%Generation\_kQueens\_Composition

%{

The program is designed to form a composition of *k* queens, which

distributed on a chessboard of size *n x n*. By composition we mean a random distribution of *k* queens on an arbitrary chessboard with size *n x n*, in such a way that three conditions of the problem are fulfilled: *in each row, in each column, as well as on the left and right diagonals passing through the position where the queen is located, there is no more than one queen*.

%}

%{

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), 2019*

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 find it possible to refer to my publication. It is a cultural element 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.

%}

%{

For the program to work, we must specify the size of the side of the checkerboard (*n*)

%}

n=1000;

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

disp(tStr);

nx=n-1; % *nx* – maximum composition size

% *n2* - size of control arrays

n2=2\*n;

%{

nFix - A fixed value for the size of the solution matrix. If a *n < nFix,*  then

execution of the decision is transferred to block-3, bypassing block-1 and block-2

%}

nFix=17;

%{

*bound\_1\_Ar, bound\_2\_Ar* - arrays of values for *eventBound1* and *eventBound2*

at *n <30*. These values are determined based on computational experiments.

%}

bound\_1\_Ar =[2 2 2 3 3 3 3 3 4 4 4 4 5 5 5 6 6 7];

bound\_2\_Ar =[4 4 5 5 5 6 6 7 7 8 8 9 10 10 11 11 12 13];

%{

Calculating values *eventBound1* и *eventBound2* based on equations

regression. These results are obtained on the basis of computational experiments

%}

if n<nFix

if n<9

eventBound1=1;

eventBound2=2;

elseif n<12

eventBound1=2;

eventBound2=3;

elseif n<14

eventBound1=3;

eventBound2=4;

else

eventBound1=3;

eventBound2=6;

end

sameRepeatBound=10;

else

sameRepeatBound=5;

if n<30

nInd=n-12;

eventBound1=bound\_1\_Ar(nInd);

eventBound2=bound\_2\_Ar(nInd);

else

u=log10(n);

w=u\*u;

if n<100

b1=293.898676\*w\*u-1495.491673\*w+2578.130423\*u-1470.692935;

b2=91.458481\*w\*u-474.647556\*w+849.173904\*u-497.393064;

elseif n<30000

b1=12.749568\*w\*u -46.535838\*w + 120.011829\*u -89.600272;

b2=9.717958\*w\*u -46.144187\*w + 101.296409\*u -50.669273;

else

b1=-0.886344\*w\*u+56.136743\*w-146.486415\*u+227.967782;

b2=14.959815\*w\*u-253.661725\*w+1584.711376\*u-3060.691342;

end

eventBound1=n-round(b1);

eventBound2=n-round(b2);

end

end

%{

Boundary values *simBound1, simBound2* and *simBound4* define maximum number of recalculations within blocks- 1,2 and 4.

%}

simBound1=3;

simBound2=5;

simBound4=10;

%{

*totSimBound* - boundary value for accounting for the total number of all

recalculations.

%}

totSimBound=1000;

% Let's define a random composition size *nComp --> (1, ... ,n-1)*

nComp=randi(nx);

%{

The algorithm will continue until the *nComp* queens are distributed on the chessboard.

%}

% Let's zero the array *Q(1:n)*

Q=zeros(1,n,'uint32');

% For work, we will zero the control arrays:

% *A(1:n)* - to control the indices of free rows.

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

% *B(1:n)* - to monitor indexes of free columns.

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

% *C(1:n2), D(1:n2)* - to control the occupancy of diagonal projections.

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

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

% *totPos* - counter of the total number of all recalculations.

totPos=0;

%{

Reset the repetition counters for each event: *simCount1, simCount2, simCount4* - repetition counters within blocks 1,2 and 4.

*totSimCount* - counter of the total number of all repetitions.

%}

simCount1=0;

simCount2=0;

simCount3=0;

simCount4=0;

totSimCount=0;

%{

In this composition generation program, calculations start from the first block. (*eventInd = 1).*

%}

eventInd=1;

% *processInd* is switch Index to exit the loop

processInd =1;

%{

All events unfold inside the loop *while swiInd==1*, until a solution is received.

