% Solution\_ nQueens\_Completion \_Problem

%{

The program is designed to complete an arbitrary composition of *k* queens

to a complete solution. By composition we mean the random distribution of *k* queens on arbitrary chessboard with size *n x n*, such that three conditions of this task are fulfilled: *in each row, in each column, and also on the left and right diagonals passing through the position where the queen is located, not more than one queen is located*. It is necessary to find at least one solution and thereby show that the solution exists, or derive a judgment that with a given probability (*P*), this composition can’t be completed.

%}

%{

License: Attribution-NonCommercial-ShareAlike  
CC BY-NC-SA – “This license lets others remix, adapt, and build upon your work non-commercially, as long as they credit you and license their new creations under the identical terms”.

%}

%{

Project author and developer - Grigoryan Eros (EricGrig), 2020

I will be glad if any sections of the code, or the entire program as a whole, will be used for scientific purposes, or for education. At the same time, I will be grateful if you consider it possible to refer to my publication. It is an element of culture and a sign of mutual respect.

For commercial use of any part of the program code, or the entire program as a whole, the written consent of the author is required.

%}

%{

The research results related to the development of this algorithm are published in arxiv.org in article: *Grigoryan E., Linear algorithm for solution n-Queens Completion problem, <https://arxiv.org/abs/1912.05935> .* It will be correct if you first read this publication before begin to analyse the source code. This will make the program description more transparent and reduce the number of possible questions.

The Russian version of the article is published on the habr.com programmer community website: [*https://habr.com/ru/post/483036/*](https://habr.com/ru/post/483036/)

%}

%{

1.The beginning

---------------

How is prepared the initial data?

---------------------------------

Denote the chessboard side size by *n*.

Let there be a one-dimensional nullified array of size *n*. If in the i-th row

of the chessboard the queen is placed in position j, then, respectively,

in the i-th cell of the data array the value of j is written.

Next, along with the name "*chessboard with size n x n*" we will

use the name "*solution matrix of size n x n*"

Let's read to array *Q* the data file with initial composition

Here, as an example, we used *kQueens\_Test\_Composition.mat* for the data file

name. This name must be replaced with the name that matches your file

of data

%}

inputFileName= 'kQueens\_Test\_Composition.mat';

iInfo=['Input file name: ' inputFileName];

disp(iInfo);

% Input data file

Q=importdata('kQueens\_Test\_Composition.mat');

n=length(Q);

%{

Output on display the first 50 lines of the composition, or the entire

composition, if n <50

%}

if n<50

nDisp=n;

else

nDisp=50;

end

disp(Q(1:nDisp));

%{

We define the number of zero cells in the array *Q*, which we denote by

*nZero*. Thus, we determine the number of free rows in the solution matrix

%}

nZero=sum(Q==0);

% Denote the size of the composition by *nComp*

nComp=n-nZero;

%{

We display for the user the necessary information about this task:

solution matrix size, composition size, number of free positions

%}

disp(' ');

tStr = sprintf('The size of a chessboard = %d',n); disp(tStr)

disp(' ');

tStr = sprintf('Composition Size = %d',nComp); disp(tStr)

tStr = sprintf('Number of free Positions = %d',nZero); disp(tStr)

%{

Output on display the first 50 lines of the composition, or the entire

composition, if n <50

%}

tStr = sprintf('The first %d positions of queens:',nDisp); disp(tStr)

disp(Q(1:nDisp));

%{

If it turns out that the size of the composition is equal to the size of a

chessboard, then we display the appropriate message and interrupt the program.

%}

if nComp==n

tStr = sprintf('Composition size the same as matrix size %d',nComp);

disp(tStr);

pause

exit

end

%{

If it turns out that the composition size is zero, i.e. no composition,

then we display the appropriate message and interrupt the program.

%}

if nComp==0

tStr = sprintf('Composition size =0. No composition!');

disp(tStr);

exit

end

%{

General

-------

In the research process, for solution the problem, three main algorithms were developed, which differs both in speed of solving the problem, and efficiency. The program also implements sections of code, that perform preparatory functions for the basic algorithms.

Denote these sections of the code, respectively: Block-0, Block-1, ..., Block-5.

*Block- 0* -- starting block. Checking the correctnes of composition, preparating

! control arrays

!

*Block- 1* -- forming a solution based on an algorithm *rand\_set & rand\_set. The b*oundary

! value of the number of correctly established queens equal to *eventBound2*

!

*Block- 2* -- preparatory operations for the transition to Block- 3

!

*Block- 3* -- forming a solution based on an algorithm *rand & rand .* The boundary value

! of the number of correctly established queens is *eventBound3*

!

*Block- 4* -- preparatory operations for the transition to Block- 5

!

*Block- 5* -- formation of a decision based on the rules of "*minimal risk*" and

"*minimal damage*"

In the process of finding a solution, the calculation is transferred from one block of the algorithm to another, like a relay stick when running.

The algorithm was developed for a fairly wide range of chessboard size: from *7* to *100,000,000*. If RAM size allows, then it is possible to carry out calculations

for large values of chessboard size, for example, for *n = 800,000,000*.

(On a computer with *RAM = 32 Gb*, the completion problem was solved for *n = 1,000,000,000*. However, due to the fact that there was not enough memory, we had to slightly change the program and carry out calculations in two stages).

%}

if n<5

tStr = sprintf('The algorithm was developed for the values n > 7');

disp(tStr);

exit

end

%{

If *nComp <= eventBound2*, then the calculations begin with Block-1.

If *eventBound2 <nComp <= eventBound3*, then the calculations begin with Block-2.

If *nComp> eventBound3*, then control is transferred to Block-4, where

preparatory work is carried out and after that a transition is made

in Block-5 for basic calculations.

