Questions & Answers

*To the project:* ***Linear algorithm for solution n-Queens Completion Problem***

***Grigoryan E***.

***1.In the comments to the program*** ***source code quite often it is said about the decision matrix, but in the program there is no such matrix. What is the reason for this?***

Answer. In fact, quite often one can come across the term “decision matrix”, which is used as an analogue of a chessboard. When solving a problem, all events unfold on the decision matrix. It’s like a theater stage. Choosing a row for the location of the queen, or searching for a free position in any row, we always focus on the decision matrix. When we keep track of the indices of the remaining free rows and the number of free positions in these rows, we also focus on the decision matrix. In the same way, we focus on the decision matrix when we take into account the diagonal constraints that are imposed on other free cells when the queen is placed in some cell. All events on the queen's location in the selected position and the consequences of these events are recorded on the decision matrix. However, in the main part of the program, a decision matrix is ​​not created and is not used, since this is not necessary. Moreover, for large values ​​of **n** a large amount of RAM would be required to create such solution matrix (for example, for a chessboard of **106 x 106** size, **1012** bytes of memory would have to be allocated). Only after the most of the queens is placed on the chessboard, we create a smaller solution matrix, where we “projectionally” reduce the remaining free rows and columns. (For example, for **n** = **106**, at the last stage of solving the problem, a decision matrix of size **547 x 547** is created only after the **999453** queens are located in arbitrary positions). This approach does not load much memory, since this is just a small part of the decision matrix. However, this gives us an advantage in the formation of the algorithm and a gain in solution speed.

***2. What is the meaning of the basic level, and how are they calculated?***

Answer. The possible set of branches for finding the solution of this problem can be divided into two subsets. One of them is a subset of the branches that deadlocked. Therefore, when a deadlock occurs in the process of solving the problem, we must go back (i.e., perform the **Back Tracking (BT)** procedure) to one of the previous levels, and again build the solution, starting from this level. In this regard, two questions arise:

 1) How many levels to go back should be?

 2) To which of the previous levels should we return?

Hereinafter, by the level we mean the number of queens correctly placed, regardless of the sequence of their arrangement.

Obviously, executing the **BT** procedure is an additional load for the program:

- for all the return levels, we need to save copies of all the necessary arrays and variables;

- based on the stored copies, it is necessary to restore the values ​​of active arrays and variables;

- it is necessary again continue to form a new solution from this level, in the hope that we will be “lucky this time”.

Obviously, the presence of deadlocked branches of the search is an objective property of this problem. And since we are forced to go back, we should choose the optimal level, given that:

- the more often the **BT** procedure is performed, and the further the level is located to go back, the more time it will take to solve the problem;

- the closer to **n** the return level, the less likely that solution from this level will be productive .

In this algorithm we form three levels to go back if the formed branch of the solution is deadlocked: **startEventBound, eventBound1** and **eventBound2**. We call these levels **basic**. Here **startEventBound** is the initial base level, which corresponds to the beginning of the problem solution. This is the level when, after entering data, the correctness of the composition was carried out and the necessary operations were performed to begin the search for a solution.

*How the baseline values were ​​obtained?*

Consider a list of chessboard size values ​​for modeling: **n** =(*10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 500, 800, 1000, 3000, 5000, 10000, 30000, 50000, 80000, 105 , 3 ∗ 105 , 5 ∗ 105 , 106 , 3 ∗ 106 , 5 ∗ 106 , 107 , 3 ∗ 107 , 5 ∗ 107 , 8 ∗ 107 , 108* ), which will be called the “**base list of n values for computer simulations**”. For each value **n** from this list, we perform the following operations:

1. For **n x n**-sized chessboard, we find the **n-Queens Problem** solution using only the **randSet & randSet** solution search algorithm. As a result, with a high probability a deadlock branch will be formed, where **k** queens will be correctly placed on the chessboard. We will form a large sample of such solutions. Define the average sample value (**xMean**) and standard deviation (**xStd**). Define the value **baseLevel1 = xMean – 3\*xStd**. Consider the obtained value of **baseLevel1** as an approximate value of the base level 2. Let us carry out similar calculations for the base list of **n** values and store the results in the array **baseLevelAr1**.

2. We will carry out similar calculations for the entire base list of **n** values, but, we will use only the **rand & rand** algorithm to solve the **n-Queens Problem**. Save the results in the **baseLevelAr2** array. Thus, we obtain approximate values ​​of two base levels for the base list of **n** values.

