*An example of using the n-Queens Completion program to complete compositions*

*Grigoryan E.*

1. *Generating composition*

In the *Generation\_k\_Queens\_Composition* program, set *n = 100* to view a *100 x 100* chessboard. After starting the program, some arbitrary composition will be formed. Let the result be a composition of *47* queens: *Q(1:100) =*

0 44 16 33 77 0 0 52 30 0

0 0 91 0 66 31 0 0 0 0

0 56 53 59 28 22 39 0 0 0

90 0 0 0 37 0 97 99 24 87

38 49 0 94 0 74 0 0 0 0

65 0 47 0 92 0 0 0 0 7

69 80 0 0 0 0 0 35 88 76

0 29 0 1 18 50 19 0 0 0

68 0 0 10 43 0 0 0 100 61

8 0 0 60 41 0 0 0 0 0

One hundred numbers, which are listed above, consistently characterize *100* rows of the decision matrix. A value of zero indicates that the queen is not set in the corresponding row (the row is free), any other number indicates the position of the queen in the row in question. For example, in the resulting composition, the first element of the array *Q (1: 100)* is equal to *0*, which means that the first row of the chessboard is free; the second element of the array is *44*, which means that in the second row of the checkerboard the queen is located in position *44*; etc.

Let's save the received data in the *kQueens\_Test\_Composition.mat*  file. This name appears at the end of the  *Generation\_k\_Queens\_Composition* program as an example. (Obviously, you can specify any other name).

1. *Completing the composition*

To complete the resulting composition, run the program *Solution\_n\_Queens\_Completion\_Problem*  for execution, where you should specify the name of the input file with the composition. (In this example, we use the name of the input data file: *kQueens\_Test\_Composition.mat* ) As a result, the composition will be completed, and the results will be saved in the file: *nQueens\_Test\_Completion\_Solution.mat*. (This file name is chosen as an example). The resulting solution has the form:

>> Solution \_nQueens\_Completion\_Problem

Input file name: kQueens\_Test\_Composition.mat

The size of a chessboard= 100

Composition Size = 47

Number of free Positions = 53

The first 50 positions of queens:

0 44 16 33 77 0 0 52 30 0

0 0 91 0 66 31 0 0 0 0

0 56 53 59 28 22 39 0 0 0

90 0 0 0 37 0 97 99 24 87

38 49 0 94 0 74 0 0 0 0

Elapsed time is 0.051897 seconds.

Number of complete re-counting cycles = 0

Total number of usage the Back Tracking procedure = 0

Solution is Ok!

The first 50 positions of solution:

58 44 16 33 77 26 63 52 30 34

20 57 91 54 66 31 93 71 55 21

84 56 53 59 28 22 39 95 51 25

90 81 3 75 37 79 97 99 24 87

38 49 11 94 6 74 40 4 12 15

65 36 47 2 92 73 67 27 85 7

69 80 14 89 96 5 83 35 88 76

48 29 13 1 18 50 19 82 32 17

68 70 72 10 43 46 86 78 100 61

8 64 98 60 41 45 9 42 23 62

Solution saved in file: nQueens\_Test\_Completion\_Solution.mat

1. *Verification of the correctness of the solution*

To check the correctness of the obtained solution, we will use the program *Validation\_n\_Queens\_Problem\_Solution* . Here, the input file name is *nQueens\_Test\_Completion\_Solution.mat*. (If another file will be considered, then you must specify the appropriate name.) After starting, the program will display the following message:

>> Validation\_n\_Queens\_Problem\_Solution

Input file name: nQueens\_Test\_Completion\_Solution.mat

The size of a chessboard = 100

Elapsed time is 0.000021 seconds.

Solutions is ok!

1. *Checking the correctness of composition*

As we have already said, the program *Validation\_n\_Queens\_Problem\_Solution* allows not only to check the correctness of the n-Queens Problem solution, but also the correctness of an arbitrary composition.