%}

%{

As the results of the study showed, in the range of values *n = (7, ..., 99)*

requires a more meticulous approach for formation the branch of search. Therefore, this interval was divided into two subintervals *(7, ..., 49)* and *(50, ..., 99)*, in each of which, the corresponding algorithm is used. (Here the boundary values can be slightly increased, or reduced. From this, the essence of the solution will not change)

%}

% *nFix1, nFix2* - Фиксированное значение размера матрицы решения.

nFix1=50;

nFix2=100;

%{

Если *n < nFix1*, то выполнение расчетов передается в Блок-4.

Если *nFix1 <= n < nFix2*, то выполнение расчетов передается в Блок-2.

%}

%{

About the boundary values of eventBound2 and eventBound3:

---------------------------------------------------------

If in the solving process the search branch leads to a deadlock, then we need to go back to one of the previous levels (*Back Tracking*), and start to construct new solution. To do this, we need to know on which of the previous levels we should return to, since we must first save the values of the main parameters of this level. Choosing the optimal return point is a rather complicated and interesting task. In this algorithm, we use the following rule. Along with the initial level, which corresponds to end of the composition validatio and formation all control arrays, we form and use two more base levels to return back, with the boundary values *eventBound2* and *eventBound3*. Here, the *accounting measure of the solution*

*level is the number of queens correctly placed on the chessboard*.

%}

%{

On the number of recalculations at the last basic level

-------------------------------------------------------

The biggest difficulties in the operation of the algorithm arise at the

last stage of solving the problem. All hidden errors that were made when

choosing the index of a free row, and (or) choosing a free position in this row,

"gradually accumulate", and at the last stage they are manifested in the fact

that among the remaining free rows, there is at least one row in which there

is no free position. It means a deadlock. Therefore, the algorithm of forming the branch search at the last stage is more meticulous. (As an analogy, here it is appropriate to use a comparison of microsurgery with conventional surgery).

In an effort to take into account possible effective ways of forming a branch

of the search, at the last stage, inside two nested cycles, we execute the third cycle, which is repeated several times, with a return to the beginning of the same cycle, without changing the parameters of two external cycles. This is similar to applying the *Back Tracking* procedure inside a nested loop system. The threshold value of the number of times that can be produced inside this cycle is denoted by *repeatBound*. Further, in the text, this will be discussed in a little more detail.

%}

%{

We compute the values of *eventBound2* and *eventBound3*, as well as the threshold value for the number of repetitions *repeatBound.*

%}

if n<nFix1

repeatBound=25;

else

repeatBound=5;

u=log10(n);

w=u\*u;

if n<30000

b2=12.749568\*w\*u -46.535838\*w + 120.011829\*u -89.600272;

b3=9.717958\*w\*u -46.144187\*w + 101.296409\*u -50.669273;

else

b2=-0.886344\*w\*u+56.136743\*w-146.486415\*u+227.967782;

b3=14.959815\*w\*u-253.661725\*w+1584.711376\*u-3060.691342;

end

eventBound2=n-round(b2);

eventBound3=n-round(b3);

end

%{

The empirical values of the *eventBound2* and *eventBound3* parameters were established on the basis of a very large number of computational experiments and were optimized for the wide range of the chessboard size. For any smaller range of *n* values, it is possible to change slightly these parameters and obtain values at which the program will work a little faster.

In the process of solving the problem, if a deadlock occurs, then some blocks of the algorithm are re-executed. Moreover, depending on the values of *n* and *nComp*, repeated calculations begin or from the very beginning, or from some achieved  level. If a repeated search at the upper levels does not lead to success, then a repeated search begins at lower levels. Here, the variables *simBound3*, *simBound5* determine the maximum number of repeated calculations within Block-3 and Block-5. *totSimBound* - determines the total number of all repeated calculations at all levels

%}

simBound3=5;

simBound5=100;

totSimBound=1000;

%{

falseNegSimBound - recalculation limit.

If in the first time it is not possible to complete the composition, then

the calculation are repeated from the starting point

%}

%{

falseNegSimCount - Number of complete re-counting cycles of the

considering composition.

This is a counter of the number of recalculations for completing

compositions, that faild to complete the first time

%}

falseNegSimCount=0;

%{

falseNegSimBound - recalculation limit.

If in the first time it is not possible to complete the composition, then

the calculation are repeated from the starting point

%}

falseNegSimBound=10;

%{

For algorithm we use several control arrays:

*A* - to control row indices,

*B* - to control column indices.

%}

A=zeros(1,n,'uint8');

B=zeros(1,n,'uint8');

%{

Also, to control the cells of the diagonal projections, we use two

arrays *D1*(1:n2) and *D2*(1:n2), where n2 is the size of the control arrays

%}

n2=2\*n;

D1=zeros(1,n2,'uint8');

D2=zeros(1,n2,'uint8');

%{

Active event index selection (*eventInd*)

-----------------------------------------

Define the block index from which the program will start. To do this,

we assign the appropriate value to the *eventInd* variable. We also define

a threshold value for the number of repeated calculations (*simBound5*)

at the last stage (Block-5)

%}

if n<nFix1

eventInd=4;

simBound5=totSimBound;

elseif n<nFix2

eventInd=2;

else

if nComp<eventBound2

eventInd=1;

elseif nComp<eventBound3

eventInd=2;

else

eventInd=4;

simBound5=totSimBound;

end

end

tic

%{

3. Verification input composition

---------------------------------

The composition is checked, and the corresponding cells of the control arrays *A,B,* *C* and *D* are filled sequentially.

In the corresponding cells of the array *Q*(i), the column indices of the correctly installed queens are written. The value of the *totPos* variable is incremented, which is used to account for the number of correctly installed queens.

