ПЛОВДИВСКИ УНИВЕРСИТЕТ "ПАИСИЙ ХИЛЕНДАРСКИ", БЪЛГАРИЯ
НАУЧНИ ТРУДОВЕ, ТОМ 36, КН. 3, 2009 – МАТЕМАТИКА
PLOVDIV UNIVERSITY "PAISSII HILENDARSKI", BULGARIA
SCIENTIFIC WORKS, VOL. 36, BOOK 3, 2009 – MATHEMATICS

ALGORITHMS TO MINIMIZE THE NUMBER OF UNIQUE TESTS IN REAL GROUP TESTING EXAMINATION

Angel Golev, Olga Rahneva, Asen Rahnev

Abstract. This paper describes algorithms for finding fast the minimal number of unique tests in real group test examinations in Distributed e-Testing Cluster – DeTC, depending on the neighboring seats configuration. It suggests how to arrange unique tests on the testing seats, so that neighbors have different tests.

The paper also suggests what algorithms should be applied and in what order for different configuration of the neighboring testing seats.

Key words: DeTC, e-testing, algorithms, unique tests, neighbors

1. Introduction and Formulation of the Problem

Testing examination is one of the most popular and well-developed assessment instruments in higher education [2]. Existing systems for electronic testing offer solutions for testing one single student. The Distributed e-Testing Cluster (DeTC) [6, 7, 8] examines the problem of testing group of students.

Proper and adequate assessment is possible only when it is guaranteed that no cheating has taken place during the examinations. Invigilators have the task of preventing cheating, but sometimes even they cannot cope with the problem of having a learner copying from one's neighbors, which are in one's field of vision.

One way to overcome the problem of cheating by copying is to randomize the order of questions in tests, and the order of answers in same questions. However, this can only reduce the chance of copying but not eliminate it.

Copying from neighbors will not be possible is each learner is given a unique test. A large number of learners, who take examinations simultaneously, make it very difficult to provide each with a unique test. Further, to ensure fair assessment, all tests must be of equal difficulty and size. Creating many unique tests requires excessive efforts by authors of questions and tests.

DeTC is used to conduct real-life electronic assessment of learners. Its Test Generating Tool (TGT) can facilitate generation of unique tests through the use of parameterized dynamic questions [7]. However, in certain subjects, questions cannot be parameterized and authors have to prepare a number of unique tests to ensure minimal chance of cheating by copying.

This paper describes algorithms which find the minimum possible number of unique tests, depending on the location of testing seats and the field of vision of each seat. The algorithms are examined for their efficiency and relevancy depending on the specific configuration of neighboring testing seats.

We define that two testing seats are neighbor seats, if one is within the field of vision of the other, and therefore the learner can copy from one's peer.

The solution of the problem can be reduced to the task of coloring a rectangular grid of vertexes (seats) [4].

The testing seats are defined through the use of an adjacency matrix. The output of the programs fills in the matrix with the numbers if the unique tests for each seat.

The neighbor seats (neighbors) can be defined for all seats with the help of a template of neighborhood. The template contains the coordinate offsets - row and column for each seat. It is possible to define a template for a whole line of seats in the room. The template can be created manually as a list of offsets, or with the help of a specialized tool for creating neighborhood templates, which allows users to select the neighbors for each set via a graphical user interface.

Example: A learner taking a test can see the seats on one's left and right, the seats on the left and right of the seat in front of the learner. Further, one's seat can be seen from the seats on the left and right of the one behind the learner. The schema is displayed on Figure 1.

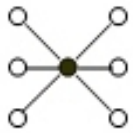

Figure 1

The template of neighborhood for this example is: -1,-1; -1,1; 0,-1; 0,1; 1,-1; 1,1.

The tool for creating neighborhood templates supports creation of any kind of neighborhood configurations.

2. Algorithms

2.1. Algorithm 1a: Greedy algorithm with traversing the neighbors

The algorithm selects a free number of a unique test for every next traversed seat, from 1 to the current maximum, depending on the number of neighbors. When there is no available unique test, the algorithm increments the maximum number.

