ตารางการจัดการแข่งขันแบบพบกันหมด และการสร้างกรณีทดสอบซอฟต์แวร์
Round-Robin Tournament Scheduling and Test Case Generating

อำพล ธรรมเจริญ*
ภาควิชาคณิตศาสตร์ คณะวิทยาศาสตร์ มหาวิทยาลัยบูรพา
Ampon Dhamacharoen*
Department of Mathematics, Faculty of Science, Burapha University

บทคัดย่อ

บทความนี้อธิบายวิธีการสร้างตารางการจัดการแข่งขันแบบพบกันหมดและการนำมาใช้ในการสร้างกรณีทดสอบซอฟต์แวร์แบบที่ผู้เขียนเสนอ ซึ่งเป็นวิธีที่สอดคล้องกับข้อกำหนดว่าทุกๆ ค่าจะต้องอยู่ในกรณีทดสอบ 2 กรณี นอกจากนี้จะกล่าวถึงความเชื่อมต่อระหว่างการสร้างกรณีทดสอบแบบคู่ กับตารางการจัดแบบจตุรัสสังคม

คำสำคัญ : ตารางการจัดการแข่งขันแบบพบกันหมด จตุรัสสังคม การสร้างกรณีทดสอบ การเป็นคู่ การจัดหมู่

Abstract

This article explains how the round robin tournament schedule is constructed and then used to construct the new proposed test suite, which meets the requirement that each value must be in a pair of test cases. The connection of pair-wise test case generation and the social square scheduling is also discussed.

Keywords : round-robin tournament scheduling, social square, Test case generating, pair-wise, combinations.

*E-mail: ampon@buu.ac.th

Round-Robin Tournament

A round-robin tournament, or all-play-all tournament, is a type of group tournament in which each participant plays every other participant an equal number of times. (DeVenezia, 2006; Wikipedia, 2007). In a single round-robin schedule, each participant plays every other participant once. If each participant plays all others twice, this is frequently called a double round-robin. The term round-robin is derived from the French term ruban, meaning "ribbon". Over a long period of time, the term was corrupted and idiomized to robin.

To illustrate: Let us consider a chess tournament which consists of 6 players. Each player has to compete with every other player once. Since there are 2 players in a match, there must be $\binom{6}{2}$ = 15 matches (pair of players). Suppose that each player plays once on each day (which we shall call a "round"). Then there are at most 3 matches in a round, and at least 5 rounds of competitions. Suppose A, B, C, D, E, F are players, the first round may consist of AB, CD, EF (means A plays B, C plays D and E plays F). The second round may be AC, BE and DF. Note that Round 2 cannot be AC, BD since it would E and F matched together again. A complete 5 round schedule may be as follows:

Round 1: AB, CD, EF

Round 2: AC, BE, DF

Round 3: AD, BF, CE

Round 4: AE, BD, CF

Round 5: AF, BC, DE

With a small number of players, the schedule is not very difficult to construct using a trial-and-error method. But, in general, constructing the schedule by utilizing such a method is not an easy task. (To appreciate the problem the reader should try to construct a schedule for 10 players.) However, there is an amazingly easy algorithm that can generate a complete schedule for the tournament. The following schedule algorithm is employed from Wikipedia (Wikipedia, 2007).

If n is the number of competitors, a single round-robin tournament requires $\binom{n}{2} = \frac{1}{2}$ n(n - 1) matches and n - 1 rounds; each round contains $\frac{n}{2}$ matches.

Algorithm 1: (Case: n is even)

Step 1: Write down all players in the 2 x $\frac{n}{2}$ matrix. Suppose the players are 1, 2, 3, . . . , n, we write

1	3	5	. . .	n-3	n-1
2	4	6	. . .	n-2	n

The matrix illustrates the schedule of the first round. Each player in the first row plays the player in the same column in the second row. That is: 1 plays 2, 3 plays 4, . . . and so on. The last column shows that the player n-1 plays n.