%}

% Define the occupied row indexes in the *Q* array and save the results in the

% *qPosInd* array

qPosInd=find(Q>0);

% Write 1 to those cells of the array *B* that correspond to the occupied columns

B(Q(qPosInd))=1;

% Ffind the sum of units in array *B*

s=sum(B);

%{

Check if two different queens are located in the same column. If so, then there

is an error in the original composition. In this case, we will display the

corresponding message and interrupt the program.

%}

if s~=n-nZero

'Error -- the same positions in different row!'

exit

end

%{

The verification algorithm works as follows: if the cell (i,j) where j= *Q*(i)

is free, taking into account diagonal restrictions and restrictions on the

number of elements in each column, then the queen is located correct in this

cell. We do not check the rule “*no more than one queen in a row*”, since the model for preparing the initial data excludes the possibility of more than one queen in the composition. Each cell in the 1-dimensional input data array characterizes the corresponding row in the decision matrix.

%}

qError=0;

for k=1:nComp

i=qPosInd(k);

j=Q(i);

r=n+j-i;

t=j+i;

if D1(r)==0 && D2(t)==0

D1(r)=1;

D2(t)=1;

else

qError=1;

break

end

end

%{

If an error is detected in the composition, i.e. the location of the queens will not correspond to the conditions of the task, a corresponding message will be displayed, and the program will be interrupted.

%}

if qError==0

A(qPosInd)=1;

totPos=nComp;

else

tStr = sprintf('Error in composition! Row = %d Position= %d',i,j);

disp(tStr)

exit

end

% Let's delete the *qPosInd* array, as we will not use it further.

clear qPosInd

%{

Saving copies of generated arrays for reuse

-------------------------------------------

We did some preparatory work. Organized input data and checked the

composition validity. We saved copies of all arrays, which necessary for procedure Back Tracking. If we return to this level, we will restore all the necessary arrays based on these backups. This level is the initial (zero) basic level, from where formation the search branch of solution begins. Here, the number of correctly installed queens equals the size of the input composition.

%}

if eventInd==1

Ax=A;

Bx=B;

D1x=D1;

D2x=D2;

Qx=Q;

xTotPos=totPos;

end

%{

Set the counters for the number of repetitions of the third (*simCount3*)

and fifth (*simCount5*) levels to zero.

%}

simCount3=0;

simCount5=0;

% *simCount3* will then be used as a switcher in Block-3.

% Zero *totSimCount* - the total count of all repetitions at various levels.

totSimCount=0;

%{

All events unfold inside the *while processInd==1* cycle until a solution for this composition is obtained, or it is established that the solution does not exist with probability *P*. The main criterion for such an assessment is the total number of all repeated calculations(*totSimCount*). In the article, the link to which is given in the commentary at the beginning of this program, is written in sufficient detail about this.

As a result of a large number of computational experiments, for a wide variety of random compositions of arbitrary size *k* and for different values ​​of the size of a chessboard *n*, it was found that if the total number of repeated calculations *totSimCount* exceeds the threshold value of *totSimBound*, and no solution was found, then the composition can’t be completed. The probability of error of such a judgment is *0.0001*

%}

%{

The beginning of formation the branch of the search

----------------------------------------------------

As stated above we consider various blocks of the program as separate events.

There are five such events. Three of them correspond to the main blocks of the program, and two events correspond to program blocks that perform preparatory

functions. We assign the variable *activeEvent* the event index, that currently is active.

%}

activeEvent=eventInd;

%{

We introduce the variable *processInd* – as "switcher" to exit the loop.

The cycle is executed if *processInd == 1* otherwise the execution of the loop is interrupted

%}

processInd=1;

%{

We introduce the variable *compositionInd*. If *compositionInd* == 1, then this will mean that the composition is positive, i.e. can be completed to full solutions.

If *compositionInd* == 0, then composition will be considered as negative, i.e. it can’t be completed to full solution. If *compositionInd* = -1, then composition will be considered as negative "*by birth*". This means that in the input array, among the free rows of this composition, there is at least one row, in which there is not free position(all positions are closed due to bans formed by previously established queens).

%}

solutionInd=1;

% Начало основного цикла

while processInd==1

% The variable *event* serves as a switcher between 5 events

switch activeEvent

case 1

%{

Block-1. Using the *rand\_set & rand\_set* algorithm

------------------------------------------------

In this block we search free row and a free position in this row for position

the queen, until the total correctly set queens will be equal the threshold value (*eventBound2*).

The algorithm that runs in this block is called *rand\_set & rand\_set*. Its essence is as follows. We find the indices of all free rows. Carry out a random permutation of these indices. Similarly, we find the indices of all free columns. Also we spend random permutation of these indices. We will consider pairs indices from these two lists (random row index, random column index). If the cell of the decision matrix corresponding to this pair of indices, does not contradict diagonal restrictions, then we set the queen in this position. In this case we write 1 in the cells of appropriate control arrays *A, B, D1* and *D2*. Total counter correctly installed queens (*totPos*) increases by one.

%}

while totPos < eventBound2

xInd=find(A==0);

nRow=length(xInd);

aInd=uint32(randperm(nRow));

yInd=find(B==0);

bInd=uint32(randperm(nRow));

for k=1:nRow

i1=aInd(k);

i=xInd(i1);

j1=bInd(k);

j=yInd(j1);

r=n+j-i;

t=j+i;

if D1(r)==0 && D2(t)==0

D1(r)=1;

D2(t)=1;

Q(i)=j;

A(i)=1;

B(j)=1;

totPos=totPos+1;

end

end

end

%{

In this block, the positions for the queens are determined quickly. And, although here all positions are determined correctly, however, the overall «picture» of the distribution of queens in the solution matrix is «rude». If we do not stop in some optimal step, then the further construction of the branch of the search is likely to lead to a deadlock. Given the high speed of the *rand\_set & rand\_set* algorithm, based on this block, we go through the maximum path from the value of *nComp* to *eventBound2* values. After this, the program execution is transferred to the next block.

