澳門科技大學
MACAU UNIVERSITY OF SCIENCE AND TECHNOLOGY

ME001, MIIE01
Information Systems Analysis and Design
Advanced Topics in Information Technology

Mini Project Report and User Guide

(Set Cover Problem)

Group ID: 8

GuangCheng-Li	Hong-Liang	Rui-Jiang	YiQi-Wang
李光程	梁轰	蒋蕊	王艺钦
16098537-II20-0016	1609853J-II20-0027	1609853J-II20-0030	1609853J-II20-0055
sky9475@126.com	coolboom@foxmail.com	1458952792@qq.com	86285457@qq.com

2016
FIT, M.U.S.T, Macau

CONTENTS

1. User Guide

(1) You should copy the application (i.e., Fig 1) to your hard-disk instead of running it in USB-Driver.

Fig. 1 Application file

(2) Running the application. If you find that a yellow message bar appears with a shield icon and the Enable Content button (i.e., Fig 2), please click "Enable Content" or "啟用內容".

Fig. 2 An example of the Message Bar when macros are in the file

(3) Input values of m, n, k, j, s, like Fig. 3.

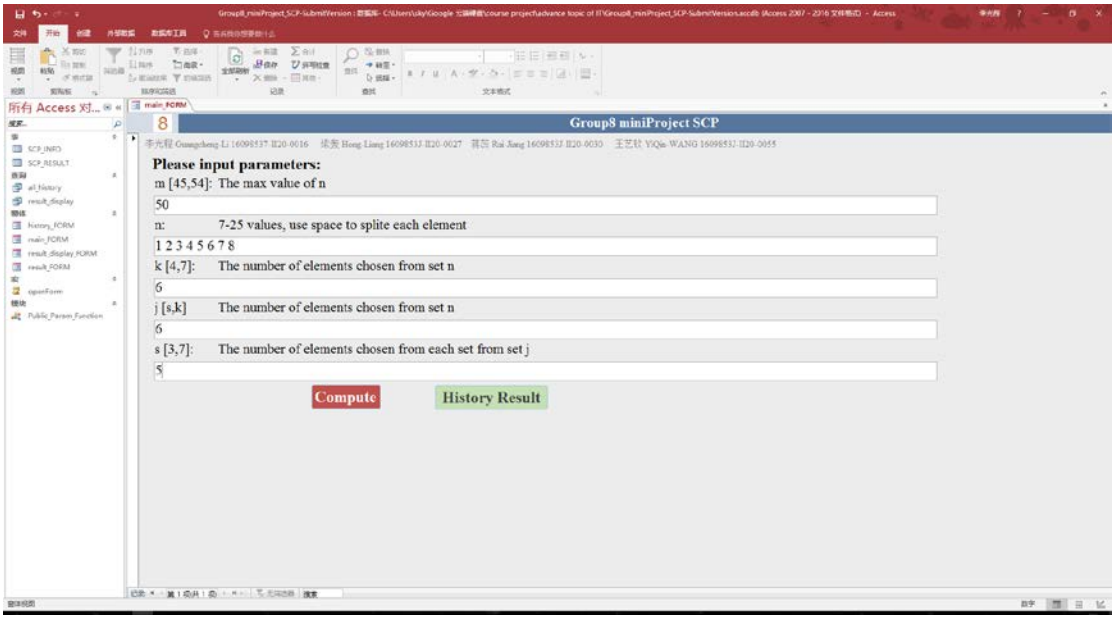

Fig. 3 Input values

(4) Click "Compute" and the program will display the amount of the selected sets (i.e., Fig. 4).