Step 2. Fix one competitor (number one in this example) and rotate the others clockwise.

$$\begin{array}{ccccccccccc} (1) & \rightarrow & 3 & \rightarrow & 5 & \rightarrow & \ldots & \rightarrow & n\text{-}3 & \rightarrow & n\text{-}1 \\ & \nearrow & & & & & & & & & \downarrow \\ 2 & \leftarrow & 4 & \leftarrow & 6 & \leftarrow & \ldots & \leftarrow & n\text{-}2 & \leftarrow & n \end{array}$$

We obtain the second round schedule:

1	2	3	5	. . .	n-5	n-3
4	6	8	10	. . .	n	n-1

Continue this process we finally get the complete round-robin schedule.

If n is odd, a dummy player can be added, whose scheduled opponent in a given round does not play and has a bye.

The Social Square

Suppose there are n^2 players. On each day (round) all of these players are grouped into n groups with n players in each group. We want to plan a schedule in such a way that in each round, each player is grouped with all new players; in other words, each one is grouped with each of the other players only once (DeVenezia, 2006).

Since during each round, each player must meet n - 1 players, then there must be n + 1 rounds to complete meeting all n^2 - 1 players.

The social square is considered to be a generalization of the round robin tournament in which each match contains more than two players. Here is an example of the social square with 3^2 players, named 1, 2, . . . , 9.

Round 1:

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

Round 2:

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

Round 3:

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

Round 4:

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

There are three groups in each round. In Round 1, the groups are 147, 258 and 369. (The group 147 contains the player 1, 4 and 7.) Round 2 consists of the group 168, 249 and 357. Round 3 are 159, 267 and 348, and Round 4 are 123, 456 and 789.

There are algorithms for generating the complete schedule for n^2 players where n is a prime number. The schedule for non-prime n is possible, (for n = 4 see DeVenezia, 2006) but the algorithm in general is not known.

Algorithm 2: (Constructing the social square schedule where n is a prime.)

Step 1: For k = 1, 2, . . . , n, construct the n x n matrix whose entries are as follows:

For i = 1, 2, . . . , n; j = 1, 2, . . . , n,

$p = [kn - (i-1)(k-1) + j - 1] \pmod n$

$a_{ij} = n(i - 1) + p + 1,$

For each k the result matrix is the Round k schedule, with each column representing the group. Note that the Round 1 schedule matrix has the entries:

$a_{ij} = n(i - 1) + j.$

Step 2: Transpose the Round 1 matrix to derive the Round n+1 schedule. In other words, the Round n+1 matrix has the entries:

$a_{ij} = n(j - 1) + i.$

The result is the set of n+1 matrices representing the social square schedule.

The proof of validity of the result is not difficult and will be omitted.

Software Testing

Introduction:

Testing and other fault detection activities are crucial to the success of a software project. (Grindal et al., 2004). Before the software is released to the public, it should be tested to ensure that it is fault free. The software test process involves identifying sufficient input test data to validate requirements of a product.

Software generally consists of functions. Each function consists of commands, input data, interface and others, which are called parameters or variables. Each parameter has elements which are called levels or values. (Phadke, 1997)

To illustrate the testing scenario, let us consider an example of testing the send function of a fax machine. The parameters are the followings: A, the time of transmission, its levels being “send it now,” “send it an hour later,” or “send it after midnight”; B, the recipients’ telephone numbers, and its levels are a single telephone number entered now, telephone numbers selected from quick-dial memory, or multiple numbers entered for broadcast transmission; C, the interruption during transmission, and its levels could be no interruption in the telephone line during transmission, line cutoff during transmission, and the telephone line becoming noisy, which requires a drop in the transmission speed, and so on. The software has to perform well for all possible values of these parameters.

The techniques of software testing can be classified into two categories, white box and black box. (Jorgensen, 1995; Phadke, 1997). Black box testing treats the software

as a black-box without any understanding as to how the internals behave. It aims to test the functionality according to the requirements. White box testing, however, is when the tester has access to the internal data structures, codes, and algorithms. The term grey box testing is sometimes used when the designing of the test cases involves having access to internal data structures and algorithms, but testing at the user, or black-box level.