%}

tic

while processInd ==1

% The *eventInd* variable serves as a toggle switch between 4 events

switch eventInd

case 1

% The *rand\_set & rand\_set* algorithm. Composition is forming up to the size *simBound1*

simCount1=simCount1+1;

if nComp<eventBound1

xEvent=nComp;

else

xEvent=eventBound1;

end

while totPos < xEvent

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 C(r)==0 && D(t)==0

C(r)=1;

D(t)=1;

Q(i)=j;

A(i)=1;

B(j)=1;

totPos=totPos+1;

end

if totPos==xEvent

break

end

end

end

if nComp<= eventBound1

processInd=0;

else

% Find the original indices of the remaining free rows in the solution matrix.

A=find(A==0);

nFreeRow=length(A);

%{

Find the original indices of the remaining free columns in the solution matrix.

%}

B=find(B==0);

%{

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

Further, if the cell *L (p, q)* turns out to be free, then instead of

units we write down to zero.

%}

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

%{

Let's create arrays rAr and *tAr* to store the matching indices control arrays.

%}

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

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

%{

Based on information about the remaining free rows and free columns, write zero in the corresponding free cells of the array *L*

We form arrays of accounting *rAr, tAr*.

%}

for p=1:nFreeRow

i=A(p);

for q=1:nFreeRow

j=B(q);

r=n+j-i;

t=j+i;

if C(r)==0 && D(t)==0

L(p,q)=0;

rAr(p,q)=r;

tAr(p,q)=t;

end

end

end

%{

Let's create backup copies of all main arrays. We will need them

if it becomes necessary to return to the beginning of event-2 for repeated

calculations (Back Tracking).

%}

Ay=A;

By=B;

Cy=C;

Dy=D;

Qy=Q;

Ly=L;

rAr\_y=rAr;

tAr\_y=tAr;

yPos=totPos;

eventInd=2;

end

case 2

% In this block, selection is made based on the *rand & rand* algorithm

simCount2=simCount2+1;

%{

Next, we will continue to form the branch of the search for a solution based on the data, collected in array *L*

%}

if nComp<eventBound2

xEvent=nComp;

else

xEvent=eventBound2;

end

while totPos < xEvent

% Determine the number of free rows on the base of array *A*

freeRowInd=find(A>0);

freeRow=length(freeRowInd);

% Selecting a random row index based on a list of free row indices.

selectRowInd=randi(freeRow);

iInd=freeRowInd(selectRowInd);

% Let's form a list of indices of free positions in row *i* of array *L*

rowFreePosAr=find(L(iInd,:)==0);

nFreePos=length(rowFreePosAr);

if nFreePos>0

%{

If there are free positions in the selected row, then we continue the solution

Here, the position of the queen in the row is randomly selected.

%}

selectPosInd=randi(nFreePos);

jInd=rowFreePosAr(selectPosInd);

j=B(jInd);

% Store the *j-index* of the queen's position in the solution array.

i=A(iInd);

Q(i)=j;

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

totPos=totPos+1;

% Write *0* to the *iInd* cell of array A to fix that row *i* in init array is busy.

A(iInd)=0;

% Write *0* to cell *jInd* of array *B* to fix that column *j* in init array is busy.

B(jInd)=0;

%{

Change the corresponding cells of the forbidden arrays C and D using

real values of indices *(i, j)*.

%}

rx=n+j-i;

tx=j+i;

C(rx)=1;

D(tx)=1;

%{

Change the corresponding cells of the array *L* using equivalent indices,

stored in arrays *rAr* and *tAr*.

%}

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

L(freeRowInd,jInd)=1;

else % if freePos>0

%{

If there are no free positions in the row under consideration, then we have reached a dead end, therefore must close the given branch and go back to the beginning *while totPos <simBound2* loop and repeat the formation of a new search branch. But before that, we must restore all the required arrays based on

backups (Back Tracking).

%}

if simCount2 < simBound2

A=Ay;

B=By;

C=Cy;

D=Dy;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yPos;

eventInd=2;

else

% Let's zero out the control arrays and transfer process to event-1

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

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

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

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

Q=zeros(1,n,'uint32');

totPos=0;

simCount2=0;

eventInd=1;

break

end

end % *if freePos>0*

end % *while totPos < simBound2*

if nComp<= eventBound2

processInd=0;

elseif totPos >= xEvent

eventInd=3;

end

%{

We have completed the second part of the formation of the search branch and reached the level, when *simBound2* queens are correctly distributed in the decision matrix. Let's go to the third stage.