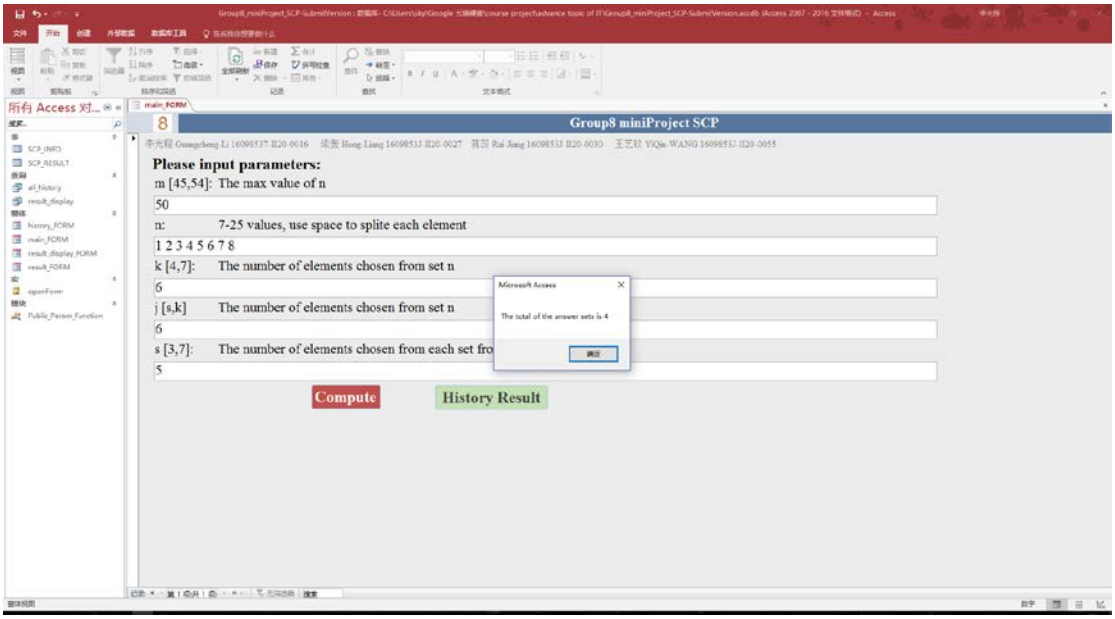

Fig. 4 The amount of the selected sets

(5) Click "Confirm" to watch the selected sets (i.e., Fig. 5).

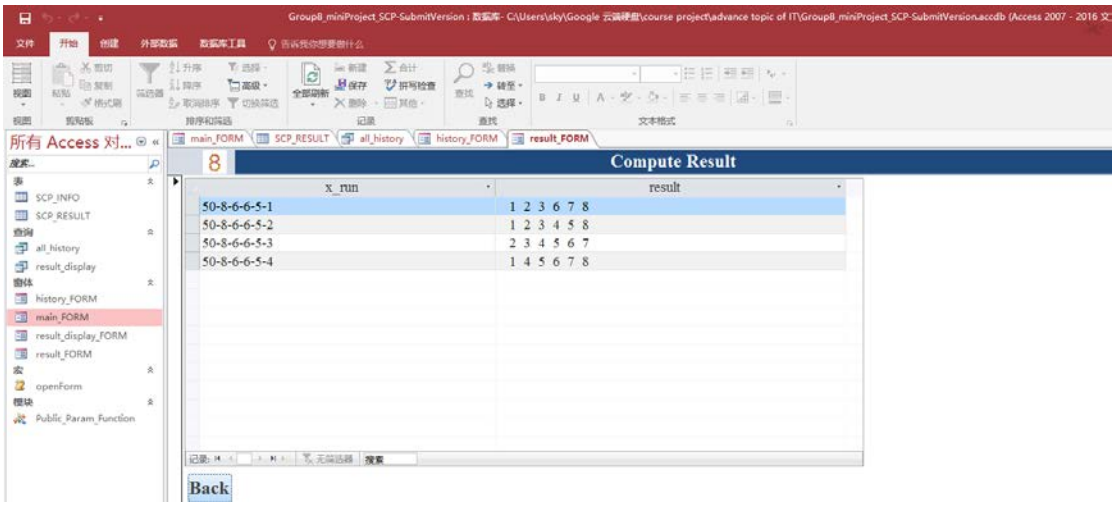

Fig. 5 The result of problem window

(6) You can click "Back" to back to the main form.

(7) You can click "History Result" in the main form to show all of results.