%}

%{

Important! This block is executed only if *n > = 100* and the size of the composition is less than *eventBound2*. As the results of almost two tens of millions of computational experiments showed, for a given value of *eventBound2*, this algorithm always completes composition to the value of *eventBound2*. There has never been a situation where the algorithm is looped and not completed. This is due to the fact that the value of *eventBound2* is not critically large, and there are many different possibilities in order to achieve this level. For this reason, in this stage we excluded control of cycle completion from the algorithm, although this possibility was taken into account in early versions of the program. *The thirst for speed was higher than the logic of embracing almost impossible permissible situations*.

%}

%{

When the number of correctly placed queens(*totPos*) is equal to *eventBound2*, event management is transferred to Block-2.

%}

activeEvent=2;

case 2

%{

Block 2. Preparation of the necessary arrays for work in Block-3

----------------------------------------------------------------

In this block, preparatory work is performed for the transition to Block-3. Its essence is as follows: let the number of remaining free rows be *nFreeRow*. We form an array *L(1: nFreeRow, 1: nFreeRow)* and collect the free position indices of all the remaining rows in it. This means the following: in the original solution matrix, we consider the intersection grid of free columns and free rows. We transfer all such cells on the intersection grid to the projection into a smaller array *L*. In this case, we take into account the correspondence of the indices of the array *L* with the corresponding indices of the original solution matrix.

%}

%{

Find the initial indices of the remaining free rows in the solution matrix and save the results in array *A*.

%}

A=find(A==0);

% Denote the number of free lines by *nFreeRow*

nFreeRow=length(A);

%{

We find the initial indices of the remaining free columns in the solution matrix and save the results in array *B*.

%}

B=find(B==0);

% Obviously, the number of free columns will be equal to the number of free rows

%{

Create an array *L(1:nFreeRow, 1:nFreeRow)* and fill all the cells with one. Further, if the cell *L(p, q)* turns out to be free, then we write zero in this cell instead of one.

%}

L=ones(nFreeRow,nFreeRow,'uint8');

%{

Let's create arrays *rAr* and *tAr* for saving the indexes of correspondence to control arrays.

%}

rAr=zeros(nFreeRow,nFreeRow,'uint32');

tAr=zeros(nFreeRow,nFreeRow,'uint32');

%{

We will need these arrays for equivalent accounting for the indices of free positions in the array *L*, with the corresponding indices of the control arrays *D1* and *D2*.

Based on the information about the remaining free rows and free columns, we write zero into the corresponding free cells of the array L. In the same cycle, we will form arrays of accounting *rAr* and *tAr*

%}

for p=1:nFreeRow

i=A(p);

for q=1:nFreeRow

j=B(q);

r=n+j-i;

t=j+i;

if D1(r)==0 && D2(t)==0

L(p,q)=0;

rAr(p,q)=r;

tAr(p,q)=t;

end

end

end

%{

Back up all the main arrays. We will need them for *Back Tracking*, if it becomes necessary to return to the beginning of Block-2 for repeated calculations.

%}

Ay=A;

By=B;

D1y=D1;

D2y=D2;

Qy=Q;

Ly=L;

rAr\_y=rAr;

tAr\_y=tAr;

yTotPos=totPos;

% We have done the preparatory work. Now we can go to Block-3.

activeEvent=3;

case 3

%{

Block 3. Using the *rand & rand* algorithm

--------------------------------------

In this block, we continue composition completion. Here, another algorithm is used, which is called *rand & rand*. Its essence is as follows. From the list of remaining free rows, a random row index is selected. Within the selected row, from the list of free positions we randomly select one index. If it turns out that the position is free from the diagonal restrictions imposed by all previously placed queens, then the position is considered free and the queen is placed in it.

%}

% Increment the counter of the number of cases when Block-3 is used.

simCount3=simCount3+1;

%{

If it turns out that the number of repetitions(*simCount3)* does not exceed the boundary value of *simBound3*, then we will continue to form solution based on the data collected in the array *L.*

%}

if simCount3 <= simBound3

while totPos < eventBound2

% Define free row indices in array *L* based on array *A*

freeRowAr=find(A>0);

% Define the number of free rows (*nFreeRow*)

nFreeRow=length(freeRowAr);

% Choose a random number(*randNumb*) in the interval(*1, nFreeRow*).

randNumb=randi(nFreeRow);

% From the list of free rows *freeRowAr*, we randomly select row index *selectRowInd*

selectRowInd=freeRowAr(randNumb);

%{

Consider an array *L*. Let us form a list of free position indices (*freePosAr*)

in a row with *selectRowInd*. Define the size of this list (*nFreePos*)

%}

freePosAr=find(L(selectRowInd,:)==0);

nFreePos=length(freePosAr);

if nFreePos>0

%{

If there are free position in the selected row, then we continue the solution. If there are no free positions, this means that the search branch has led to a deadlock. In this case, we must interrupt the execution of the algorithm in this block and return to the previous base level.

%}

%{

If there is free position in the row, then we select a random number(*randNumb*) in the interval (*1, nFreePos*)

%}

randNumb=randi(nFreePos);

%{

After that, from the list of free positions(*freePosAr*), we select the position *selectPosInd* on the base of selected random number *randNumb.*

%}

selectPosInd=freePosAr(randNumb);

%{

We randomly selected the free row index (*selectRowInd*) and randomly selected the free position index (*selectPosInd*) in this row. All these actions were performed within the array *L*. Now, we will restore the original index of the selected position based on array *B* (this is the index that corresponds to the original data matrix).

%}

j=B(selectPosInd);

% We will also restore the original index of the selected row based on array *A.*

i=A(selectRowInd);

% We save the result (queen position in the row) in *Q* array

Q(i)=j;

% We increment the counter of the number of positions occupied by the queen.

totPos=totPos+1;

%{

We write 1 in the *selectRowInd* cell of free rows control array *A* to fix that the corresponding row is closed

%}

A(selectRowInd)=0;

%{

We write 1 in the *selectPosInd* cell of array *B* to fix that the corresponding column is busy.

