% Solution\_n\_Queens\_Completion\_Problem

%{

Le programme est conçu pour compléter à une composition arbitraire de *k* reines. Par composition, nous entendons la distribution aléatoire de *k* reines à

un échiquier arbitraire de taille *n x n*, de sorte que trois conditions de cette tâche soient remplies: *dans chaque ligne, dans chaque colonne, ainsi que sur les diagonales gauche et droite passant par la position où se trouve la reine, il n'y a pas plus d'une reine*.

Il est nécessaire de trouver au moins une solution et de montrer ainsi que la solution existe, ou de juger qu'avec une probabilité donnée (*P*), cette composition ne peut pas être complétée.

%}

%{

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”.

%}

%{

Auteur et développeur du projet - Grigoryan Eros (EricGrig), 2020

Je serai heureux si des sections du code, ou l'ensemble du programme dans son ensemble, sont utilisées à des fins scientifiques ou à des fins éducatives. En même temps, je vous serais reconnaissant de bien vouloir vous référer à ma publication. C'est un élément de la culture et un signe de respect mutuel.

Pour une utilisation commerciale de toute partie du code du programme, ou du programme dans son ensemble, le consentement écrit de l'auteur est requis.

%}

%{

Les résultats de recherche associés au développement de cet algorithme sont publiés dans *arxiv.org* dans l'article: *Grigoryan E., Linear algorithm for solution n-Queens Completion problem,* [*https://arxiv.org/abs/1912.05935*](https://arxiv.org/abs/1912.05935) *.* Ce sera correct si vous lisez d'abord cette publication avant de commencer à étudier le code source du programme. Cela rendra la description du programme plus transparente et réduira le nombre de questions possibles.

La version russe de l'article est publiée sur le site Internet de la communauté des programmeurs habr.com: [*https://habr.com/ru/post/483036/*](https://habr.com/ru/post/483036/)

%}

%{

1.La début

----------

Comment les données sont préparées?

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

Indiquez *n* la taille du côté de l'échiquier. Soit un tableau unidimensionnel annulé de taille *n*. Si la reine en position j est située dans la i-ème rangée de l'échiquier, alors, en conséquence, la valeur j est écrite dans la i-ème cellule du tableau de données unidimensionnel.

Ensuite, avec le nom "*échiquier de taille n x n*", nous utiliserons le nom "*matrice de la solution de taille n x n*"

Lisons le fichier de données de la composition et enregistrons le résultat dans le tableau *Q*. Ici, à titre d'exemple, le nom du fichier de données est utilisé *kQueens\_Test\_Composition.mat* . Ce nom doit être remplacé par le nom qui correspond à votre fichier de données.

%}

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

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

disp(iInfo);

% Input data file

Q=importdata(inputFileName);

n=length(Q);

%{

Nous définissons le nombre de cellules nulles dans le tableau *Q*, que nous désignons par *nZero*. Ainsi, nous déterminons le nombre de lignes libres dans la matrice de solution

%}

nZero=sum(Q==0);

% Indique la taille de la composition par *nComp*

nComp=n-nZero;

%{

Nous affichons pour l'utilisateur les informations nécessaires sur ce problème: la taille de la matrice de décision, la taille de la composition, le nombre de positions libres

%}

disp(' ');

tStr = sprintf(' Chessboard Size = %d',n); disp(tStr)

disp(' ');

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

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

%{

Affichage des 50 premières lignes de la composition (ou de la composition entière si n <50

%}

if n<50

nDisp=n;

else

nDisp=50;

end

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

disp(Q(1:nDisp));

%{

S'il s'avère que la taille de la composition est égale à la taille de l'échiquier, nous afficherons le message correspondant et terminerons le programme.

%}

if nComp==n

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

disp(tStr);

pause

exit

end

%{

S'il s'avère que la taille de la composition est nulle, c'est-à-dire qu'il n'y a pas de composition, nous afficherons le message correspondant et terminerons le programme

%}

if nComp==0

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

disp(tStr);

exit

end

%{

Général

-------

Au cours de la recherche, trois algorithmes de base pour résoudre le problème ont été développés, qui diffèrent à la fois par la vitesse de résolution du problème et par son efficacité. Le programme implémente également des sections de code qui exécutent des fonctions préparatoires pour les algorithmes de base.