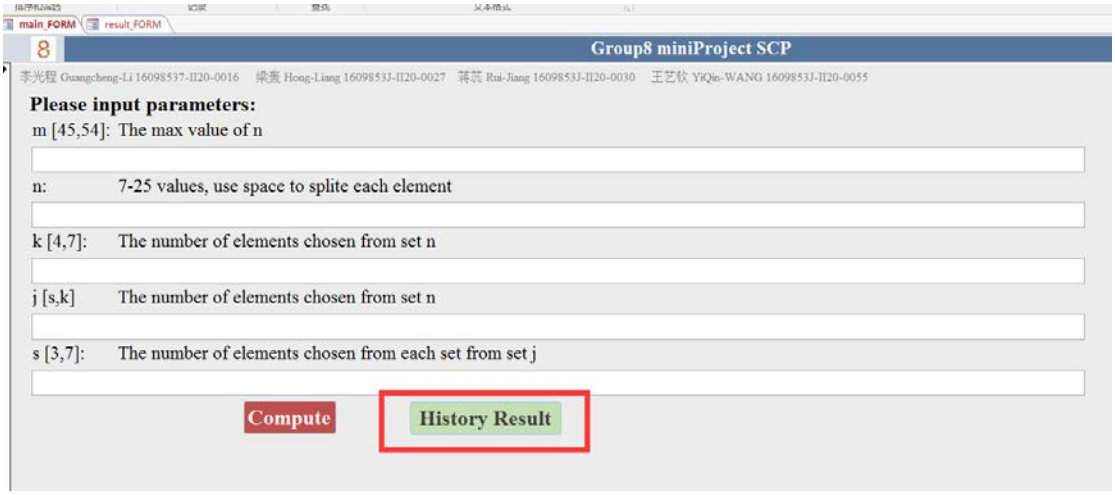

Fig. 6 The method to display history results

(8) If you want to delete an item in result, you can select it first then click "Delete" to remove it (i.e., Fig. 7, Fig. 8).