%}

case 3

simCount3=simCount3+1;

%{

Next, we will exclude occupied rows and occupied columns from consideration. Let's form a new compact matrix *L* as the intersection of the number of remaining rows and the number the remaining columns. To do this, find the indices of the remaining free rows, according to the array of accounting for occupied rows *A*.

%}

T=find(A>0);

A=A(T);

nRow=length(T);

% Let's define the array of indices of free columns in the same way.

T=find(B>0);

B=B(T);

%{

Create an array *L (1: m, 1: m)* and fill all cells with one. Further, if

cell *L (p, q)* turns out to be free, then instead of one we write in this cell

zero.

%}

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

%{

Let's create arrays to store the indexes of compliance with the control arrays.

%}

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

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

% Let's create arrays to account for the cumulative list of restrictions.

Cs=zeros(1,n2,'uint32');

Ds=zeros(1,n2,'uint32');

Bs=zeros(1,n,'uint32');

%{

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

%}

for p=1:nRow

i=A(p);

for q=1:nRow

j=B(q);

r=n+j-i;

t=j+i;

if C(r)==0 && D(t)==0

L(p,q)=0;

rAr(p,q)=r;

tAr(p,q)=t;

Cs(r)=Cs(r)+1;

Ds(t)=Ds(t)+1;

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

end

end

end

% Let's calculate the sum of the elements of each row of the array L.

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

%{

Sort the sum values in ascending order of the number of free positions

in each row.

%}

[sumSort,rowRangInd]=sort(rowSum);

%{

Here, in the *rowRangInd* array, the row indices with an increasing number of free positions in the row. If it turns out that in all the remaining rows "collected" in the array *L* there are free positions, then the *rowRangInd* array will be used further, in block 4.

%}

if sumSort(1)>0

%{

Here *sumSort (1)* is the minimum number of free positions in the list of all rows of the array *L (m, m)*. If the minimum number of free *positions> 0*, then we continue building the search branch, since until this step, the constructed branch remained promising.

Let's create a control array of accounting *E* of size *nRow x nRow*, in each cell of which we store the cumulative value of the accumulative arrays of restrictions.

%}

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

%{

We calculate and store in *E* the cumulative value of the accumulative arrays of constraints.

%}

for p=1:nRow

for q=1:nRow

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

t=tAr(p,q); % Index *t* for array *Ds*

j=B(q); % Index *j* for array *Bs*

if r>0 && t>0

E(p,q)=Cs(r)+Ds(t)+Bs(j);

end

end

end

% Further, instead of arrays *Cs, Ds, Bs*, we will use the array *E*

%{

Before moving on to the next event, let's save copies of these arrays for reuse.

%}

Az=A;

Bz=B;

Qz=Q;

Lz=L;

Ez=E;

zPos=totPos;

% Next, let's move on to *event- 4*

eventInd=4;

else % *if sumSort(1)>0*

%{

If it turns out that among the remaining rows there is a row in which there are no free positions, then we restore the initial values of the arrays and transfer control to *event- 2*

%}

A=Ay;

B=By;

C=Cy;

D=Dy;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yPos;

eventInd=2;

end % *if sumSort(1)>0*

case 4

simCount4=0;

for iRow=1:nRow

selectRowInd=rowRangInd(iRow);

%{

Determine the corresponding (original) value of the row index in the array *L* using the index of inite data.

%}

baseRowInd=A(selectRowInd);

% Determine the number of free positions in the selected row.

T=L(selectRowInd,:);

baseFreePosInd=find(T==0);

nFreePos=length(baseFreePosInd);

%{

Further, here, at the basic level, within the considered row with the current minimum value of the number of free positions in the row, we will sequentially, in a loop, consider each free position.

%}

for jCol=1:nFreePos

% Let's assign *i* the real number of the selected row (according to the index of the source data)

%}

i=baseRowInd;

jPos=baseFreePosInd(jCol);

jPosBase=jPos;

% Assign *j* to the value of the selected free position (according to the index of the source data).

%}

j=B(jPos);

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

baseFreePos=j;

%{

Let's assign *minRowInd* the row index in the array *L (1: nRow, 1: nRow)* with the minimum number of free positions in the row.