Désignez ces sections du code, respectivement: Block-0, Block-1, ..., Block-5.

*Bloc-0* -- Bloc de départ. Vérification de la bonne composition, préparation des

! tableaux de contrôle

!

*Bloc-1* -- formation d'une solution basée sur l'algorithme *rand\_set & rand\_set. L*a valeur

!limite du nombre de reines correctement définies est *eventBound2*

!

*Bloc-2* -- opérations préparatoires pour le passage au Bloc-3

!

*Bloc-3* -- formation d'une solution basée sur un algorithme *rand & rand*

! La valeur limite du nombre de reines correctement établies est *eventBound3*

!

*Bloc-4* -- opérations préparatoires pour le passage au Bloc-5.

!

*Bloc-5* -- formation d'une décision basée sur les règles du "*risque minimal*" et

"*dummage minimal*".

Dans le processus de recherche d'une solution, le calcul est transféré d'un bloc de l'algorithme à un autre sous forme de stick relais.

L'algorithme a été développé pour une plage assez large de valeurs de la taille d'un échiquier: d'une valeur de *7* à *100 000 000*. Si la taille de la ordinateur RAM le permet, vous pouvez également effectuer des calculs pour de grandes valeurs de la taille d'un échiquier, par exemple, pour *n = 800 000 000*. (Sur un ordinateur avec *RAM = 32 Go*, le problème de sélection a été résolu pour *n = 1 000 000 000*. Cependant, en raison du manque de mémoire, nous avons dû modifier un peu le programme et effectuer des calculs en deux étapes).

%}

if n<5

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

disp(tStr);

exit

end

%{

Si *nComp <= eventBound2*, puis les calculs commencent par le Bloc-1

Si *eventBound2 < nComp <= eventBound3*, puis les calculs commencent par le Bloc-2

Si *nComp > eventBound3*, le contrôle est ensuite transféré au Bloc-4, où les travaux préparatoires sont effectués et après quoi une transition est effectuée dans le Bloc-5 pour les calculs de base

%}

%{

Comme les résultats de l'étude l'ont montré, dans la plage de valeurs *n=(7,...,99)*

nécessite une approche plus méticuleuse de la formation d'une branche de la recherche d'une solution.

Par conséquent, cet intervalle a été divisé en deux sous-intervalles *(7,...,49)* et

*(50,...,99)*, dans chacun d'eux, l'algorithme correspondant est utilisé

solutions. (Ici, les valeurs limites peuvent être légèrement augmentées ou diminuées. De là, l'essence de la solution ne changera pas)

%}

% *nFix1, nFix2* - La valeur fixe de la taille de la matrice de décision.

nFix1=50;

nFix2=100;

%{

Si *n < nFix1*, puis l'exécution des calculs est transférée au Block-4.

Si *nFix1 <= n < nFix2*, puis l'exécution des calculs est transférée au Block-2.

%}

%{

À propos des valeurs limites *eventBound2* et *eventBound3*:

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

Si le processus de formation d'une branche pour trouver une solution conduit à une impasse, nous devez revenir en arrière (*Back Tracking*), à certains des niveaux précédents, et recommencer la recherche d'une solution. Pour ce faire, nous devons savoir lequel des niveaux précédents doit être renvoyé afin de conserver à l'avance les valeurs des principaux paramètres de ce niveau. Le choix du point de retour optimal est une tâche plutôt compliquée et intéressante. Dans cet algorithme, nous utilisons la règle suivante. En plus du niveau initial, qui correspond à l'achèvement du contrôle de composition et à la formation de tous les tableaux de contrôle, nous formons et utilisons deux niveaux de base supplémentaires pour revenir en arrière: *eventBound2* et *eventBound3*. *Ici, la mesure comptable du niveau de décision est le nombre de reines correctement placées sur l'échiquier*.