The faults in the software, caused by some values in some parameters, can be detected when the computer with given phenomenon behaves abnormally. (Phadke, 1997; Bolton, 2004). Designing the test case is aimed at detecting all the faults the software has. The simplest kind of fault is that which is caused by a single variable in a single state. This is called a single-mode fault. The double-mode fault is the fault caused by the interaction of two values in different parameters. The triple-mode fault, or more generally, the multi-mode fault, is the fault caused by the interaction of three or more values. It is believed that most faults were of the single-mode fault or the double-mode fault variety. The multiple mode faults are not as likely to occur, and may be neglected in some testing (Bolton, 2004).

Test case generating:

A test case is a set of values from each parameter. In testing we deal with how to construct the test cases, which consists of implementing the parameters and their values to be appropriated for test, and specifying or selecting values for the test case. For example in a registration form, a user has to input his name which is a group of alphabets up to 20 characters long. After some implementation the values of this parameter could be “no input”, a name with 1 character, a name with 20 characters, and a name with 21 characters. The implementation part could be done by the boundary value analysis method or the equivalent class method (Jorgensen, 1995). In this article we focus on the strategy of selecting the test cases for the test suite that achieves the purposes.

Assume that we are given the parameters and their values to be tested. The first thing every one involved in testing wants to do is plan the test suite that can detect all the faults, that is the all coverage test. (Phadke, 1997; Bolton, 2004). This aim leads to the all combination testing, which generally has a huge number of test cases. For example, if the function we want to test has 4 parameters, and each parameter has 3 values, then all combinations testing could have $3^4 = 81$ test cases. Thus all combination testing becomes unfeasible when the number of parameters and values are large.

The pair-wise or all pairs test is designed to detect the single and double mode faults. The test suite must be such that each pair of values from different parameters has to be in at least one test. This is the all pair coverage. The all pair test may not detect the multiple-mode faults, but the risk of having that type of fault is small. (Phadke, 1997; Bolton, 2004). Suppose that the system under testing has k parameters, with n values each, the number of pair-wise test cases is n^2, provided that $k \leq n + 1$. For example as in the above paragraph, n = 3 and k = 4, the number of pair-wise test cases is 9.

Generating the pair-wise test suite is generally a difficult task, but there are good algorithms provided by Maity and Nayak. (Maity & Nayak, 2005)

To detect all the single-mode faults, the test suite must contain all values from all parameters; this may be called 1-wise coverage. Then each choice combination strategy will include each value of each parameter in the test case in the easiest way. Suppose that there are 4 parameters, A, B, C and D, and each parameter has 3 values, 1, 2 and 3. The each-choice combination test suite could be as follows: (Phadke, 1997; Grindal *et al,* 2004)

Test 1 = (A1, B1, C1, D1)

Test 2 = (A2, B2, C2, D2)

Test 3 = (A3, B3, C3, D3)

Note that there are only 3 test cases. It is easily seen that when a value causes the fault, it must be shown in some test cases. The weak part of this strategy is that it cannot identify which value causes the fault.

Another 1-wise coverage method is the base choice combination. It starts by identifying one base test case. From the base test case, new test cases are created by varying the values of one parameter at a time, keeping the values of the other parameters fixed on the base test case. (Phadke, 1997; Grindal *et al*, 2004). For example as in the above paragraph, the base choice combination test suite is as follows: (We write 2111 instead of (A2, B1, C1, D1) for short.)

Test 1 = 1111 (as a base)
Test 2 = 2111
Test 3 = 3111
Test 4 = 1211
Test 5 = 1311
Test 6 = 1121
Test 7 = 1131
Test 8 = 1112
Test 9 = 1113

The advantage of this method is that it can identify which value caused the fault. For example, suppose that all test cases work properly except Test 2. From this, it is possible to derive that the value A2 caused that fault. Note that there are 9 test cases. If the system under test has k parameters, with n values each, the number of base-choice combination test cases is k(n - 1) + 1. As in the example, if we have n = 3 and k = 4, then the number of test cases is 4(3 - 1) + 1 = 9.