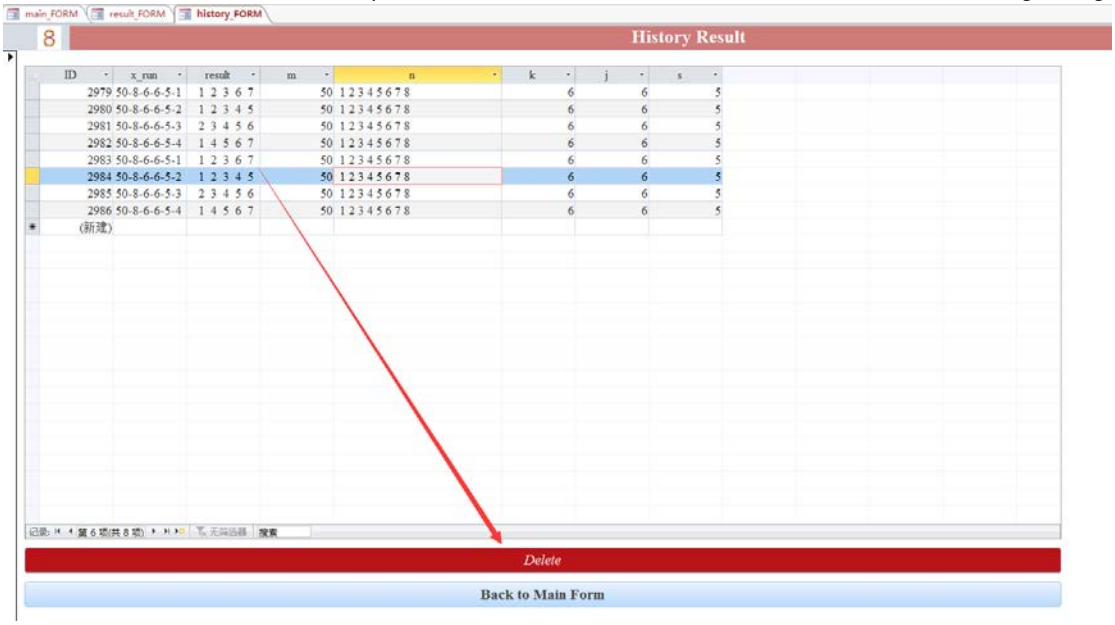

Fig. 7 Remove a result item

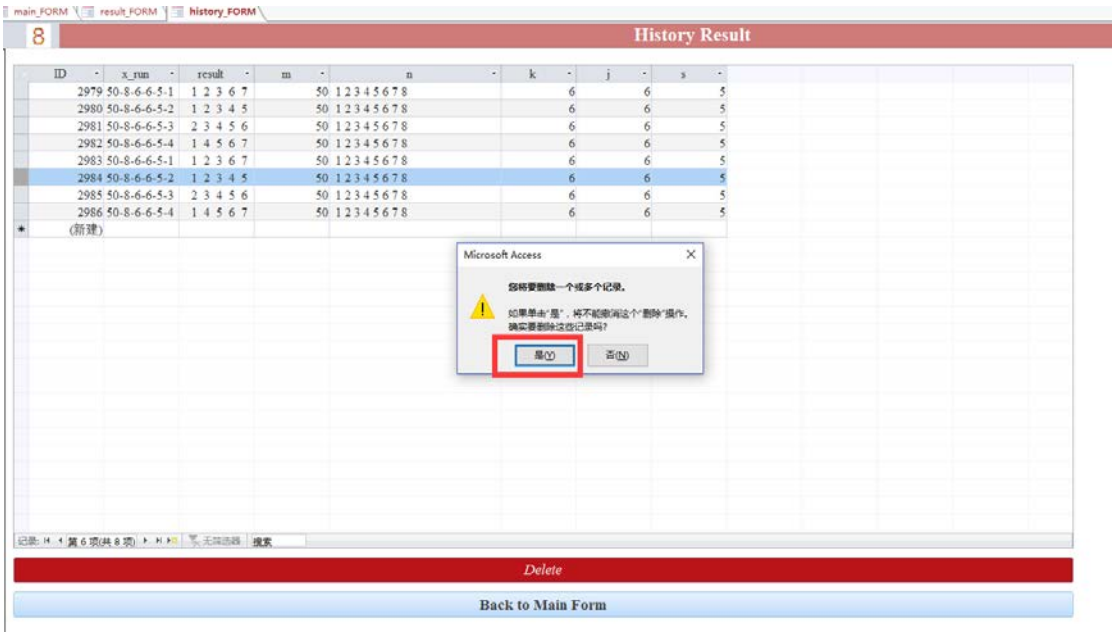

Fig. 8 Click “yes” to confirm

(9) You can also select several items to remove them at same time (i.e., Fig. 9), and you can click “Back to Main Form” to back to the main form.

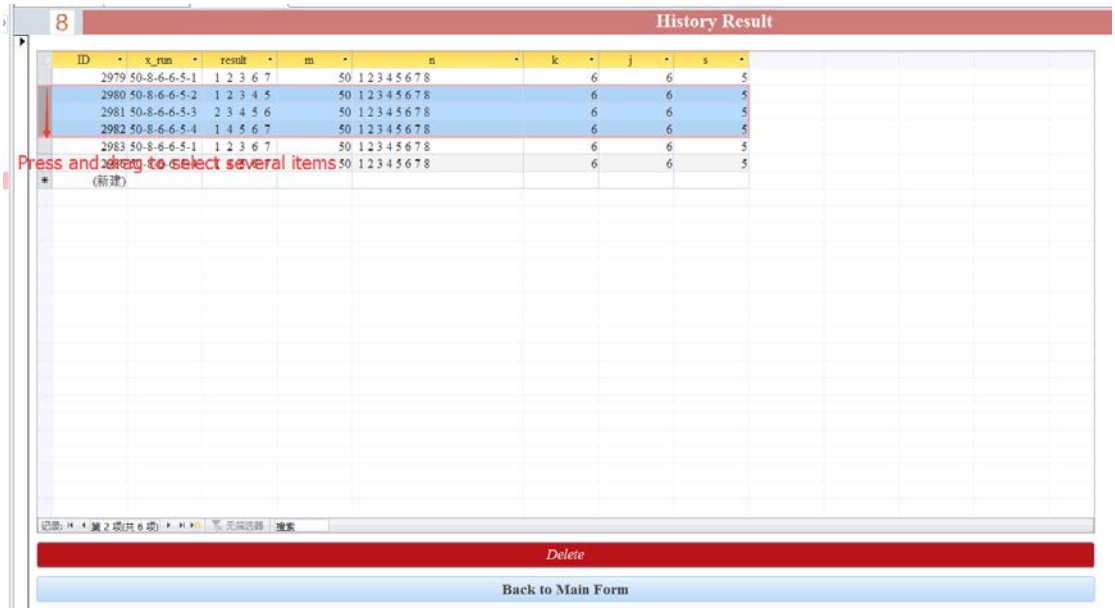

Fig. 9 Remove several items

2. Introduction

The method that we proposed to solve the minimum set cover problem can be divided into two parts. The first part is to solve how to generate sets in a way that each set is as far different as the ones generated before, which means each generated set should have the least same numbers as former ones. The second part is to label the sets those needed to be covered (we call them as samples) as covered using the sets generated from the first part. When all the samples are covered, the program will stop and the selected sets from sets generated by the algorithm will be the answer.

The following parts will be arranged as follow: The manual of our system is introduced in section 1. Section 3 will be an introduction and explanation of the generation algorithm. Section 4 will be the method we use to calculate samples covering using sets generated by the generation algorithm. In section 5, we give out the conclusion, unsolved problems and analysis to them.

3. Generation algorithm

E.g. m=45, n=9, k=6, j=5, s=5. We want to select 6 numbers from 9 numbers, but if we select 9-6=3 numbers, the operation will be easier. So, if the selected numbers are more than the unselected numbers (we call them absent numbers), in this example we will find the combinations of 3 numbers from 9 numbers. So, the total number of the combination of absent parameters is C_9^3=84. Finally, we will calculate the complementary sets of these combinations which will be the answer of the minimum set cover problem. Our main idea is to let each of the n numbers be absent with the same times.

To make the explanation simple, for each set, 3 numbers are absent. We do the following steps.

Step 1. Let ABC be absent in the first set.

Step 2. Let DEF be absent in the second set.

Step 3. Let GHI be absent in the third set.