%}

B(selectPosInd)=0;

%}

Change the corresponding cells of the diagonal control arrays *D1* and *D2* using the real values of the indices (i,j) (which correspond to the original chessboard)

%}

rx=n+j-i;

tx=j+i;

D1(rx)=1;

D2(tx)=1;

%{

In all the free rows of the array L in the *selectPosInd* column, we write 1 (to close the corresponding cells).

%}

L(freeRowAr,selectPosInd)=1;

%{

Important! We work with an array *L*, where all free rows and all free columns from the original “large” data matrix are projected. When we place the queen at the position (i,j) in the initial data matrix, then, at the same time, should be excluded from further consideration: row(i), column(j) and all cells of the data matrix that lie on the left and right diagonals passing through the point (i,j). Above, we excluded the corresponding row and the corresponding column, zeroing the corresponding cells in arrays *A* and *B*. Now, we must by "*projection*" exclude those cells of the array *L* that correspond to diagonal exceptions to the original data matrix. To do this, we use the corresponding equivalent indexes previously stored in the arrays *rAr* and *tAr.*

%}

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

%{

Thus, we performed all the procedural steps associated with the selection of one position (i,j) in the original data matrix for the location of the queen.

%}

else % if freePos>0

%{

If there are no free positions in the row, this means that we have reached a deadlock, so we must close this search branch and go back to Block-3, and again repeat to form soluttion. Before that, we must restore all the necessary arrays based on backups. We increment the counter of total number repeated calculations, since we go back to recalculate.

%}

totSimCount= totSimCount+1;

% Based on the saved copies, we will restore the values of the necessary arrays.

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

% Let's move to the Block-3 for re-counting.

activeEvent=3;

end % if freePos>0

end %while totPos < simBound2

else

%{

If it turns out that the number of repetitions of *simCount3* exceeds the boundary value *repeatBound3* and, at the same time, *eventInd == 1*, then we need to return to base level 1 and build the search branches again. Before that, we need to restore all the necessary arrays that correspond to this return point.

%}

if eventInd==1

% We increment the total counter of the number of repeated calculations.

totSimCount= totSimCount+1;

% Let's restore arrays and transfer control to Block-1.

A=Ax;

B=Bx;

D1=D1x;

D2=D2x;

Q=Qx;

totPos=xTotPos;

% Zero the value of the counter *simCount3.*

simCount3=0;

% Let's go to Block-1.

activeEvent=1;

else

%{

If in this block there were *simBound3* repetitions, and in each case, at some step it turned out that among the remaining free rows, there is a row in which there is’nt a free position, this means that this composition is negative, and it can’t be completed. For this reason, the program should be interrupted. We set the variable *compositionInd* to zero to fix that this composition is negative. We also set the variable *processInd* to zero to interrupt the program

%}

solutionInd=2;

processInd=0;

break

end

end % if simCount3 > simBound3

%{

Upon successful completion of calculations in Block-3, the number of queens correctly located in the solution matrix will be equal to *eventBound3*.

Let's move to the Block-4.

%}

if totPos >= eventBound2

activeEvent=4;

end

case 4

%{

Block 4. Preparation of the necessary arrays to work in Block-5.

-------------------------------------------------------------

We go to Block-4 in three cases:

1) Immediately after completion in Block-3, i.e. if *eventInd* was equal 1 or 2,

2) If the value *n <= nFix1*,

3) If the value *nComp> = eventBound2*.

This block is preparatory, where we perpare the necessary arrays before transition in Block-5. To a certain extent, the operation of the algorithm in this block is similar to the operation of the algorithm in Block-3. Its essence is as follows. Let the number of remaining free rows in the solution matrix be *nRow*. We form an array *L(1:nRow,1:nRow)* and collect the data from all free rows and free columns. The algorithm for generating the array *L* is similar to that used in Block-2. As in Block-2, we will take into account the correspondence of the indices of the array *L* with the corresponding indices of the original solution matrix. The projection translation of a solution from the original matrix to a smaller matrix *L* gives us the opportunity at each step to effectively find row with a minimum number of free positions and significantly reduce the amount of computation. But, no less important is the fact that, based on the array *L*, we simultaneously keep track of the status of all remaining free rows. This allows us to control all the rows and determine whether a situation has occurred when in any of the remaining rows the number of free positions is zero. In this case, we exclude the search branch as deadlock. This approach allows us to carry forward the forecast and this is important. We stop computing much earlier than the moment when it is "*suddenly*" found that this search branch is deadlock and needs to be interrupted.

%}

%{

The direct transition from the beginning of the program to Block-4, and the sequential transition along the Block-2 -> Block-3 -> Block-4 chain differ in the form of representation of arrays *A* and *B*. This must be taken into account.

%}

if eventInd==4

%{

We find the initial indices of the remaining free rows in the solution matrix and store them in array *A*

%}

A=find(A==0);

% Denote by *nRow* the number of free lines

nRow=length(A);

%{

We find the initial indices of the remaining free columns in the solution matrix and store them in array *B*

%}

B=find(B==0);

else

T=find(A>0);

A=A(T);

nRow=length(T);

T=find(B>0);

B=B(T);

end

% Create an array *L(1:nRow,1:nRow)* and fill all the cells with one.

