0	0	0	0	0	0
T2	t2	0	*	0	1	0	0
	t4	*	0	*	*	1	0

This example shows that it is important to capture the conflicts between existing tests and missing tuples.

2) Graph model with type 3 existing tests

For an existing test in which old parameters and the new parameter have the "don't care" value, called type 3 in Section II, the value of the new parameter can still be changed. A type 3 existing test provides opportunities for missing tuples involving any possible value of the new parameter to be covered, which leads to competition among missing tuples not only in the same group but also in different groups. Recall that missing tuples involving the same value of the new parameter are grouped together. This makes different groups no longer independent. In this respect, if there exists a type 3 test, missing tuples do not need to be grouped.

We represent missing tuples as uncolored vertices, and the existing tests as colored vertices with separate colors. Recall that two tuples have a conflict if they cannot be covered by the same test. One tuple and one existing test have a conflict if the tuple cannot be covered by changing the existing test. Note that, there may be conflicts due to different values of the new parameter since tuples are no longer grouped.

For example, let *f* be the new parameter being covered. Given a type 3 test as a colored vertex T1 = {*a*.0, *b*.0, *c*.0, *d*.*, *e*.*, *f*.*}, assume that all the missing tuples are t1 = {*c*.1, *d*.0, *f*.0}, t2 = {*d*.0, *e*.1, *f*.0}, t3 = {*d*.0, *e*.1, *f*.1}. Fig. 6 shows the conflict graph.

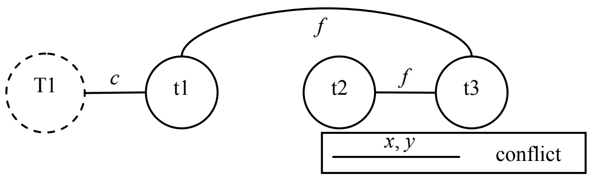

Fig. 6. Conflict graph with a type 3 existing test

An optimal tuple order (t1, t3, t2) is derived from this conflict graph, which allows all the tuples to be covered by two tests, as shown in TABLE V.

TABLE V. AN OPTIMAL TEST SET GENERATED FROM THE CONFLICT GRAPH IN FIG. 6

Existing Test	Covered Tuples	Extended Test					
		a	***b***	***c***	***d***	***e***	***f***
T1	t3	0	0	0	0	1	1
	t1, t2	*	*	1	0	1	0

If we do not represent the conflicts between missing tuples involving different values of the new parameter *f*, i.e., the two conflicts between t1 and t3, t2 and t3, we would obtain an insufficient conflict graph as shown in Fig. 7.

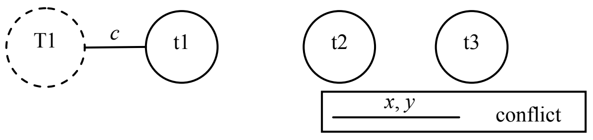

Fig. 7. An insufficient conflict graph with a type 3 existing test

If this insufficient conflict graph is used, the tuple order based on the non-increasing order of degrees would be (t1, t2, t3), which would generate one more test as shown in TABLE VI.

TABLE VI. A LESS OPTIMAL TEST SET GENERATED FROM THE INSUFFICIENT CONFLICT GRAPH IN FIG. 7

Existing Test	Covered Tuples	Extended Test					
		a	***b***	***c***	***d***	***e***	***f***
T1	t2	0	0	0	0	1	0
	t1	*	*	1	0	*	0
	t3	*	*	*	0	1	1

It indicates that, if there exist type 3 tests, the conflicts between missing tuples involving different values of the new parameter become important.

3) Graph model with type 2 and 3 existing tests

If there exist tests of type 2 and 3, the conflicts between missing tuples and type 2 existing tests involving different values of the new parameter can also be proved to be important.

For example, let *f* be the new parameter being covered. Given a type 3 test as a colored vertex T1 = {*a*.0, *b*.0, *c*.0, *d*.*, *e*.*, *f*.*}, and a type 2 test as a colored vertex T2 = {*a*.1, *b*.0, *c*.0, *d*.*, *e*.*, *f*.0}, assume that all the missing tuples are t1 = {*c*.0, *d*.0, *f*.0}, t2 = {*d*.0, *e*.1, *f*.1}. Fig. 8 shows the conflict graph.