%}

%{

Sur le nombre de recalculs au dernier niveau de base

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

Les plus grandes difficultés dans le fonctionnement de l'algorithme surviennent à la dernière étape de la résolution du problème. Toutes les erreurs cachées qui ont été commises lors du choix de l'index d'une ligne libre, et (ou) du choix d'une position libre dans cette ligne, "s'accumulent progressivement", et à la dernière étape se manifestent par le fait que parmi les lignes libres restantes, il y a au moins une ligne, dans qui n’existe pas une seule position libre. Cela signifie une impasse. Par conséquent, l'algorithme pour former la branche de la recherche d'une solution au dernier stade est plus méticuleux. (Par analogie, il convient d'utiliser ici une comparaison de la microchirurgie avec la chirurgie).

Afin de prendre en compte les moyens efficaces possibles de former une branche de la recherche d'une solution, à la dernière étape, à l'intérieur de deux cycles imbriqués, nous exécutons le troisième cycle, qui est répété plusieurs fois, en revenant au début du même cycle, sans changer les paramètres des deux cycles externes. Cela revient à appliquer la procédure de *Back Tracking* à l'intérieur d'un système de boucles imbriquées. La valeur seuil du nombre de fois qui peuvent être produites à l'intérieur de ce cycle est indiquée par *repeatBound*. De plus, dans le texte, cela sera discuté un peu plus en détail.

%}

%{

Calculez les valeurs *eventBound2* et *eventBound3*, ainsi que le nombre seuil de répétitions *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

%{

Valeurs empiriques des paramètres *eventBound2* и *eventBound3* sont établies sur la base d'un très grand nombre d'expériences de calcul et sont optimisées pour toute la gamme de résolution de problèmes. Pour toute plage plus petite de *n* valeurs, ces paramètres peuvent être légèrement modifiés et obtenir des valeurs auxquelles le programme fonctionnera un peu plus rapidement.

Dans le processus de résolution du problème, si un blocage se produit, certains blocs de l'algorithme sont réexécutés. De plus, selon les valeurs *n* et *nComp*

des calculs répétés commencent dès le début ou à partir d'un certain niveau atteint. Si une recherche répétée d'une solution aux niveaux supérieurs ne mène pas au succès, alors une recherche répétée d'une solution commence aux niveaux inférieurs. Ici, les variables *simBound3*, *simBound5* déterminer le nombre maximum de recalculs dans les Bloc-3 et les Bloc-5. *totSimBound* - détermine le montant total

tous les recalculs à tous les niveaux.

%}

simBound3=5;

simBound5=100;

totSimBound=1000;

%{

falseNegSimCount - nombre de cycles de recomptage complets du compte tenu de la composition.

Ceci est un compteur du nombre de recalculs pour terminer compositions,

compositions, qui n'ont pas réussi à terminer la première fois

%}

falseNegSimCount=0;

%{

falseNegSimBound - limite de recalcul.

Si, dans un premier temps, il n'est pas possible de terminer la composition, alors

alors le calcul est répété à partir du point de départ

%}

falseNegSimBound=10;

%{

L'algorithme utilise plusieurs tableaux de contrôle:

*A* - pour contrôler les indices de ligne,

*B* - pour contrôler les indices de colonne.

%}

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

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

%{

De plus, deux tableaux sont utilisés pour contrôler les cellules des projections diagonales: *D1*(1:n2), *D2*(1:n2), où *n2* est la taille des tableaux de contrôle

%}

n2=2\*n;

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

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

%{

Sélection d'index d'événements actifs *eventInd.*

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

Définissez l'index de bloc à partir duquel le programme démarrera. Pour ce faire, affectez la valeur appropriée à la variable *eventInd.* Nous déterminons également la valeur seuil pour le nombre de calculs répétés à la dernière étape (*simBound5*).

%}

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. Vérification de l'entrée de composition

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

Пla composition sera vérifiée et les cellules correspondantes des matrices de contrôle seront remplies séquentiellement *A, B, C* et *D*.