3. Further, it is necessary to optimize the obtained values ​​of the base levels by changing their values ​​with a certain step up or down. The selection optimality criterion will be such values ​​of **baseLevel1** and **baseLevel2**, in which the problem solution for a large sample of compositions leads to a minimum number of **False Negative** solutions and a minimum average counting time. We save the values ​​of the base levels optimized in this way for the base list of **n** values ​​in the arrays **eventBoundAr1** and **eventBoundAr2**.

4. Conduct a regression analysis to determine the dependence of the value of the **baseLevel1** on the value of **n**. Similarly, we will conduct a regression analysis and establish the dependence of the value of the **baseLevel2** on the value of **n**. Thus, we get the opportunity to determine the values ​​of the base levels for an arbitrary value of **n**.

In the **n-Queens Completion Problem** program, after entering the source data array, the size **n** of the initial chessboard is determined, and based on it, using the regressions, the values ​​of the base levels **eventBound1** and **eventBound2** are determined.

***3. What is the meaning of the rules of “minimal risk” and “minimal damage” that are used at the last stage of problem solving.***

Answer. These two rules are of fundamental importance for the operation of the algorithm, in particular, the “minimal risk" rule. Both rules are described in detail in the publication: *Grigoryan E., Linear algorithm for solution n-Queens Completion problem,* [*https://arxiv.org/abs/1912.05935*](https://arxiv.org/abs/1912.05935)

The rule of minimal risk. Suppose that in the process of solving we placed most of the queens on the chessboard, and only a small number of free rows remained on the chessboard. If we determine the number of free positions in each of the remaining free rows, then it may turn out that among them there is a row in which only one free position remains, and all other positions in this row are closed. (In the process of solving, the less free rows remain, the greater the likelihood of a similar situation). If, at this step of solution, select not the row in which there is only one free position, but any other, then such a choice will lead to great risk. The reason is that the diagonal restrictions associated with the choice of such a position can close the only free position in the risk row, and thereby lead the solution to a deadlock. To eliminate the risk of a similar situation, at each step we find a row with a minimum number of free positions, and it is there that we choose a position for the queen. If, in the list of free rows, there are two or three rows with the same minimum value of the number of free positions, then we randomly select one of these two (three) rows.

The rule of minimal damage. After we have selected the row index, according to the minimum risk rule, we have to choose a position in this row for the location of the queen. If there is only one position left in the row, then we select it. If there are two or more positions left in the row, then we select the position that, due to diagonal restrictions, causes minimal damage to the free positions in the remaining free rows. If it turns out that two positions in the selected row cause the remaining positions the same minimal damage, then the index of one of these two positions is randomly selected.

***4. Why is the program taking into account the total number of cases when the BT procedure is applied?***

Answer. Taking into account the number of cases when the **BT** procedure is used (the number of cases when the search branch leads to a deadlock), it is important for the program to work. In the process of problem solving, all cases are taken into account when the search branch for the solution is interrupted and a return is made to one of the basic levels. When the total value (**totSimCount**) exceeds the specified threshold value (**totSimBound**), the calculations are interrupted and the completion is repeated again. The maximum number of such repetitions is **10**. If, as a result of these actions, it is not possible to obtain a solution, then a decision is made that the composition is negative. It is **totSimCount** that is the critical indicator on the basis of which a decision is made to interrupt the calculations and repeat it again.

A useful feature of the **totSimCount** counter is that it allows us to determine how efficiently the algorithm works. If, during the completion, a deadlock never arises, then this means that the algorithm is correct. These are the unique solutions that we can talk about, that the algorithm “solves the non-deterministic problem deterministically”. Therefore, it is obvious that, ceteris paribus, the smaller the number of cases of application of the **BT** procedure, the more effective the operation of this algorithm.

***5. If the composition turns out to be negative, the program displays a message: “This composition cannot be completed. The probability of error of such a decision is less than 0.0001. " How was obtained this value?***

Answer. At the beginning, it is important to note that such a message for negative compositions appears only if the size of the chessboard is **n < 100**. In this interval, in fact, the error in making such a decision is less than **0.0001**. If **100 < = n < 800**, then the decision error that the considered composition is negative is less than **0.00001**, and for values **n > = 800**, the corresponding decision error is smaller than **0.000001**.