We gather these 3 sets to form a group. Then we can find out that in this group of 3 sets, ABCDEFGHI are absent one time for each. In this way, these 3 sets can cover as many samples as possible. Fig.10 shows the main idea of our generation algorithm.

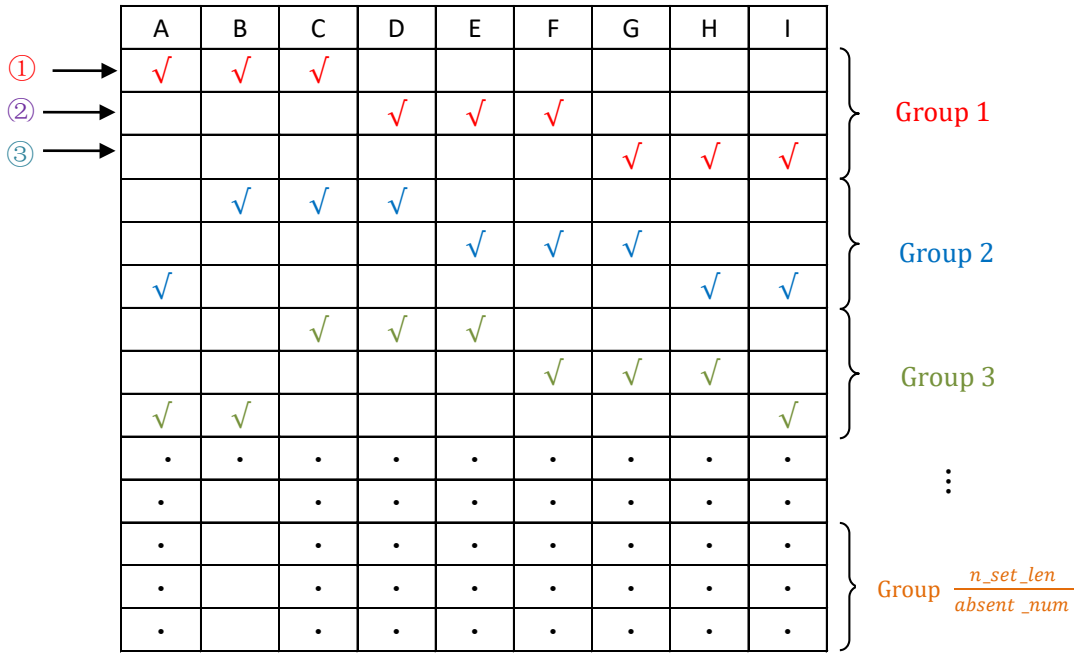

Fig.10 Main idea of the generation algorithm

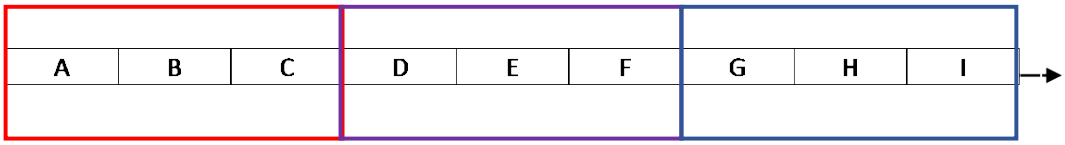

Fig. 11 A group of sets of absent elements

Fig. 10 and Fig. 11 just show the main idea of our generation algorithm. But to generate these sets actually in the way we want, we built a Hash function: *y=group*order+c*. It is used for absent elements selection (*group*: the index of the current order, order: the order the current element's selection, *c*: the number absent parameters). The number of absent parameters is (*n- k*).

When we apply this Hash function in our system, we found out that it has a cycle, which means after several times generation, we will get the same sets as before. So, we add a random function for absent elements selection after the Hash function has been run out off.

In the example above, we can see that 9(*n*) can be exactly divided by 3(*n-k*), but actually we still have a condition that *n* can't be exactly divided by *n-k*. So, we need to deal with these two conditions separately.

To generate the sets, we need to calculate:

(1) How many sets there are in each group (*set_per_group*);

(2) How many groups of sets are divided into (*group*).

And we store a group into a one dimensional array *a[group][0..(n-1)]*. Each row of the array a should be divided into *set_per_group = n/(n-k)* sets.

But for the condition that *n%(n-k) ≠ 0*, we need to extend the array a's column so that the amount of the elements in each of the array *a*'s row can be exactly divided by (*n-k*). The extended length should be *n + ((n-k) – n%(n-k))*. Detailed explanations and calculations are as the follow.

3.1 Divisible

In this condition, n can be divisible by the number of absent parameters.