In the examined cases it is better to start traversing vertexes from a corner of the matrix. If traversal starts from an inner vertex, the output is very poor. The algorithm produces close to optimal solutions when all seats have very similar fields of vision, i.e. neighbors, and when traversal always starts in one and the same direction – for example, clockwise. When neighbor seats form different configurations in several columns, the result of the algorithm depends on the order of traversal of neighbors, and it is not clear what traversal order will produce best results.

```
Algorithm 1 // Greedy
{
1.   x, y, i, j:  Integer;
2.
3.   while find_empty_element(x,y) do
4.     places[x,y] := first_test_num;
5.     push_queue(x,y);
6.     while pop_queue(x,y) do
7.       foreach ngh in possible_neighbours(x,y) do
8.         i := x + neighbours[ngh].dx;
9.         j := y + neighbours[ngh].dy;
10.        if valid_test_num(i,j) and places[i,j] = 0 then
11.          places[i,j] := get_test_num(i,j);
12.          push_queue(i,j);
13.        endif
14.      done
15.    done
16.  done
}
```

Figure 2. Algorithm 1a

The example on Figure 3 describes neighbors for all seats: 1,1; 1,0; 1,-1; 0,-1; -1,-1; -1,0; -1,1; 0,1 and different neighbors for columns 2 and 3, as shown on Figure 3.

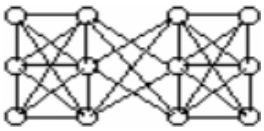

Figure 3. Example neighborhood template

When traversal is done in the order of neighbors as described above, the solution is:

1 4 1 3
3 2 2 4
1 4 1 3

If traversal is done in the order: -1,1; 0,1; 1,1; 1,0; 1,-1; 0,-1; -1,-1; -1,0, and the upper neighbors for columns 2 and 3, the solution will be:

1 2 2 3
4 3 1 5
1 5 4 2

The greedy algorithm can be also applied when the neighbors of seats are given in random order. The output is usually not optimal in this case, but when the number of unique tests is greater than 9 and the neighbors are at maximum a line and column away, the following solution with 9 unique tests can be used:

(1)
1 2 3 1 ...
4 5 6 4 ...
7 8 9 7 ...
1 2 3 1 ...
... ...

2.2. Algorithm 1b: Greedy algorithm with traversing vertexes in descending order by number of neighbors

This algorithm determines the number of unique test for each seat just like Algorithm 1a. Algorithm 1b can be used, when neighbors are defined in a random order. It gives a good results, if the probability of having neighbors

for each seat is less than approximately 0.8, with having neighbors a row and column away at maximum. When seats have more neighbors or the number of seats is large, the algorithm is increasingly inefficient.

2.3. Algorithm 1c: Greedy algorithm with assigning non-neighbor seats with same unique tests

The algorithm starts assigning the elements of the matrix with the number of unique test 1. It traverses the whole array by diagonals. The algorithm assigns 1 to each seat, which has no test assigned, and has neighbors with number 1. The algorithm repeats the procedure with the next number for unique test, until all seats are assigned a test.

```
Algorithm 1c // Greedy
{
1.   new_test := 1; exists_empty_element := true;
2.   while exists_empty_element do
3.     row := 1; col := 0;
4.     exists_empty_element := false;
5.     while find_next_diagonal_element(row,col) do
6.       if places[row,col] = 0 then
7.         if test_fits_place(row,col,new_test) then
8.           places[row,col] := new_test;
9.         else
10.          exists_empty_element := true;
11.        endif
12.      done
13.      inc(new_test);
14.  done
}
```

Figure 4. Algorithm 1c

2.4. Algorithm 2: Backtracking

The algorithm performs full traversal of the first t rows with backtracking, while simultaneously assigning values to the next t−1 rows: matrix[i+t, j] = matrix[i, j], i=1, t−1.

Users define in advance the maximum number of unique tests they want to have. The algorithm selects t based on the desired maximum number of unique tests. The solution for the first t lines is replicated until the end of the array. This algorithm is applicable when neighbors of all seats are defined by

one and the same template. The solution may be not the optimal one, but the arrangement of tests per testing seats will be the same for all groups of t−1 rows.