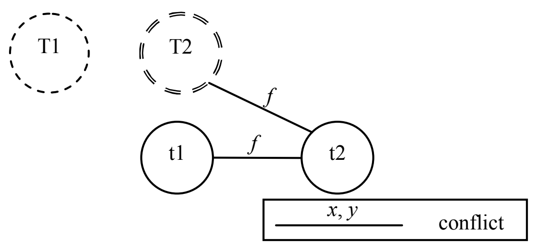

Fig. 8. Conflict graph with a type 3 existing test and a type 2 test

An optimal tuple order (t2, t1) is derived from this conflict graph, which allows all the tuples to be covered by two tests, as shown in TABLE VII.

TABLE VII. AN OPTIMAL TEST SET GENERATED FROM THE CONFLICT GRAPH IN FIG. 8

Existing Test	Covered Tuples	Extended Test					
		a	***b***	***c***	***d***	***e***	***f***
T1	t2	0	0	0	0	1	1
T2	t1	1	0	0	0	*	0

If we do not represent the conflict between the missing tuple and the type 2 test involving different values of the new parameter *f*, i.e., the conflict between t2 and T2, we would obtain an insufficient conflict graph as shown in Fig. 9.

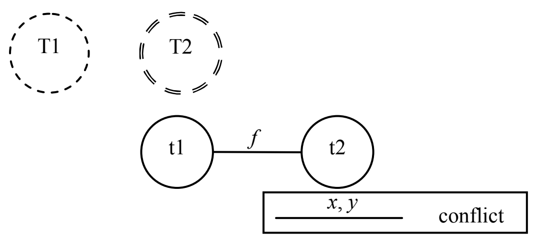

Fig. 9. An insufficient conflict graph with a type 3 test and a type 2 test

If this insufficient conflict graph is used, the tuple order based on the non-increasing order of degrees would be (t1, t2), which would generate one more test as shown in TABLE VIII.

TABLE VIII. A LESS OPTIMAL TEST SET GENERATED FROM THE INSUFFICIENT GRAPH IN FIG. 9

Existing Test	Covered Tuples	Extended Test					
		a	***b***	***c***	***d***	***e***	***f***
T1	t1	0	0	0	0	*	0
T2		1	0	0	*	*	0
	t2	*	*	0	*	1	1

This scenario only happens when a type 3 test is in front of a type 2 test, which not often occurs during the horizontal growth phase of IPOG, but still there is a chance.

In summary, if there are existing tests of type 2 or 3, we must capture conflicts between missing tuples and existing tests in the conflict graph. Also, there may exist more conflicts for tests of type 3, as missing tuples cannot be grouped in this case.

B. Implementation

The vertical growth phase (Fig. 1 lines 12-19) of our original IPOG is already a greedy algorithm that covers each tuple in the first test encountered that could cover this tuple, which is similar to greedy coloring. So, the work needed to implement the proposed improvement is to build the conflict graph for missing tuples and existing tests, and then to derive the tuple order from the conflict graph. Note that once a tuple is covered in a test, it may also cover other missing tuples which should be marked as covered.

First, we use a matrix to represent the conflict graph consisting of missing tuples, existing tests and their conflicts. For instance, assume that the new parameter is *d* and there are missing tuples t1 = {*a*.0, *b*.1, *d*.0}, t2 = {*b*.1, *c*.1, *d*.0}, t3 = {*a*.0, *b*.1, *d*.1}, t4 = {*a*.1, *c*.1, *d*.1} and existing tests T1 = {*a*.1, *b*.0, *c*.*, *d*.0}, T2 = {*a*.0, *b*.*, *c*.0, *d*.1}, T3 = {*a*.1, *b*.1, *c*.1, *d*.*}. We create a conflict matrix, which consists of an (upper triangular) matrix that capture conflicts between missing tuples, and a matrix that capture conflicts between missing tuples and existing tuples, shown as TABLE IX.

TABLE IX. AN EXAMPLE CONFLICT MATRIX

		***d*.0**		***d*.1**		***d*.0**	***d*.1**	***d*.***
		t1	**t2**	**t3**	**t4**	**T1**	**T2**	**T3**
***d*.0**	**t1**	F	F	T	T	T	T	T
	t2		F	T	T	T	T	F
***d*.1**	**t3**			F	T	T	F	T
	t4				F	T	T	F

In the above conflict matrix, only the values in the sub-matrices in bold-lined boxes need to be determined by comparing the values of the parameters that appear in both tuples indicated by the row and column indices. All other values in the rest of the matrix must be true, since combinations with different values of the new parameter always conflict with each other. We use this observation to reduce the space requirement for storing conflict matrix. The reduction can be significant when the number of missing tuples *m* is large and/or domain size *s* is big.