Dans les cellules du tableau *Q*(i), les index des positions des reines correctement définies pour les lignes correspondantes sont stockés. Incrémentation de la variable *totPos* pour tenir compte du nombre de reines correctement installées.

%}

%{

Définissez les indices de ligne occupés dans le tableau *Q* et enregistrez les résultats dans le tableau *qPosInd*

%}

qPosInd=find(Q>0);

%{

Nous écrivons l'unité dans les cellules du tableau *B* qui correspondent aux colonnes occupées.

%}

B(Q(qPosInd))=1;

% Définissez la somme des unités du tableau *B*

s=sum(B);

%{

Vérifiez si deux reines différentes se trouvent dans la même colonne. Si c'est le cas, il y a une erreur dans la composition d'origine. Dans ce cas, nous afficherons le message correspondant et interromprons le travail ultérieur du programme.

%}

if s~=n-nZero

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

exit

end

%{

L'algorithme de vérification fonctionne comme suit. Si cellule (i,j)), où j=*Q*(i) libre, en tenant compte des restrictions diagonales et des restrictions sur le nombre d'éléments dans chaque colonne, la reine se situe correctement dans cette cellule. Nous ne vérifions pas la règle «*pas plus d'une reine par ligne*», car le modèle de préparation des données initiales exclut la possibilité de plus d'une reine dans la composition. Chaque cellule d'un tableau de données d'entrée à une dimension caractérise la ligne correspondante dans la matrice de décision.

%}

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

%{

Si une erreur est détectée dans la composition, c'est-à-dire l'emplacement des reines ne répondra pas aux conditions de la tâche, un message correspondant s'affichera et le programme sera interrompu.

%}

if qError==0

A(qPosInd)=1;

totPos=nComp;

else

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

disp(tStr)

exit

end

% Supprimons le tableau *qPosInd*, car nous ne l'utiliserons plus.

clear qPosInd

%{

Enregistrement de copies des tableaux générés pour réutilisation

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

Nous avons fait un travail préparatoire. Organisation des données d'entrée et vérification de la validité de la composition. Des copies des tableaux formés sont créées. S'il est nécessaire de revenir à ce niveau et de retraiter la composition, nous restaurerons tous les tableaux nécessaires en fonction de ces sauvegardes. Ce niveau est le niveau de base initial (zéro), d'où commence la formation de la branche de recherche de solution. Ici, le nombre de reines correctement installées est égal à la taille de la composition originale.

%}

if eventInd==1

Ax=A;

Bx=B;

D1x=D1;

D2x=D2;

Qx=Q;

xTotPos=totPos;

end

%{

Remettre à zéro les compteurs de répétition pour le troisième (*simCount3*) et cinquième (*simCount5*) niveaux.

%}

simCount3=0;

simCount5=0;

% *simCount3* sera ensuite utilisé comme commutateur dans le Bloc-3*.*

%{

Mettons à zéro *totSimCount* - le compteur du nombre cumulé de toutes les répétitions à différents niveaux.

%}

totSimCount=0;

%{

Tous les événements se déroulent à l'intérieur du cycle *while processInd == 1* jusqu'à ce qu'une solution pour cette composition soit obtenue, ou qu'il soit établi que la solution n'existe pas avec la probabilité *P*. Le critère principal pour une telle évaluation est le nombre total de tous les calculs répétés (*totSimCount*). Dans l'article, dont le lien est donné dans le commentaire au début de ce programme, est écrit suffisamment en détail à ce sujet.

À la suite d'un grand nombre d'expériences de calcul, pour une grande variété de compositions aléatoires de taille arbitraire *k* et pour différentes valeurs de la taille d'un échiquier *n*, il a été constaté que si le nombre total de calculs répétés *totSimCount* dépasse la valeur seuil de *totSimBound*, et aucune solution n'a été trouvée, la composition ne peut pas être terminée. La probabilité d'erreur d'un tel jugement est de *0,0001 .*

%}

%{

Le début de la formation la branche de recherche solutions

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

Comme mentionné ci-dessus, nous considérons les différents blocs du programme comme des événements distincts.