The algorithm is constructed in such a way that **False Positive** solutions impossible, i.e. if a decision is received, then it cannot be wrong, since the control of the correctness of the selected position is checked at each step. However, the algorithm does not exclude the possibility of the appearance of **False Negative** solutions, i.e. compositions that are positive, but the program, after many attempts, makes an erroneous decision and decides that the composition is negative.

*What is the probability of getting False Negative solutions in this program?*

In order to answer this question, a fairly voluminous computational experiment was performed.

1. Consider a basic list of different sizes of a chessboard for computational simulations.

2. For each **n** value from this list, we will form and save large samples of **n-Queens Problem** solutions.

3. Based on each sample of **n-Queens Problem** solutions, we will form and save a large sample of random compositions. For *n = (20, 30, ..., 90, 100, 200, 300, 500, 800, 1000, 3000, 5000)*, the sample size of compositions was equal to **100,000**, and further, the sample size decreased with increasing **n**.

Obviously, each such composition can be completed at least one way, since each composition is part of some kind of complete solution.

4. Consider each sample of compositions, and run the program to complete each composition. In the process of analyzing each sample, we determine the total number of cases when the program generates **False Negative** solutions.

5. As a result of the analysis of all data, it was found:

a) The program successfully solves the problem of completing almost all compositions.

b) In the range of values ​​**7 < = n < 100**, the program failed to complete some compositions and the share of such **False Negative** solutions in the considered samples was less than **0.0001**.

c) In the range of values ​​**100 < = n < 800**, the share of **False Negative** solutions in the considered samples was less than **0.00001**.

d) In the range of values ​​**n > = 800**, all compositions were completed, and there was not a single **False Negative** solution. However, it is obvious that this does not exclude the possibility of the appearance of **False Negative** solutions with a multiple increase the sample size. In any case, in the range of values ​​*n = (800, 1000, 3000, 5000, 10000)*, the value of the decision error that the considered composition is negative will be less than **0.000001** and, with an increase in the value of **n** this error will decrease.

In the process of computational simulations, we considered various boundary values ​​of the total number of applications of the **BT** procedure. The most optimal value was **1000**. Moreover, (this is important), if after applying the **BT** procedure thousands of times the composition cannot be completed, then a second attempt is made to complete the composition (from the beginning). The number of such attempts equals ten. Increasing this value will increase the time after which the message that the composition is negative will appear, and the proportion of **False Negative** solutions will slightly decrease. If we decrease this value, then the decision-making time will correspondingly decrease and the share of **False Negative** decisions will slightly increase.

Thus, on the basis of this series of computational experiments, the following rule was established: “if, during the analysis of an arbitrary composition, the **BT** procedure is used **1000** times and the composition cannot be completed, then this procedure is repeated from the very beginning. The total number of such repeated calculations is **10**. If as a result of these actions it is not possible to obtain a solution, then such a composition is considered negative, that is, such a composition cannot be completed. The probability of error when classifying a positive composition into a group of negative compositions, i.e. the probability of a **False Negative** solution depends on the size of the chessboard **n**:

**7 < n < = 100**, then the decision error value is less than **0.0001**,

**100 < = n <800**, the decision error value is less than **0.00001**,

**n > = 800**, the value of the decision error is less than **0.000001**

Obviously, this error only applies to positive compositions. Any negative composition, the program determines with a **100%** guarantee.

Selecting **1000** as the maximum allowable number of re-applying the **BT** procedure, and repeating this procedure **10** times, form the error rate for **False Negative** solutions (*0.0001, 0.00001, 0.000001*). This is the result of a trade-off between the speed of the program and the values of the decision error. If there is a need to reduce the probability of an error in the formation of False Negative solutions in the range of values ​​**7 < n <= 800**, then instead of the indicated values, we can consider a much larger number. Then, when processing a large sample of arbitrary compositions, only the processing time of negative compositions will increase, and those positive compositions that the program will mistakenly classify as negative compositions. The processing speed of all other compositions will remain the same.

It is important to note that in the previous version of the program on the basis of which the research was carried out and the results were published in *arhiv.org*, the number of repeated solutions was **3**. Here, the number of re-decisions is **10**. The boundary value of the number of re-use of the **BT** procedure remains the same and is equal to **1000**.