L=ones(nRow,nRow,'uint32');

%{

Create arrays rAr and *tAr*. We save in them cell indices of the diagonal control arrays that correspond to free positions in the array *L.*

%}

rAr=zeros(nRow,nRow,'uint32');

tAr=zeros(nRow,nRow,'uint32');

%{

Let's create arrays to take into account the cumulative list of restrictions formed by the left diagonal(*D1s*), the right diagonal(*D2s*) and the column projections(*Bs*)

%}

D1s=zeros(1,n2,'uint16');

D2s=zeros(1,n2,'uint16');

Bs=zeros(1,n,'uint16');

%{

Based on the information about the remaining free rows and free columns, we write zero in the corresponding free cells of the array *L*. We form the arrays *Cs*, *Ds*, *Bs*, as well as the arrays of accounting *rAr*, *tAr*. For all(*nRow*) rows and, accordingly, for the remaining free positions in these rows, we form cumulative list of restrictions for the left *D1s* and right *D2s* diagonal projections, as well as for the projections of the column *Bs*

%}

for p=1:nRow

i=A(p);

for q=1:nRow

j=B(q);

r=n+j-i;

t=j+i;

if D1(r)==0 && D2(t)==0

L(p,q)=0;

rAr(p,q)=r;

tAr(p,q)=t;

D1s(r)=D1s(r)+1;

D2s(t)=D2s(t)+1;

Bs(j)=Bs(j)+1;

end

end

end

% We calculate the sum of the elements of each row of the array *L*

rowSum=sum(L==0,2);

%{

We sort the sum values in increasing order of the number of free positions in the row .

%}

[sumSort,rowRangInd]=sort(rowSum);

%{

Here, in the *rowRangInd* array, row indices are sequentially stored with an increasing number of free positions in the row.

%}

if sumSort(1)>0

%{

Here *sumSort(1)* is the minimum number of free positions in the list of all rows of the array *L(nRow,nRow).*

%}

%{

If the minimum number of free positions are greater than zero, then we continue the solution and build the branch of the search.

%}

%{

Create an accounting control array *E* of size *nRow x nRow*, in each cell of which we will store the total value of the corresponding restrictions.

%}

E=zeros(nRow,nRow,'uint16');

%{

We calculate and store in array *E* the total value of the constraints of the control accounting arrays

%}

for p=1:nRow

for q=1:nRow

r=rAr(p,q); % Index r for array Cs

t=tAr(p,q); %Индекс t для массива Ds

j=B(q); %Индекс j для массива Bs

if r>0 && t>0

E(p,q)=D1s(r)+D2s(t)+Bs(j);

end

end

end

%{

Delete arrays that will not be used further.

For big values of n - clear D1s D2s Bs for free memory

%}

%clear D1s D2s Bs

% Next, instead of the arrays *D1s*, *D2s* and *Bs* we will use the array *E.*

%{

Before proceeding to the next event, we will save a copy of these arrays for reuse.

%}

Az=A;

Bz=B;

Qz=Q;

Lz=L;

Ez=E;

zPos=totPos;

% We have completed preparatory work in Block-4. Next, we go to Block-5

activeEvent=5;

else % if sumSort(1)>0

%{

If it turns out that among the remaining rows there is a roe in which there are no free position, then this means:

a)If *eventInd = 4*, then the composition in initially can’t be completed, since in the composition there is at least one free row without free position. (We can say that this composition is negative since «*birth*»).

b) If *eventInd <3*, then we must return to Block-2 and repeat the formation of the search branch.

%}

if eventInd<3

%{

If the event index is 1 or 2, then we return back to the beginning of Block-3. To do this, before going to Block-3, we will restore the initial state of the control arrays that we had at the end of Block-2.

%}

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

activeEvent=3;

% We increment the total counter of the number of repeated calculations

totSimCount= totSimCount+1;

elseif eventInd > 3

%{

If *eventInd = =4*, then this means that the size of the composition was such that we immediately went to this level, bypassing levels 1, 2 and 3. And since among all the remaining free rows there is at least one row in which there is no free position, then this composition initially can’t be completed. Therefore, we display the appropriate message and interrupt the program.

We set the variable *compositionInd* to -1 to fix that this composition is initially negative, and can’t be completed. Also, we assign zero to the *processInd* variable to interrupt further program operation

%}

solutionInd=2;

processInd=0;

break

end

end %if sumSort(1)>0

% After the preparatory work in Block-4, we go to Block-5.

case 5

%{

Block 5. The final stage of problem solving.

--------------------------------------------

We are at the last basic level of solution. There are a few free rows left until the end of the solution. If, starting from this level, in the process of solving the problem, the search branch leads to a deadlock, then we will return to this basic level. At this step, we must choose only one position in any free row, for location the queen. In this step, the number of possibilities of such a choice is equal to the sum of free positions in all remaining free rows. The two nested loops that are used in Block-5 serve only one purpose, to select the index of a free row at a given level, and select the free position in that row. The entire further search for the remaining free rows is performed only within the third nested loop. Therefore, first in Block-5:

- we select the row with the minimum number of free positions;

- we select a free position in this row, and place the queen.

After that, the following sequence of actions is performed in third loop:

a) *Among the remaining free rows, we select row with the minimum number of free positions,*

b) *Among the free positions in the selected row, we select that position that causes minimal damage to all remaining free positions.*

This cycle continues until a complete solution is obtained. If at some step the search branch leads to a deadlock, then the cycle is interrupted. Based on the backup copies, all arrays and variables corresponding to the current base level are restored. In this case, the third nested loop repeats again, without any changes in the parameters of the first and second nested loops. The number of such repeated calculations at the level of the third nested loop should not exceed the boundary value of *repeatBound*. If the number of repetitions exceeds the value of *repeatBound*, then in this case, after returning to the base level, the parameters of the first two nested loops changes as usual. The use of such a model of three nested loops is not entirely obvious at first glance. The fact is that in cases where there are several rows with the same minimum value of the total number of free positions, we randomly select the index of one of the two such rows (or a random index of one of three rows if three rows have the same minimum value). Similarly, a random selection of a free position in a row is performed if two positions in a row cause the same minimal damage to all remaining free positions. (Here, a random selection is made of only two positions that cause the same minimal damage). We use such an algorithm with only one purpose in order to maximize the use of the "*task resources*" that remain to this step. The closer to the end of the solution, the less likely it is that the selected free row will have a free position. According to the minimum risk rule, we must first place the queen in that free row, where the number of free positions is minimal. That is what we are doing. But in situations where two rows, or two free positions have the same minimum characteristics, we select such index randomly. When the third nested cycle is repeated several times without changing the parameters of the cycle, this gives us the opportunity to use more "*resource capabilities*" of the task at this level, because at some steps of forming the search branch random selection is used.