The number of groups can be calculated using formula (3.1.1).

$$set\_per\_group = n/(n-k) \qquad (3.1.1)$$

$$group \in [\ 1, \lceil C_n^k / set\_per\_group \rceil\ ] \qquad (3.1.2)$$

For formula 3.1.1, *n* is the length of the n numbers that use input in our system and *k* is the number that user input, too. For formula 3.1.2, C_n^k is the combination of k numbers from n numbers.

$$a[group][0..n-1] = \{set\_n(idx)\ \ | idx = [group * order + (n-k)]\ \ \%\ \ n, idx \epsilon [0, n-1]\} \qquad order \in [1, (n-k) * C_n^k] \qquad (3.1.3)$$

For formula 3.1.3, *idx* means the index of the n numbers array which is calculated by the hash function we proposed before. If the *idx* value is larger than (*n*-1), we set *idx* to 0, and increase it by 1 until that index is not selected.

$$sets_i = \{divide\ \ each\ \ a[group][n-1] into\ \ (set\_per\_group) groups, each\ \ group\ \ has\ \ (n-k)\ \ parameters, i.e. sets_i,\ \ i \in [1, C_n^k]\} \qquad (3.1.4)$$

For formula 3.1.4, each set generated by the algorithm comes from each row of the array *a*. For example, a[0][0..n-1] can be divided into *n/(n-k)* sets and each set has (*n-k*) numbers.

$$answer\_sets_i = \{s \in (set\_n\_sets_i), i \in [1, C_n^k]\} \qquad (3.1.5)$$

For formula 3.1.5, the answer sets come from the set of $sets_i$.

3.2 Aliquant

In this condition, n can't be divisible by the number of absent parameters. Similar to 3.1, we can calculate *set_per_group* (in formula 3.2.1), *group* (in formula 3.2.2) and *order* (in formula 3.2.4).

$$set\_per\_group = \lceil n/(n-k) \rceil \qquad (3.2.1)$$

$$group \in [\ 1, \lceil C_n^k / set\_per\_group \rceil\ ] \qquad (3.2.2)$$

$$array\_len = n + [(n-k) - n\%(n-k)] \qquad (3.2.3)$$

$$order \in [1, (n-k) * C_n^k] \qquad (3.2.4)$$

But there are a few differences. The column length should be extended since *n* can't be exactly divided by (*n-k*). So, we need to increase the length of the column to *(n + (n-k) - n%(n-k))*. The array *a* is divided into two parts. The first part is from 0 to (*n-1)* and the second part is from *n* to (*array_len* – 1). As for the part 0 to (*n-1*), we use the hash function to get the selected elements. But for the part n to (*array_len - 1*), we have two constrains, which are the index can't be the index of the last *(n%(n-k))* elements and the index of the elements selected in array *a[group][n..(array_len-1)]*.
For example, if *n* is 8 and *k* is 5. Then the (*n-k*) is 3. The *array_len* is 8 + ((8-5)-8%(8-5)) = 9. Then for the first part 0 to 7, we may have the indexes like 1,2,3,4,5,6,7,8. We have to select one more index added to these 8 indexes. This index can be the same as 7 or 8, otherwise when these 9 number are divided into 3 sets, the last set will be 7,8,7 or 7,8,8. Another constraint is that consume that we need to select more than one elements added to those 8 indexes, that is 2. These 2 indexes cannot be the same as 7 or 8 and themselves, otherwise it may come up with this kind of situation that the last group will be 7,8,1,1 or 7,8,7,7. We add these two constraints in order to avoiding the last set has repeated elements. Formula can be seen below.

$$a[group][0 \cdots (n-1)] = \{idx \ \ | idx = [group * order + (n-k)] \ \% \ n, idx \epsilon [0, n-1]\}$$
$$a[group][n, array\_len - 1] = \{set\_n(idx) \ \ | idx = [group * order + (n-k)] \ \% \ n, idx \epsilon [0, n-1], idx \notin \{ \ idex \ \ of \ \ a[group][(n - absent)..(n - 1)] \ \ and \ \ a[group][n..(array \ \ \_len - 1)] \ \}\} \qquad (3.2.5)$$

For formula 3.2.5, *idx* means the index of the n numbers array which is calculated by the hash function we proposed before. If the *idx* value is larger than (*n*-1), we set *idx* to 0, and increase it by 1 until that index is not selected.

$$a[group][0 \cdots (array_{len} - 1)] = a[group][0 \cdots (n-1)] + a[group][n, arraylen - 1] \qquad (3.2.6)$$