According to the above conflict matrix, we can easily find the degree of each tuple by counting the conflict flags in its involved row and column, such as degree(t1) = 5, degree(t2) = 4, degree(t3) = 5, degree(t4) = 5.

The order of tuples to be covered can be determined as follows. We first choose a missing tuple of maximum degree in the conflict graph, and then cover it in its first compatible test by searching in the extending test set from top to bottom. In this example, the order of tuples is determined to be (t1, t3, t4, t2), which can be covered by four tests as shown in TABLE X.

TABLE X. AN OPTIMAL TEST SET GENERATED FROM AN OPTIMAL ORDER

Existing Test	Covered Tuples	Extended Test			
		a	*b*	*c*	*d*
T1		1	0	*	0
T2	t3	0	1	0	1
T3	t4	1	1	1	1
	t1, t2	0	1	1	0

V. EXPERIMENT

We implemented in ACTS a revised IPOG algorithm that incorporates the Graph Coloring-based (GC) approach to vertical growth. We report several experiments that were conducted to evaluate the effectiveness of the revised IPOG algorithm. In particular, we conducted an experiment that compared ACTS with an existing tool PICT [8]. The experimental results show that the revised IPOG algorithm performed better than the original IPOG algorithm in ACTS and PICT for a set of real-life systems.

A. *Experiment design*

TABLE XI shows nine real-life systems used in our experiments. These systems have been packaged as example SUTs in ACTS releases for several years.

TABLE XI. MODELS AND SOURCES

Software Name	SUT File Name	Source
Apache HTTP server	apache.xml	http://httpd.apache.org
Berkeley DB	Berkeley.xml	https://oss.oracle.com/berkeley-db.html
Bugzilla bug tracker	bugzilla.xml	https://bugzilla.mozilla.org
GCC compiler	gcc.xml	https://gcc.gnu.org
replace c program	replace.xml	SIR repository, http://sir.unl.edu
SPIN simulator	Spin_S.xml	http://spinroot.com
SPIN verifier	Spin_V.xml	http://spinroot.com
tcas c program	tcas.xml	SIR repository, http://sir.unl.edu
Violet UML editor	Violet.xml	http://sourceforge.net/projects/violet

We adopt the exponential notation to denote parameter configurations, where d^n means that there are *n* parameters of domain size *d*. The configurations of these real-life systems are shown in TABLE XII. (Recall that we do not handle constraints in this paper.)

TABLE XII. CONFIGURATIONS OF REAL-LIFE SYSTEMS

Name	Num. of Parameters	Parameter Configuration
apache	172	$2^{158}\ 3^{8}\ 4^{4}\ 5^{1}\ 6^{1}$
Berkeley	78	2^{78}
bugzilla	52	$2^{49}3^{1}4^{2}$
gcc	199	$2^{189}3^{10}$
replace	20	$2^{4}4^{16}$
Spin_S	18	$2^{13}4^{5}$
Spin_V	55	$2^{42}3^{2}4^{11}$
tcas	12	$2^{7}\ 3^{2}\ 4^{1}\ 10^{2}$
Violet	101	2^{101}

B. *Results and analysis*

The experimental environment is set up as the following: OS: Windows 7 64bits, CPU: Intel Dual-Core i5 2.5GHz, Memory: 4 GB DDR3, Java SDK (Since ACTS is a Java tool): Java SE 1.6, Java Max Heap Size: 1024MB.

We generate 4-way, 3-way and 2-way test sets for the nine subject systems, by using PICT 3.3, ACTS 2.92, and ACTS 2.92-GC. Experimental results are shown in TABLE XIII, TABLE XIV and TABLE XV. Note that cells highlighted by gray background indicate the smallest test set sizes produced by the three tools.

TABLE XIII. RESULT OF 4-WAY TEST GENERATION

Name	# of Tuples	PICT 3.3		ACTS 2.92 original IPOG		ACTS 2.92-GC revised IPOG	
		size	*Time(s)*	*size*	*Time(s)*	*size*	*Time(s)*
apache	728304446	N/A	Crash	834	1456.944	828	1523.073
Berkeley	22822800	121	188.168	120	20.178	119	20.312
bugzilla	5204192	220	64.609	230	3.886	227	4.147
gcc	N/A	N/A	Crash	N/A	OOM	N/A	OOM
replace	800784	1062	23.38	987	0.497	970	62.263
Spin_S	125040	353	1.364	343	0.079	341	0.137
Spin_V	11873396	808	430.535	779	12.336	749	57.495
tcas	64696	1410	2.274	1359	0.07	1358	0.443
Violet	65326800	131	588.096	131	67.896	132	71.572