%}

% Zero the counter of the number of repeated calculations in Block-5

simCount5=0;

%{

The cycle *for iRow = 1: nRow* serves for sequential analysis of the remaining free rows, ranked in ascending order of the total number of free positions in the row. The indices of the corresponding rows are stored in the array *rowRangInd (1: nRow)*. Here *nRow* is the number of remaining free rows. The corresponding calculations *rowRangInd* array were carried out in the Block-4

%}

for iRow=1:nRow % First (external) nested loop

% Choose a row from the ranked list

selectRowInd=rowRangInd(iRow);

%{

*selectRowInd* is the row index in the array *L*. Let us determine the initial value of the row index on the chessboard, which in the array *L* corresponds to the index *selectRowInd.*

The value of *baseRowInd* will be needed later for repeated calculations.

%}

baseRowInd=A(selectRowInd);

% Copy the row with index *selectRowInd* from the array *L* into the temporary array *T*

T=L(selectRowInd,:);

%{

Define the free position indices in this row and save the result in the *baseFreePosAr* array (once again, note that the zero positions in the *L* array correspond to the free positions in the original solution matrix)

%}

baseFreePosAr=find(T==0);

% Define the total number of free positions(*nFreePos*) in this row

nFreePos=length(baseFreePosAr);

%{

The *for jCol = 1: nFreePos* loop is used for sequential analysis free positions in the row

%}

for jCol=1:nFreePos % Nested loop-2

% Assign i the real index of the selected row

i=baseRowInd;

%{

From the *baseFreePosAr* array, we select the index of the column, which is written in the cell with the number *jCol*. Here *jPos* is the column index of the array *L*

%}

jPos=baseFreePosAr(jCol);

jPosBase=jPos;

%{

We determine the real value of the column index (j), which corresponds to the chessboard in question

%}

j=B(jPos);

% Save the value of j in the *baseFreePos* variable for repeated calculations

baseFreePos=j;

%{

Assign to the variable *minRowInd* the value of the row index of the array *L*, which has the minimum number of free positions in the row.

%}

minRowInd=selectRowInd;

% Zero the count of the number of retries of the third nested loop.

repeatCount=0;

%{

The *while totPos <n* loop is the third nested loop, where, at each step, a free position is searched for the queen to be located in any of the remaining free rows.

%}

while totPos < n % Nested loop-3

%{

The initial index value of the selected row (i) and the column index value (j) for the first step, we determined above (in Block-4), before entering in the cycle.

In array *Q* in row (i) we save the (j) index of queen position.

%}

Q(i)=j;

totPos=totPos+1;

% Check if a complete solution is formed, then stop the calculations.

if totPos==n

solutionInd=1;

processInd=0;

break

end

%{

We used the result prepared in Block-4 and placed the queen in the cell (i,j) of the solution matrix. Thus, we completed the next cycle of determining the position on the chessboard for the location of the queen. After that, we must change the corresponding cells in all control arrays, given the indices (*minRowInd*, *jPos*) of array *L*

%}

A(minRowInd)=0;

B(jPos)=0;

%{

Change the corresponding cells of the array *L* using the equivalent indexes stored in the arrays *rAr* and *tAr*

%}

rx=n+j-i;

tx=j+i;

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

%{

We decrement the value of the accumulative control array *E*, since we placed the queen at the position (i,j)

%}

E(rxInd)=E(rxInd)-1;

E(txInd)=E(txInd)-1;

% Write 1 to all active cells in the *jPos* column of array *L*

A1=find(A>0);

L(A1,jPos)=1;

%{

At this step, inside the *while totPos <n* loop, we performed the following actions:

- we set the queen in cell (i,j), using the previously prepared information;

- performed the necessary procedural actions with control arrays, after the queen is placed in the cell (i,j).

%}

%{

Selection of a free row and free position in the row

----------------------------------------------------

Now, among the remaining free rows, we find the row with the minimum number of free positions, and from these positions we choose the one that, in the case of closing the position, will cause minimal damage to all remaining free positions in the remaining rows. To do this, follow these steps:

%}

%1.Define the amount of free positions in each remaining free rowz

rowSum=sum(L(A1,:)==0,2);

%{

2. We rank the *rowSum* array in increasing order.

3. Save the ranked values of the sums in the *freePosAr* array, and the indices of the corresponding rows in the *rowIndAr* array.

%}

[freePosAr,rowIndAr]=sort(rowSum);

%{

Since at this stage we simultaneously keep track the status of all remaining free rows, this gives us the opportunity to establish whether such a situation has occurred that in any of the remaining rows the number of free positions is zero. In this case, we consider the generated search branch as a deadlock and return to the beginning of the cycle.

This approach allows us to carry forward the forecast - we stop the calculations before it is established at the next step that there is’nt a free position in the row.

%}

%{

Here is the control point for the generated search branch. If, in each of the remaining free lines, there is at least one free position, then the formation of the search branch continues.

%}

if freePosAr(1)>0

%{

It may be that in a ranked list, the first two elements of the list, or the first three elements of the list, have the same minimum value. In this case, we randomly select the index of one of the two rows with the same minimum value (or, the index of one of the three rows, if there are three).