The array a is divided into two parts. The first part is from 0 to (*n*-1) and the second part is from n to (*array_len* – 1). As for the part 0 to (*n*-1), we use the hash function to get the selected elements. But for the part n to (*array_len* - 1), we have two constrains, which are the index can't be the index of the last (*n*%(*n-k*)) elements and the index of the elements selected in array *a[group][n..(array_len-1)]*. We can calculate the process by using formula 3.3.6.

$$sets_i = \{divide \ \ each \ \ a[group][0 \cdots (array\_len - 1)] \ \ into \ \ array\_len/(n-k) \ \ groups, and \ \ each \ \ group \ \ has \ \ (n-k) \ \ parameters, i.e., sets_i, i \in [1, C_n^k]\} \qquad (3.2.7)$$

For formula 3.2.7, each set generated by the algorithm comes from each row of the array *a*. For example, *a[0][0..n-1]* can be divided into *n/(n-k)* sets and each set has (*n-k*) numbers.
For formula 3.2.7, we divide each row of the array *a* into *array_len/(n-k)* sets as mentioned in 3.1. E.g. m=45, n=8, k=5. An example of C_8^3 is shown in Fig.12.

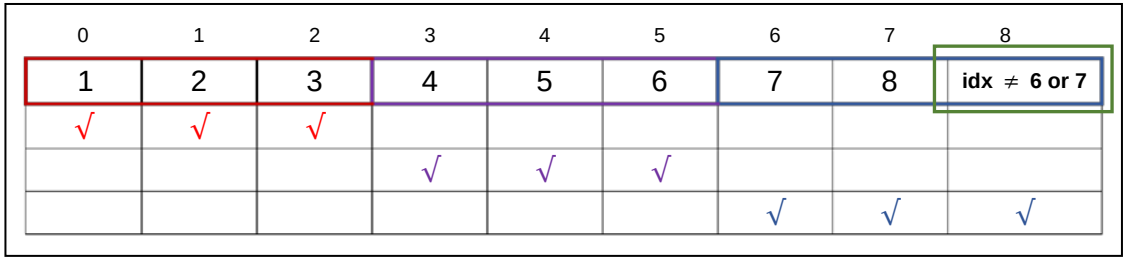

Fig. 12 An example of C_8^3

4. Method of verification

In this part, the process is to calculate how many samples can be covered by the generated sets. If all the samples are covered then we will stop the generation step and stop the system to get the results. And to satisfy the two conditions which are j=s and j≠s, we need two different methods. As for j=s, it means we need to cover all the whole set j. And for j≠s, we just need to cover part of the set j. Detailed methods are explained as below.

4.1 Whole covered

We have to sets which are the combination of C_n^k and the combination of C_n^j, set C_n^k is generated by generation algorithm mentioned in Part I. When a group of sets are generated, our system will calculate which set in C_n^j can be covered by each of them. If the set in C_n^j is not covered yet, we will set that set j as covered and if the current set k is not selected we will set this set k as selected. To realize this function, we make use of the index 0. For set C_n^k, 1 means selected and 0 means unselected. For set C_n^j, 1 means covered and 0 means uncovered. Fig. 13 shows the idea of covering C_n^j samples when j is equal to s, which means whole covered.