Proposed Method and Algorithm

Here we propose a 1-wise coverage method. The strategy is that each value must be in a different pair of test cases. By this method, all values that cause a single-mode fault will be detected and identified. For example as in the example shown above, the test suite consists of the following 6 test cases:

Test 1 = 1111 Test 4 = 2312
Test 2 = 1222 Test 5 = 3231
Test 3 = 2133 Test 6 = 3323

If Test 3 and Test 5 show an unexpected result, then it must be because of the value C3. The test suite can work as well as the base-choice combination while the number of test cases from this method is lower.

Let us introduce the two basic properties:

Property 1: If the system has k parameters, with n values in each parameter, the number of test cases is 2n, provided that $k \leq 2n - 1$.

(As in the example, if we have n = 3 and k = 4, then the number of test cases is 2 x 3 = 6.)

Proof: The number 2n is derived from the fact that each value has to be in a pair of tests. We have k parameters and each parameter has n values, therefore there must be kn values in the system.

From 2n test cases there are $\binom{2n}{2}$ = n(2n - 1) pairs of test cases. So, we have the condition $n(2n - 1) \geq kn$, which can be reduced to $k \leq 2n - 1$. //

Next we compare the number of test cases of the new method to the base-choice combination method.

Recall that the number of test cases of the new method is 2n, and that of the base-choice combination method is k(n - 1) + 1. We see that the inequality

$$2n \leq k(n - 1) + 1$$

is equivalent to

$$k \geq 2 + \frac{1}{n-1}. \text{ ----------(A)}$$

Assume that n > 1, then the inequality (A) is true provided that k > 2. So, we draw the conclusion:

Property 2: If there are more than two parameters in the system, and more than 1 value in each parameter, then the number of test cases of the proposed method is less than that of the base-choice combination method.

The algorithm below generates the test suite by the proposed strategy. Let the system have n parameters, and each parameter have n values. Also, let $k \leq 2n - 1$.

Algorithm 3:

Step 1. Construct the round robin tournament schedule of 2n players.

Step 2. Map the table from step 1 to the test case table. Each round corresponds to the parameter, and each pair corresponds to the pair of test cases in which the value is put.

Example: Let n = 3 and k = 4. (We have $k \leq 2n - 1$.) There are 6 test cases.

Step 1: We have the table of a round robin tournament

Round:	1	2	3	4
pair:	12	13	14	15
	34	25	26	24
	56	46	35	36

Step 2: The i^{th} row of pair indicates the test case which the value i is to be inserted. Let us consider Round 1 which corresponds to parameter A. The pair 12 in the first row indicates that the value 1 has to be inserted in the Test 1 and Test 2; the pair 34 in the second row indicates that the value 2 has to be put in Test 3 and Test 4. Continue this process until we have the complete table:

Parameter:	A	B	C	D
Test 1:	1	1	1	1
Test 2:	1	2	2	2
Test 3:	2	1	3	3
Test 4:	2	3	1	2
Test 5:	3	2	3	1
Test 6:	3	3	2	3

This is the same test suite as shown before.

What happens if the numbers of values in parameters are different? In this case the number of test cases is twice the maximum number of values in the parameters. Some values in the small parameter may appear in more than two test cases. It is not difficult to see that Property 2 is also true in this case.

If there are too many parameters, i.e. if k > 2n -1, more test cases must be added to the test suite so that the number of pairs of test cases is larger than the number of values. For example, 8 test cases can handle up to 9 parameters with 3 values each. In this case, some values must appear in more than two test cases.

If we use the strategy that 3 test cases specify a fault value, if n = 3 then 9 test cases are required. The number of different 3-test-case sets is $\binom{9}{3}$ = 84. So the test suite can handle up to 84 values, or 28 parameters.

The pair-wise test suite and the social square schedule

Here we give a table of pair-wise test suite for 4 parameters, with 3 values each. The test suite consists of 9 test cases. (Bolton, 2004; Maity & Nayak, 2005)

Test 1	=	1111	Test 6	=	2312
Test 2	=	1222	Test 7	=	3132
Test 3	=	1333	Test 8	=	3213
Test 4	=	2123	Test 9	=	3321
Test 5	=	2231			

It is not difficult to see that the pair (i, j) is contained in some test cases. For example, the pair (A1, B3) is in Test 3; (B2, C1) is in Test 5; (A2, C2) is in Test 6.