To check the correctness of the resulting composition, run the program *Validation\_n\_Queens\_Problem\_Solution* for execution, where you should specify the name of the input file with the composition. (Here, as the input data file, the name *kQueens\_Test\_Composition.mat* is specified, which we used to save the composition).

As a result of checking the considered composition, the program displays the following message, confirming the correctness of the composition:

>> Validation\_n\_Queens\_Problem\_Solution

Input file name: kQueens\_Test\_Composition.mat

The size of a chessboard = 100

Elapsed time is 0.000014 seconds.

Composition size = 47

Composition is ok!

1. *Completing a "complex" composition*

Let's look at another example. Let, as a result of generation, a composition is obtained that has the following form: *Q(1:100) =*

0 91 10 0 29 39 59 77 26 2

95 20 94 58 64 22 83 76 34 4

37 0 51 25 0 38 0 50 68 0

0 87 19 21 92 8 96 75 48 0

60 93 46 100 24 66 23 9 53 80

65 0 85 45 28 61 31 89 18 54

0 3 0 0 27 32 0 13 15 0

82 0 70 1 98 47 73 0 62 49

79 36 41 74 0 17 55 90 0 78

0 44 11 16 14 56 0 7 12 99

The composition here consists of *80* queens. The check shows that the composition is correct .. When the  *Solution\_n\_Queens\_Completion\_Problem* program starts, the calculations take a little longer than usual, but as a result the program displays the following message:

>> Solution \_nQueens\_Completion\_Problem

Input file name: kQueens\_Test\_Composition.mat

The size of a chessboard = 100

Composition Size = 80

Number of free Positions = 20

Elapsed time is 2.187818 seconds.

falseNegSimCount = 5

Total number of usage the Back Tracking procedure = 5474

Columns 1 through 10

30 91 10 72 29 39 59 77 26 2

95 20 94 58 64 22 83 76 34 4

37 69 51 25 5 38 81 50 68 43

84 87 19 21 92 8 96 75 48 40

60 93 46 100 24 66 23 9 53 80

65 97 85 45 28 61 31 89 18 54

71 3 33 57 27 32 88 13 15 52

82 67 70 1 98 47 73 42 62 49

79 36 41 74 63 17 55 90 6 78

35 44 11 16 14 56 86 7 12 99

Solution saved in file: nQueens\_Test\_Completion\_Solution.mat

As we can see, the algorithm tried five times from the very beginning to complete this composition (*falseNegSimCount = 5*), and only on the sixth attempt did it succeed. During this time, the Back Tracking procedure was performed more than *5000* times.

The complexity of this composition is that *80* queens are located in it in such a way that not only *80* columns are closed in the remaining free rows, but also most of the remaining free positions. This is the result of the work of the diagonal constraints, which are formed by the previously set queens. During completion, at the last steps, a situation arises when one free position remains in two or three free rows and an attempt to place the queen in any of them will immediately close the free position in another row (due to diagonal restrictions). This is a rather rare and interesting example.

We call such compositions that cannot be completed as negative. Below is an example of negative composition: *Q (1:100) =*

21 31 89 20 50 46 28 39 78 66

57 45 37 52 41 100 92 8 29 19

58 75 6 25 96 34 71 95 15 23

79 73 68 88 67 72 35 98 86 1

83 12 24 44 32 61 14 70 99 80

60 47 81 77 7 59 2 38 0 13

17 51 90 55 49 16 22 26 11 0

76 18 91 87 63 93 9 0 64 94

4 0 56 0 10 5 30 85 65 97

42 40 0 69 54 0 0 62 27 3

After starting the program, the following message is displayed:

>> Solution \_nQueens\_Completion\_Problem

Input file name: kQueens\_Test\_Composition.mat

The size of a chessboard = 100

Composition Size = 92

Number of free Positions = 8

Elapsed time is 0.148292 seconds.

falseNegSimCount = 10

Total number of usage the Back Tracking procedure = 10084

This composition cannot be completied!