%}

if numel(freePosAr)==1||freePosAr(1)<freePosAr(2)

randPos=1;

elseif numel(freePosAr)>2 && freePosAr(1)==freePosAr(3)

randPos=randi(3);

else

randPos=randi(2);

end

minRow=rowIndAr(randPos);

minRowInd=A1(minRow);

%{

We determine the number of free positions in the selected row and store the indices of these rows in the *freePosAr* array

%}

freePosAr=find(L(minRowInd,:)==0);

% Define the number of free positions (*nFreePos*)

nFreePos=length(freePosAr);

%{

Among these positions, we choose the one that closes the minimum number of free positions in the remaining rows. To do this, we use the array *E*. If two rows have the same minimum number of free positions, then we randomly select the index *jPos* of one of them. *Here we introduce an element of “healthy” randomness into the algorithm in all cases when two rows have the same number of free positions.*

%}

if nFreePos==1

jPos=freePosAr(1);

else

T=E(minRowInd,freePosAr);

[tSort,tInd]=sort(T);

if tSort(1)<tSort(2)

jPos=freePosAr(tInd(1));

else

jInd=randi(2);

jPos=freePosAr(tInd(jInd));

end

end

%{

Based on the array of source indices *A*, we restore the real index i given row

%}

i=A(minRowInd);

%{

Based on the array of source indices *B*, we restore the real index j this column

%}

j=B(jPos);

else %if numel(rowSum)>0 && freePosAr(1)>0

%{

If it turns out that there are no free positions in the row, this means that the search branch has led to a deadlock. In this case, we close the search branch and increment the counters: *repeatCount, simCount5, totSimCount*

%}

repeatCount=repeatCount+1;

simCount5=simCount5+1;

totSimCount=totSimCount+1;

%{

We will restore the values of all control arrays (at the beginning of the execution of the *while totPos <n* cycle) and transfer control to the beginning of the cycle. If the number of repeated use of the cycle (*while totPos <n*) does not exceed the threshold value *repeatBound*, then control is transferred to the beginning of the cycle *while totPos <n* , without changing the parameters of two external cycles

%}

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

i=baseRowInd;

j=baseFreePos;

minRowInd=selectRowInd;

jPos=jPosBase;

if repeatCount>repeatBound

%{

If the number of loop reuse (*while totPos <n*) exceeds the *repeatBound* threshold, control is transferred to the outer loop *while jCol <= colPos*.

Above, we restored the corresponding parameters for the transition.

%}

repeatCount=0;

i=baseRowInd;

break

% Exiting the loop *while totPos <n*

% Going into the *while jCol <= colPos* loop

end

end %if freePosAr(1)>0

end %while totPos < n

if processInd==0

break

% Exiting the *while jCol <= colPos* loop

end

end %while jCol<=colPos

if processInd==0

break

% Exit the loop *for iRow = 1: nRow*

end

%{

Here, inside the *for iRow = 1: nRow* loop, is the only place where we control *totSimCount*. If the value of *totSimCount* exceeds the value of *totRepeatBound*, a message is displayed stating that the probability that this composition can be completed until the complete solution is less than *0.0001*

%}

if totSimCount > totSimBound

solutionInd=3;

break

%{

Let totSimCount <= totSimBound. Then,if, after exiting the *while jCol <= colPos* cycle, it turns out that the number of retests at this level (*simCount5*) exceeds the permissible limits (*simBound5*), then in case *eventInd <3*, we transfer control to event 2. If *eventInd == 4* , then the program continues, subject to restrictions.

%}

elseif simCount5 > simBound5 && eventInd<3

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

simCount3=0;

simCount5=0;

activeEvent=3;

break

end

end %for iRow=1:nRow

if solutionInd==3

if falseNegSimCount < falseNegSimBound

falseNegSimCount=falseNegSimCount+1;

switch eventInd

case 1

%Restore the arrays and transfer control to event 1

A=Ax;

B=Bx;

D1=D1x;

D2=D2x;

Q=Qx;

totPos=xTotPos;

activeEventInd=1;

case 2

%Restore the arrays and transfer control to event 3

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

activeEventInd=3;

case 4

%Restore the arrays and transfer control to event 5

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

activeEventInd=5;

end

%zeroing the counters for the corresponding events

simCount3=0;

simCount5=0;

totSimCount=0;

else

processInd=0;

break

end

end

if processInd==0

break

end

end %switch event

end %while processInd==1

toc

tStr = sprintf('Number of complete re-counting cycles = %d',falseNegSimCount);

disp(tStr)

if falseNegSimCount>0

totSimCount= falseNegSimCount\*totSimBound + totSimCount;

end

tStr = sprintf('Total number of usage the Back Tracking procedure = %d',totSimCount);

disp(tStr)

if solutionInd == 1

disp(' ');

disp('Solution is Ok!');

else

disp('This composition cannot be completied!');

end

if solutionInd==3

if n < 100

tStr= sprintf('The error of such conclusion is less than 0.0001');

disp(tStr);

elseif n < 800

tStr= sprintf('The error of such conclusion is less than 0.00001');

disp(tStr);

else

tStr= sprintf('The error of such conclusion is less than 0.000001');

disp(tStr);

end

end

tStr = sprintf('The first %d positions of solution:',nDisp); disp(tStr)

disp(Q(1:nDisp));

%{

We will save the result of completion in the file *nQueens\_Test\_Solution.mat.*If as a result of the solution it was not possible to complete the composition to the full solution, then zero values are saved in the corresponding cells of the array Q.

The file name *nQueens\_Test\_Solution.mat* is given as an example. Obviously, you can use any other name.

%}

outputFileName= 'nQueens\_Test\_Completion\_Solution.mat';

if solutionInd == 1

save(outputFileName,'Q');

iInfo=['Solution saved in file: ' outputFileName];

disp(iInfo);

end