2.5. Algorithm 3: Traversal by diagonals with backtracking

Again, users define in advance the maximum number of unique tests they want to have. If the algorithm cannot find an available unique test for a seat during traversal, it tracks back. The optimal solution can be achieved reasonably enough for the given configurations. The algorithm starts with two unique tests, and increases their number until a solution is found, or the maximum number of unique tests reaches 9. When there is no solution with k unique tests, computations quit for time of approximately O(k!), which is a reasonable value for k < 9.

```
Algorithm 3 (n,m,max_test_num) // Diagonally_backtracking
{
1.  row, col, current_test_num:  Integer;
2.
3.   places[1,1] = 1;
4.   row = 2;
5.   col = 1;
6.   current_test_num = 0;
7.   while true do
8.     new_test_num =
9.       get_next_possible_test_num(row, col, current_test_num);
10.    if new_test_num <= max_test_num then
11.      places[row,col] = new_test_num;
12.      if row==n and col==m then exit(there_is_solution);
13.      find_next_diagonal_element(row,col);
14.      current_test_num = 0;
15.    else
16.      places[row,col] = 0;
17.      find_prev_diagonal_element(row,col);
18.      if row == 2 and col == 1 then exit(no_solution);
19.      current_test_num = places[row,col];
20.    endif
21.  done
}
```

Figure 5. Algorithm 3

2.6. Algorithm 4: Traversing neighbors in width with backtracking

This algorithm selects in advance a maximum number of unique tests k. The order of traversal is determined on the first iteration. Then the algorithm performs traversal with backtracking in the selected order. This algorithm is faster than Algorithm 3, when there is a solution.

```
Algorithm 4 (n,m,max_test_num) // Backtracking
{
1.   row, col, current_queue_el,
2.   new_test_num, current_test_num:  Integer;
3.
4.   find_the_order_for_Breadth_First_Search;
5.
6.   set_array_places(n,m,0);
7.   places[1,1] = 1;
8.   current_queue_el = 2;
9.
10.  while current_queue_el <= length(srch_queue) do
11.    get_coord_from_queue(row,col,current_queue_el);
12.    current_test_num = places[row,col];
13.    new_test_num =
14.      get_next_possible_test_num(row, col, current_test_num);
15.    if curr_test_num <= max_test_num then
16.      places[row,col] = new_test_num;
17.      inc(current_queue_el);
18.    else
19.      places[row,col] = 0;
20.      dec(current_queue_el);
21.      if current_queue_el = 1 then exit(no_solution);
22.    endif
23.  done
}
```

Figure 6. Algorithm 4

2.7. Algorithm 5: An ant algorithm

This algorithm is applied with probability node_probability = 0.8, and test_num_probability = 0.6, and maximum number of moves set to 100 000. The number of ants depends on the size of the room.

```
Algorithm 5 (max_test_num:  Integer) // Ants algorithm
{
1.   make_some_not_good_solution;
2.   Count_Violations;
3.   foreach ant in all_ants do
4.     set_ant_coord(ant);
5.   foreach 1..max_ant_moves do
6.     foreach ant in all_ants do
7.       make_ant_move(ant);
8.       choose_new_test_num(ant);
9.       if good_arrangement then exit(there_is_solution);
10.    done
11.  done
12.  exit(no_solution);
}
make_ant_move(ant);
{
13.  if random() <= node_probability then
14.    max_violations = 0;
15.    foreach neigh in neighbours_list(ant.position) do
16.      max_violations = max(max_violations, violations[neigh]);
17.    done
18.    ant.position = chose_random_neighbour(ant, max_violations);
19.  else
20.    ant.position = chose_random_neighbour(ant);
21.  endif
}
choose_new_test_num(ant);
{
22.  if random() <= test_num_probability then
23.    new_test_num = find_free_test_num(ant.neighbours);
24.    if new_test_num == 0 then
25.      new_test_num = random(new_test_num)+1;
26.  else
27.    new_test_num = random(new_test_num)+1;
28.  endif
29.  places[ant.position.x, ant.position.y] = new_test_num;
30.  foreach neigh in neighbours_list(ant.position) do
31.    recount_violations(neigh);
}
```

Figure 7. Algorithm 5

3. Solution for seats with the same neighborhood template

When all seats have the same neighborhood template, Algorithms 4 or 3 are most suitable to find an optimal solution. They both determine k number of unique tests, with $2 \leq k < 9$. These two algorithms are fast even when there is no solution to the problem. The solution of these algorithms can be improved further by running Algorithm 2 – it will find a solution, symmetric by rows.

The tests are carried out with seat matrices of maximal size 200×200 seats.