OOM = Out of Memory Java exception

TABLE XIV. RESULT OF 3-WAY TEST GENERATION

Name	# of Tuples	PICT 3.3		ACTS 2.92 original IPOG		ACTS 2.92-GC revised IPOG	
		size	*Time(s)*	*size*	*Time(s)*	*size*	*Time(s)*
apache	8087048	202	132.912	173	10.007	170	10.315
Berkeley	608608	45	1.831	46	0.32	44	0.543
bugzilla	203104	68	0.648	67	0.131	66	0.21
gcc	11147562	88	89.99	78	17.238	76	18.582
replace	52768	203	0.343	181	0.058	181	0.58
Spin_S	13328	96	0.071	79	0.036	80	0.094
Spin_V	377128	168	2.3	159	0.227	156	0.758
tcas	9158	402	0.124	400	0.045	400	0.038
Violet	1333200	47	3.77	48	0.826	48	1.114

TABLE XIII shows that for 4-way test generation, compared with the original IPOG implementation, the revised IPOG algorithm reduces test set size for seven of the nine real-life systems, and slightly increases test set size for one of the nine systems. Note that for gcc, none of the three tools can generate 4-way test sets due to limited memory space.

TABLE XIV shows that for 3-way test generation, compared with the original IPOG algorithm, the revised IPOG algorithm reduces test set size for five of the nine real-life systems, and slightly increases test set size for one of the nine systems. Both algorithms produce the same test set size for the other three systems.

These results show that our new vertical growth algorithm is effective for *t*-way test generation when *t* is greater than two.

TABLE XV. RESULT OF 2-WAY TEST GENERATION

Name	**# of Tuples**	**PICT 3.3**		**ACTS 2.92 original IPOG**		**ACTS 2.92-GC revised IPOG**	
		size	***Time(s)***	***size***	***Time(s)***	***size***	***Time(s)***
apache	66930	39	0.204	33	0.112	33	0.123
Berkeley	12012	15	0.052	16	0.055	16	0.077
bugzilla	5822	20	0.035	18	0.038	18	0.044
gcc	82809	21	0.133	20	0.148	20	0.173
replace	2456	38	0.044	38	0.026	37	0.061
Spin_S	992	23	0.033	24	0.025	24	0.03
Spin_V	8797	32	0.047	33	0.046	34	0.067
tcas	837	100	0.038	100	0.037	100	0.032
Violet	20200	16	0.053	16	0.08	16	0.095

TABLE XV shows that for 2-way test generation, compared with the original IPOG algorithm, the revised IPOG algorithm only slightly reduces test set size for one of the nine real-life systems, and slightly increases test set size for one of the nine systems. Both algorithms produce the same test set size for the other seven systems. This is expected, as the vertical growth phase in the original IPOG algorithm is optimal for 2-way testing.

VI. RELATED WORK

In our past work, we presented three variants of the IPO algorithm: IPOG [3], IPOG-F/IPOG-F2 [17] and IPOG-D [4]. IPOG implements the generalization to *t*-way testing of the original IPO algorithm. This algorithm explicitly enumerates all possible combinations, and does not scale well to big systems, where the number of combinations is large, or when resources are limited. IPOG-F is a variant of IPOG whose implementation has been optimized for speed, which also achieves better test set size than IPOG for some systems, but gets worse for some other systems [18]. IPOG-D is a variant of IPOG, incorporating a recursive technique, namely the doubling-construct, and developed just to address the problem of reducing the number of combinations that have to be enumerated, so to improve the algorithm scalability.

Younis et al. [19-21] presented MIPOG which is a modification of the IPOG algorithm for *t*-way testing. The differences between the MIPOG and IPOG algorithms lie in both horizontal and vertical extensions. In horizontal extension, the MIPOG algorithm checks all the values of the input parameter and chooses the value that contains the maximum number of uncovered tuples. Also, it optimizes "don't care" values. In vertical extension, MIPOG reorders the set of missing tuples in the decreasing order of the size of the remaining tuples. After that, it chooses the first tuple from the missing set and combines that tuple with others missing tuples (i.e. the resulting test case must have the maximum weight of the uncovered tuples). Their results show that the MIPOG algorithm will always give the same or less test set than IPOG. MIPOG uses a different approach than our work. Also, it does not provide a public tool, which prevents an experimental comparison between the two approaches.