***6. How is the event index determined in the program?***

Answer. The event index depends on a number of indicators:

- chessboard size (**n**),

- composition size (**k**),

- from the result of comparing these values with:

a) fixed values **nFix1 = 50**, and **nFix2 = 100**,

b) and the boundary values **eventBound1** and **eventBound2**, which are calculated based on the regression equation for a given value of **n**.

For clarity of presentation, we select three sections on the numerical axis of the natural series

**1**  **2** **3**

- -! - - - - - - - - - - - - !- - - - - - - - - - - - - - !- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

! ! !

**7** **50** **100**

1. **7 <= n <50, eventInd = 4**

Here, the calculations start from the **4th** preparatory block and then are performed in **block 5**.

1. **50 <= n <100, eventInd = 2**

Here the calculations start from the 2nd preparatory block and then are performed in blocks **3, 4, 5**

1. **n > = 100**,

Here, the choice of the event index value depends on the ratio of the composition size (**k**) and the boundary values of the **eventBound1** and **eventBound2** event blocks:

1. if  **k < eventBound1**, then e**ventInd = 1**;
2. if **eventBound1 < = k < eventBound2**, then **eventInd = 2**;

c) if **k > eventBound2** then **eventInd = 4**.

It can be noted that the event index does not take on the values **3** or **5**. This is due to the fact that in order to go to block **3**, we must first perform a small number of preparatory operations in block **2**, and after that, make a transition to block **3**. A similar situation with a bunch of blocks **4** and **5**. Here, as in the previous case, before going to block **5**, it is necessary to perform preparatory operations in block **4**.

***7. Can this algorithm be used for values n> 100 000 000?***

Answer. During the research, the algorithm was developed for a wide range of values ​​for the chessboard size (**n**): from **7** to **100** million. In fact, the upper limit of the interval indicated here is of no fundamental importance. It was chosen only for the convenience of modeling, taking into account the time costs associated with the formation and analysis of samples for large values ​​of **n**. For each user, the value of the upper limit can be limited only by the size of the computer's **RAM**. On a computer with 3**2 G**B of RAM, the calculations were carried out for a chessboard of **n=800 million** or more.

On the lower boundary of the interval. Although the value **7** is indicated as the lower boundary of the interval, those who like to “press the red button” can carry out calculations for the values ​​n = (**5**, **6**), the program allows such values.

***8. How to evaluate the effectiveness of the obtained algorithm?***

Answer. Here are some important indicators that are used to assess the effectiveness of any algorithm:

1. The algorithm should work quickly, i.e. the time for solving the problem should be minimal.

2. It is important that the algorithm is linear in time **O (n)** (or close to that).

3. The algorithm should work reliably in a wide range of changes in the basic parameters of the problem being solved.

4. The algorithm should be designed in such a way as to work “freely” within a reasonable amount of RAM (for example, 32 GB), that is, there should be no additional operations associated with a lack of RAM.

5. Everything that is needed to solve the problem, the algorithm "must find itself", without requiring additional actions from the user.

These are characteristics that are obvious to every programmer and can be supplemented with something else. However, these are external attributes of the program. It is important for us to determine the internal criterion for the operation of the algorithm that makes it effective. In non-deterministic problems, one of the important criteria is the number of cases of using the **Back Tracking** (**BT**) procedure in the process of problem solving. The fewer **BT** procedures, the better. It would be ideal if the **BT** procedure was never used at all during the solution process. This will mean that by sequentially choosing some free row and some free position in this row, we reach the goal without ever making a critical error - all the steps performed are correct. In this sense, the proposed algorithm is quite effective. For example, for **n = 1000**, in the experiment where **one million** compositions were assembled, in **60.5 0**% of cases the **BT** procedure was never used, and in the remaining solutions - in **22.72**% of cases the BT procedure was used only once, in **9.21**% of cases the procedure **BT** was used two times, and in **3.95**% of cases the **BT** procedure was used three times. Together, this amounts to **96.38**%. The remainder, which is **3.62**%, used a different number of **BT** procedures.

The described performance of the algorithm is not related only to the value **n = 1000**. A similar pattern is typical for a wide range of **n** values. For this reason, the speed of this algorithm is quite high. For example, for the considered value **n = 1000**, the average completion time for one arbitrary composition is **0.035 seconds**. And as mentioned above, on a computer with **32 GB** of RAM, the algorithm allows calculations for a chessboard of **800 million** or more. Therefore, in general, according to the above criteria, the algorithm can be considered quite effective.