Let us recall the social square schedule for 9 players which are grouped into 3 groups for each round. We rewrite them in the new matrix form in such a way that each round is listed in columns while each group is listed in rows, correspondingly. For convenience, we rotate the column from 4 to 1, 1 to 2, and so on. Now, the columns represent the parameters, the rows represent the values, and the number in the rows represent the test case number that contains that value.

Column:	1	2	3	4
	123	147	168	159
	456	258	249	267
	789	369	357	348

Next we map the table into the test case table using the rule mentioned above. For example, the number 123 in the first row and first column means that the value 1 of parameter A has to be put in Test 1, Test 2 and Test 3; the

number 357 in the third row and third column means that the value 3 of parameter C has to be put into Test 3, Test 5 and Test 7. After finishing the mapping we obtain:

Parameter:	A	B	C	D
Test 1	1	1	1	1
Test 2	1	2	2	2
Test 3	1	3	3	3
Test 4	2	1	2	3
Test 5	2	2	3	1
Test 6	2	3	1	2
Test 7	3	1	3	2
Test 8	3	2	1	3
Test 9	3	3	2	1

This table shows the same pair-wise test suite as given before.

We shall prove in general that the test suite generated by the social square schedule is pair-wise.

Proof: Let us consider the system of k parameters, with n values each. Suppose that we have the social square for n^2 people, which consists of n + 1 rounds. So, the number k must be less than or equal to n + 1. The test case table mapped from the social square schedule has n^2 rows and n + 1 column; each row (test case) corresponds to the player and each column (parameter) corresponds to the round.

In each column, each value must be contained in n test cases. This is derived from the fact that each group consists of n players.

We will show that each pair of values (i, j) is contained in not more than one test case. Consider any two columns, say A and B. Suppose that the value i of A is in Test p and Test q, if the value j of B is in Test p, then that value must not be in Test q. This is because Test p (player p) must meet Test q (player q) once.

Next, we will show that each pair of values (i, j) in columns (A, B) must be contained in some test case. For each column, we have n groups of test cases and each group corresponds to each value. Since the value j of B must be in only one test case in one group of A, then each value j of B has to be in one test case in each group of A. That is, the pair (i, j) is in the same test case. Thus, the table is the pair-wise test suite. //

Conclusion

We have shown that the round-robin tournament scheduling can be used to generate the test suite for software testing. The proposed method based on the strategy that a value must be contained in two or more test cases, produce the 1-wise coverage test suite with the less number of test cases than that of the base-choice combination method. In practice, if we know in advance that there is no relation between parameters so that the pair-wise testing is unnecessary, then using the proposed test suite will save time and energy greatly.

References

Bolton, Michael "DevelopSense Pairwise Testing", 2004. http://www.developsense.com/testing/Pairwise-Testing.html

DeVenezia, Richard. A., "Round-Robin Tournament Scheduling", 2006, http://www.devenezia.com/downloads/round-robin/index.html

Grindal, M., B. Lindstrom, A., J. Offutt, and S. F. Andler, "An Evaluation of Combination Strategies for Test Case Selection", 2004. Technical Report HS-IDA-TR-03-001, Department of Computer Science, University of Skovde, Sweden.

Jorgensen, Paul C. "Software Testing: A Craftsman's Approach", 1995, CRC Press, ISBN 0849308097

Maity, Soumen, and Amiya Nayak, "Improved Test Generation Algorithms for Pair-Wise Testing", 2005, Department of Mathematics, Indian Institute of Technology, Guwahati, India.

Phadke, Madhav S., "Planning Efficient Software Tests", Phadke Associates, Inc. http://www.stsc.hill.af.mil/crosstalk/1997/10/planning.asp

Wikipedia: The free encyclopedia, "Round-Robin Tournament", 2007. http://en.wikipedia.org/wiki/Round-robin tournament