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented an improvement on the vertical growth phase of the IPOG algorithm. In this algorithm we create a graph model that captures the conflict relationship among different tuples and existing tests. We reduce the vertical growth problem to a classical NP-hard problem "Minimum Vertex Coloring" on this graph. A greedy coloring algorithm is used to determine the order in which missing tuples can be covered. Specifically, the higher the degree of a missing tuple in the conflict graph, the earlier it should be covered. The experimental results show that the improvement can further reduce the number of tests that are generated by the IPOG algorithm for a set of real-life systems.

In the future, we will consider the impact of constraints on vertical growth. Constraints will create more conflicts between tuples. In addition, constraints may introduce conflicts that involve more than two tuples. We will explore how to represent such higher-degree conflicts and how to consider them during vertical growth.

ACKNOWLEDGMENT

This work is partly supported by a research grant (70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

DISCLAIMER: NIST does not endorse or recommend any commercial product referenced in this paper or imply that a referenced product is necessarily the best available for the purpose.

REFERENCES

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

[2] K-C Tai, and Y. Lei. "A test generation strategy for pairwise testing." IEEE Transactions on Software Engineering 28, no. 1 (2002): 109-111.

[3] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. "IPOG: A general strategy for t-way software testing." In Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the, pp. 549-556. IEEE, 2007.

[4] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. "IPOG/IPOG‐D: efficient test generation for multi‐way combinatorial testing." Software Testing, Verification and Reliability 18, no. 3 (2008): 125-148.

[5] L. Yu, Y. Lei, M. Nourozborazjany, R.N. Kacker, and D. R. Kuhn. "An efficient algorithm for constraint handling in combinatorial test generation." In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pp. 242-251. IEEE, 2013.

[6] ACTS (Advanced Combinatorial Testing System), http://csrc.nist.gov/acts/

[7] L. Yu, Y. Lei, R.N. Kacker, and D.R. Kuhn. "Acts: A combinatorial test generation tool." In Software Testing, Verification and Validation

(ICST), 2013 IEEE Sixth International Conference on, pp. 370-375. IEEE, 2013.

[8] PICT (Pairwise Independent Combinatorial Testing tool), http://blogs.msdn.com/b/nagasatish/archive/2006/11/30/pairwise-testing-pict-tool.aspx

[9] J. Czerwonka. “Pairwise testing in the real world: Practical extensions to test-case scenarios.” In Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer, pp. 419-430. 2006.

[10] N. Christofides. “An algorithm for the chromatic number of a graph.” The Computer Journal 14, no. 1 (1971): 38-39.

[11] D.W. Matula, G. Marble, and J.D. Isaacson. “Graph coloring algorithms.” Graph theory and computing (1972): 109-122.

[12] S. Skiena. “Finding a Vertex Coloring.” §5.5.3 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 214-215, 1990.

[13] B. Manvel. “Extremely greedy coloring algorithms.” Graphs and Applications (1985): 257-270.

[14] L. Kučera. “The greedy coloring is a bad probabilistic algorithm.”d Journal of Algorithms 12, no. 4 (1991): 674-684.

[15] D.J. Welsh, and M.B. Powell. “An upper bound for the chromatic number of a graph and its application to timetabling problems.” The Computer Journal 10, no. 1 (1967): 85-86.

[16] M.M. Sysło. “Sequential coloring versus Welsh-Powell bound.” Annals of Discrete Mathematics 39 (1989): 241-243.

[17] M. Forbes, J. Lawrence, Y. Lei, R.N. Kacker, and D.R. Kuhn. “Refining the in-parameter-order strategy for constructing covering arrays.” Journal of Research of the National Institute of Standards and Technology 113, no. 5 (2008): 287-297.

[18] A. Calvagna, and A. Gargantini. “T - wise combinatorial interaction test suites construction based on coverage inheritance.” Software Testing, Verification and Reliability 22, no. 7 (2012): 507-526.

[19] M.I. Younis, K.Z. Zamli, and N.M. Isa. “MIPOG-Modification of the IPOG strategy for T-Way software testing.” Proceeding of the Distributed Frameworks and Applications (DFmA) (2008).

[20] M.I. Younis, K.Z. Zamli, and N.A.M. Isa. “A strategy for grid based t-way test data generation.” In Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on, pp. 73-78. IEEE, 2008.

[21] M.I. Younis, and K.Z. Zamli. “MIPOG-An Efficient t-Way Minimization Strategy for Combinatorial Testing.” Int. J. Comput. Theory Eng 3, no. 3 (2011): 388-397.