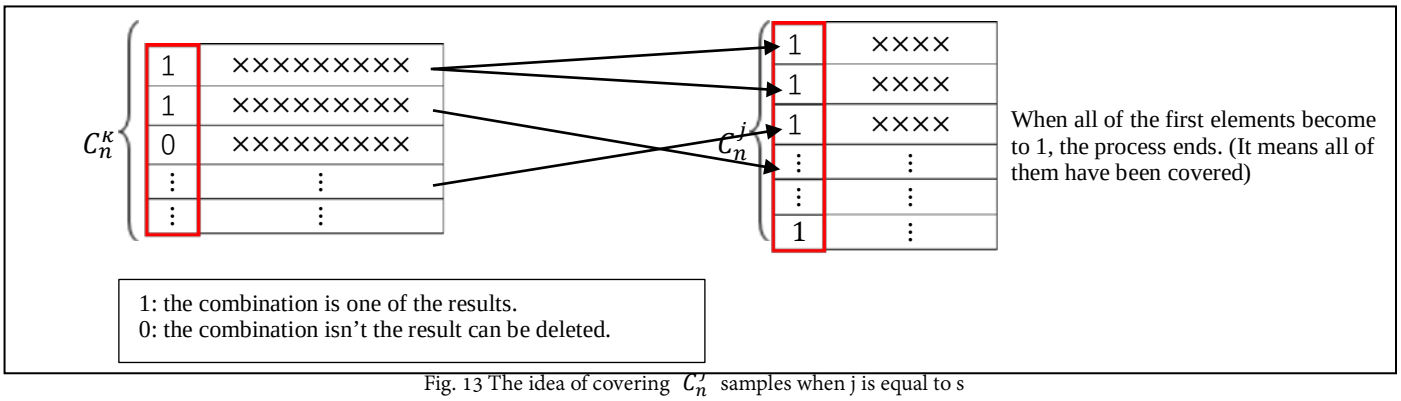

Fig. 13 The idea of covering C_n^j samples when j is equal to s

4.2 Partly covered

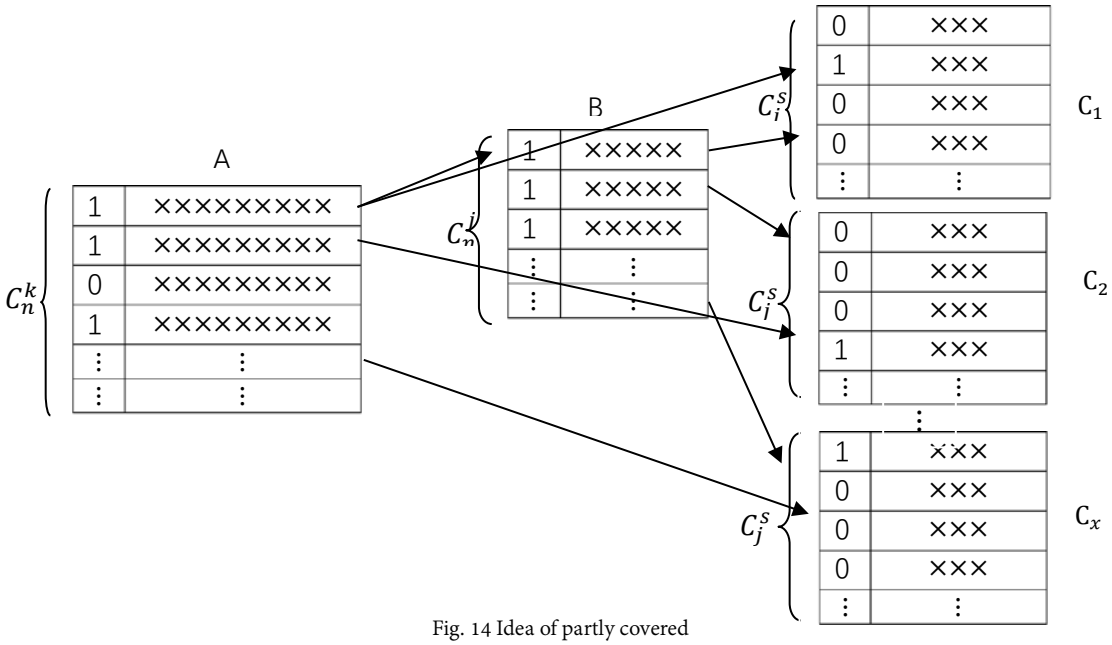

Fig. 14 Idea of partly covered

Since in this condition, we just need to cover set *j* partly, which means we just need to cover *s* elements of *j* elements (*s*<*j*) in each of the set *j*. So, for each of the set *j*, we generate its subsets of C_j^s, and if any one of these sets is covered by current selected set *k*, then this set *j* is set as covered and set *k* as selected. Every time when we do the next verification to see whether next set *j* can be covered, we firstly check the already selected set *k* to see whether any one of these selected can cover this set *j*. If so, we can just set the new set *j* as covered, or we need to find an unselected set *k* to cover it and then set the new set *j* as covered as well as set this unselected *k* as selected. If all the sets of C_n^j are set as covered, the system will stop finding any set *k* and we have already got the answer. By doing so, we can get as least set k as possible.

5. Conclusion and Unsolved Problems

We use the given examples and several other conditions to test our system. We found that our generation algorithm works quite well when the sample of uncovered is big. We can use the least sets to cover most of the samples. However, although we can get the exactly the same optimal answer for the given example 1 and 4, we can't get as optimal answers as the other 5 given examples. After our debugging and analysis, we found out two possible reasons, one is that when the generated sets have already covered

nearly all of the samples and only several samples were not covered. This may be because the probability p1 of covering these uncovered samples is quite low. Meanwhile, the newly generated sets have a high probability p2 of being more similar to the already generated sets. The probability P3 of covering these uncovered sets is obtained by multiplying the above two probabilities. In our assumption, p1 may be 0.2 or even smaller and (1-p2) may be 0.2 or even smaller. So the multiplication of p1 and p2 will be 0.04 which is the value of p3. The probability will decrease dramatically. Another possible reason is that when the amount of n is large, our system is unable to solve too big an integer for the verification loop.