The following example demonstrates that algorithm will not produce an optimal solution regardless of the traversal order. The neighborhood template is:

```
  o o o
o o o o o
o o × o o
o o o o o
  o o o
```

Solutions with Algorithms 1a, 3 4 for a matrix of 6×6 seats and the upper neighborhood template:

Alg. 1a

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

Alg. 3

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

Alg. 4

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

4. Solution for randomly selected neighbors

Algorithms 3 and 4 are not suitable when neighbors are selected randomly. They are not efficient, and sometimes it may be even impossible to produce a solution with them, because the increased number of possible unique tests causes the total number of combinations to grow drastically.

In this case Algorithms 1a, 1b and 1c should be used. When the number of neighbors is relatively small, Algorithm 1b is better than 1a. Algorithm 1c gives a better solution than 1a when the probability for selecting a neighbor is greater than 0.8, or when the number of seats is greater than 100. The solution can be enhanced with the algorithm of the ant by searching for a solution with less than 9 unique tests, and less than the result of the greedy algorithms – 1a, 1b, and 1c.

The algorithm of the ant seldom produces a solution for large number of neighbors and 8 unique tests. The experiments demonstrate that the algorithm of the ant gives a better solution than the greedy algorithms when the number of neighbors is relatively small. Experiments are conducted with seat matrix of 30×30.

The solution with minimal number of unique tests is selected from all solutions, and if the number is greater than 9, the standard solution (1) with selected maximum number of neighbors is selected.

5. Acknoledgement

This research has been partially supported by the Bulgarian NSF under Contract No VU-MI 107/2005 and by project No IS-M-4/2008 of Department for Scientific Research, Plovdiv University "Paisii Hilendarski".

References

[1] Brassard G., Bratley P., *Algorithmics: Theory and Practice*, Prentice Hall, 1988.

[2] Brusilovsky P., Miller P., *Web-based Testing for Distance Education*, WebNet 1999, pp. 149-155.

[3] Comellas F., Ozon J., *An Ant Algorithm for the Graph Colouring Problem*, ANTS'98 - From Ant Colonies to Artificial Ants: First International Workshop on Ant Colony Optimization, Brussels, Belgium, 1998.

[4] Cormen T., Leiserson C., Rivest R. , Stein C., *Introduction to Algorithms*, 2nd edition, The MIT Press, 2001.

[5] Nakov P., Dobrikov P., *Programming = ++Algorithms*, TopTeam Co, Sofia, 2005 (in Bulgarian).

[6] Rahneva O., *Testing and Assessment in Distributed Electronic Testing Cluster – DeTC*, 12th International Conference Electronics '2003, Sozopol, 24-26 Sept. 2003, Conference Proceedings, v. 4, pp. 214-219.

[7] Rahneva O., *Generating Dynamic Questions in Distributed e-Testing Cluster – DeTC*, ECEST'04, Bitola, 2004, v.1, pp 305-308.

[8] Rahneva O., Rahnev A., Pavlov N., *Functional Worklflow and Electronic Services In a Distributed Electronic Testing Cluster – DeTC*, Proceedings 2nd International Workshop on eServices and eLearning, Otto-von-Guericke Univ. Magdeburg, 2004, pp 147-157.

Angel Golev, Asen Rahnev
Faculty of Mathematics and Informatics
University of Plovdiv
236 Bulgaria Blvd.,
4003 Plovdiv, Bulgaria
e-mail: angelg@uni-plovdiv.bg, assen@uni-plovdiv.bg

Received 24 January 2008

Olga Rahneva
University of Food Technologies
Dept. of Informatics & Statistics
26 Maritsa Blvd.,
4000 Plovdiv, Bulgaria
e-mail: rahneva@hiffi-plovdiv.acad.bg

АЛГОРИТМИ ЗА НАМИРАНЕ НА МИНИМАЛЕН БРОЙ РАЗЛИЧНИ ТЕСТОВЕ ПРИ РЕАЛНО ГРУПОВО ТЕСТОВО ИЗПИТВАНЕ

Ангел Голев, Олга Рахнева, Асен Рахнев

Резюме. В работата се описват алгоритми за бързо намиране на минимален брой тестови варианти при провеждане на реално групово тестово изпитване в разпределения клъстер за електронно тестване – DeTC в зависимост от конфигурациите на съседните места за изпитване. Предлага се как тестовите варианти да се разположат на местата за изпитване, така че на съседните места да има различни тестове.

За постигане на оптимално решение се предлага кои от алгоритмите и в какъв ред да се приложат при различни конфигурации на съседните места за изпитване.