The error of such conclusion is less than 0.00001

As we can see, the algorithm tried ten times from the very beginning to complete this composition. At the same time, in aggregate, the *Back Tracking* procedure was used more than *10,000* times to form a solution. But all these attempts were unsuccessful. Therefore, taking into account that the permissible limit of options for finding a solution has been exhausted), a decision is made that with a probability of *0.00001* this composition cannot be completed.

If we print the number of free positions in the remaining rows, then each time the result will be something like this:

*Step-93:*  1 1 1 1 1 2 3

*Step -94:* 0 1 1 1 1 1

or

Step -94: 0 0 1 1 2 3

Or options close to this. Whichever free row we choose to position the queen, it will close the last free position in one of the remaining free rows.

1. *An example of a large-size composition*

Let's generate a composition for a chessboard, the size of which is *100 \* 106*. Let the size of the composition be equal to *7 888 826*. Here are the first *50* values of the array of values of queens' positions:

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 9002222 0

0 0 0 0 0 0 0 13573604 0 0

84739423 0 0 0 0 0 39546789 0 0 0

21295044 99655382 0 0 0 0 0 0 0 0

- The time required to generate such a composition was *17.59* seconds (all calculations were carried out on *DeskTop-13*, the configuration of this computer is described in the publication in *arhiv.org:* <https://arxiv.org/abs/1912.05935> ).

- The time it took to complete the resulting composition was *384.51* seconds. Here are the first *50* consecutive values of the *1-dimensional* solution array:

89582163 60368706 59234218 20261866 86935797 30224854 77724126 36762224 45267928 73444501

30441569 40452963 29233166 74466403 31027892 16431542 83945356 25649731 9002222 26312393

99907709 4403090 19464886 43088993 92254324 37208851 91076429 13573604 48390687 50900879

84739423 22553311 18099114 61465930 76001929 56522925 39546789 31942742 33849160 75874627

21295044 99655382 79211814 74172179 90416939 47905163 37217953 24667656 80381835 35026191

- The time to check the correctness of the solution obtained was *15.71* seconds.

>> Validation\_n\_Queens\_Problem\_Solution

Input file name: kQueens\_Test\_Composition.mat

The size of a chessboard = 100000000

Elapsed time is 15.712291 seconds.

Solutions is ok!

1. *On the linearity of the algorithm of completing compositions*

All programs keep track of the running time of the algorithm, and upon completion, the corresponding result is displayed. Let us assume that a large number of experiments were carried out with different compositions (both positive and negative) for different values ​​of *n*. Let us determine the average time to complete each composition for each value of *n*. As a result, we get a sample of average completion times for each value of *n*. If we divide the average value of the completion time by the corresponding value of *n*, we get the "*one row processing time* ". This is the average time it takes for the algorithm to place the one queen on one row. If the considered algorithm is linear in time, then with increasing value of *n*, the *one row processing time* should not change. Within a small error, the given time should remain constant.

For a quick evaluation of the algorithm, it is enough to generate *20-30* compositions for a list of values *​​n = (100, 1000, 10,000, 100,000, 1,000,000, 10,000,000).* You can get more accurate results if build and test very large samples of compositions for different values ​​of *n*. In the course of the research, samples of *100,000* compositions were generally formed for various values ​​of *n*. Where the total counting time increased significantly, the sample size decreased correspondingly. For *n = 1000*, for the study, a sample was formed, consisting of one million compositions.

The study examined the range of chessboard size values from *7* to *one hundred million*. For values *n = (7, ...,99)*, the algorithm disables some procedures. This is due to the fact that for a given range of *n* values, it is necessary to more carefully select the indices of free rows and indices of free positions in the selected rows. Figuratively speaking, we can say that this zone is quite turbulent and the calculation algorithm in this section (which is *0.000001* part of the entire interval) differs in efficiency from the main algorithm.