%}

minRowInd=selectRowInd;

% Event-4. The beginning of the main part of the algorithm.

sSame=0;

while totPos < nComp

%{

For the first step in this cycle, the values *i, j* are defined above. Store the *j*-index of the queen's position in the solution array.

%}

Q(i)=j;

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

totPos=totPos+1;

%{

Let's check if a complete solution is formed, then we complete the calculations.

%}

if totPos==nComp

totSimCount=totSimCount+1;

processInd=0;

break

end

%{

We have completed another cycle of determining indices for the location of the queen and placed the queen in the cell *(i, j)* of the decision matrix. After that, we must change the corresponding cells in all control arrays, taking into account the indices *(minRowInd, colInd)* of the array *L*.

%}

A(minRowInd)=0;

B(jPos)=0;

%{

Change the corresponding cells of the array *L* using the equivalent indices 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 cumulative control array, i.e. reduce the "*effect of influence*" of free positions in the selected row, after the queen has been placed there.

%}

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

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

%{

Let's write 1 to all active cells of the *colInd* column. Active cells are specified by array A1.

%}

A1=find(A>0);

L(A1,jPos)=1;

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

% Determine the row index *minRowInd* with the minimum number of free positions

[freePosAr,rowIndAr]=sort(rowSum);

if freePosAr(1)>0

%{

If two rows have the same minimum number of free positions, then randomly choose the index of one of these rows.

%}

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

randPos=1;

else

randPos=randi(2);

end

minRow=rowIndAr(randPos);

minRowInd=A1(minRow);

i=A(minRowInd);

% Determine the number of free positions in this row.

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

nFreePos=length(rowFreePosAr);

%{

Choose among these positions the one that closes the minimum number of free positions in the remaining rows. To do this, we use the array *E (m, m).*

%}

if nFreePos ==1

jPos= rowFreePosAr (1);

else

T=E(minRowInd, rowFreePosAr);

[tSort,tInd]=sort(T);

if tSort(1)<tSort(2)

jPos= rowFreePosAr(tInd(1));

else

jInd=randi(2);

jPos= rowFreePosAr(tInd(jInd));

end

end

j=B(jPos);

%{

Thus, choosing *jPos* from the list *freePosInd (1: freePos)* in the current row will close the minimum number of free positions in the remaining rows.

%}

else % if minFreePos>0

%{

If there are no free positions in the row, (minFreePos = 0), then close the search branch and increment the value of the recalculation counter.

%}

sSame=sSame+1;

simCount4=simCount4+1;

totSimCount=totSimCount+1;

%{

Let's restore the values of the original arrays and go back to the beginning of the loop: *while jCol <= colPos*

%}

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

%{

If the number of internal repetitions exceeds the allowable boundary of *sameRepeatBound*, then we interrupt and go to the beginning of the loop: *while jCol <= colPos*.

%}

if sSame>sameRepeatBound

sSame=0;

break % Go to the beginning of the cycle:

% *for jCol=1:baseFreePos*

end

end % *if freePosAr(1)>0*

end % *while totPos < n*

if processInd==0 % Exiting the loop: w*hile jCol <= colPos*

break

end

%{

If the number of repeated calculations *simCount4* inside the *while jCol <= colPos* loop exceeds the *repeatBound4* threshold, then this loop is interrupted.

%}

if simCount4 > simBound4

break

end

end % while jCol<=colPos

if processInd ==0

break

end

if totSimCount > totSimBound

processInd =0;

break

end

%{

If, after sequentially performing the appropriate procedures in *blocks- 1,2,3,4* we do not get a solution, then we repeat the search for a solution starting from block-2.

%}

if simCount4 > simBound4

A=Ay;

B=By;

C=Cy;

D=Dy;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yPos;

eventInd=2;

break

end

if totSimCount > totSimBound

falseNegative=falseNegative+1;

end

end % *for iRow=1:nRow*

otherwise

processInd=0;

end % *switch eventInd*

if processInd==0

break

end

end % *while processInd==1*

toc

tStr = sprintf(' The size of Composition = %d', nComp);

disp(tStr);

nDisp=50;

if n<=nDisp

nDisp=n;

disp('Positions of all Queens on the chesboard:');

else

disp('Positions of the first 50 Queens on the chesboard:');

end

disp(Q(1:nDisp))

% Let's save the generated composition.

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

save(outputFileName,'Q');

iInfo=['Composition is saved in file: ' outputFileName];

disp(iInfo);