Il y a cinq événements de ce type. Trois d'entre eux correspondent aux blocs principaux du programme et deux événements correspondent à des blocs de programme qui exécutent des fonctions préparatoires. Affectez la variable *activeEvent* à l'index de l'événement actuellement actif.

%}

activeEvent=eventInd;

%{

Nous introduisons la variable *processInd* - comme "switcher" pour quitter la boucle.

Le cycle est exécuté si *processInd == 1* sinon l'exécution de la boucle est interrompue.

%}

processInd=1;

%{

Nous introduisons la variable *compositionInd*. Si *compositionInd == 1*, cela signifie que la composition est positive, c'est-à-dire qu'elle peut être complétée en solutions complètes.

Si *compositionInd == 0*, alors la composition sera considérée comme négative, c'est-à-dire qu'elle ne peut pas être complétée en solution complète. Si *compositionInd = -1*, alors la composition sera considérée comme négative "*par naissance*". Cela signifie que dans le tableau d'entrée, parmi les rangées libres de cette composition, il y a au moins une rangée, dans laquelle il n'y a pas de position libre (toutes les positions sont fermées en raison des interdictions formées par des reines préalablement établies).

%}

solutionInd =1;

% Début du cycle principal

while processInd==1

% L'événement *event* sert de commutateur entre 5 événements

switch activeEvent

case 1

%{

Bloc-1. Utilisation de l'algorithme *rand\_set & rand\_set*

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

Dans ce bloc, nous recherchons une ligne libre et une position libre dans cette ligne pour positionner la reine, jusqu'à ce que le total des reines correctement définies soit égal à la valeur de seuil (*eventBound2*). L'algorithme qui s'exécute dans ce bloc est appelé *rand\_set & rand\_set*. Son essence est la suivante. On retrouve les indices de toutes les lignes libres. Effectuer une permutation aléatoire de ces indices. De même, on retrouve les indices de toutes les colonnes libres. Nous passons également une permutation aléatoire de ces indices. Nous considérerons les indices de paires de ces deux listes (indice de ligne aléatoire, indice de colonne aléatoire). Si la cellule de la matrice de décision correspondant à cette paire d'indices ne contredit pas les restrictions diagonales, alors nous mettons la reine dans cette position. Dans ce cas, nous écrivons 1 dans les cellules des tableaux de contrôle appropriés *A, B, D1* et *D2*. Le total des reines correctement installées (*totPos*) augmente d'une unité.

%}

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

%{

Dans ce bloc, les positions des reines sont déterminées rapidement. Et, bien qu'ici toutes les positions soient déterminées correctement, cependant, le «tableau» global de la distribution des reines dans la matrice de la solution est «grossier». Si nous ne nous arrêtons pas à une étape optimale, la poursuite de la construction de la branche de la recherche risque de conduire à une impasse. Étant donné la vitesse élevée de l'algorithme *rand\_set & rand\_set*, sur la base de ce bloc, nous passons par le chemin maximum de la valeur de *nComp* aux valeurs *eventBound2*. Après cela, l'exécution du programme est transférée au bloc suivant.

%}

%{

Important! Ce bloc n'est exécuté que si *n> = 100* et la taille de la composition est inférieure à *eventBound2*. Comme l'ont montré les résultats de près de deux dizaines de millions d'expériences de calcul, pour une valeur donnée de *eventBound2*, cet algorithme complète toujours la composition à la valeur de *eventBound2*. Il n'y a jamais eu de situation où l'algorithme est bouclé et non terminé. Cela est dû au fait que la valeur de *eventBound2* n'est pas critique, et il existe de nombreuses possibilités différentes pour atteindre ce niveau. Pour cette raison, à ce stade, nous avons exclu le contrôle de l'achèvement du cycle de l'algorithme, bien que cette possibilité ait été prise en compte dans les premières versions du programme. *La soif de vitesse était plus élevée que la logique d'embrasser des situations admissibles presque impossibles*.

%}

%{

Lorsque le nombre de reines correctement placées (*totPos*) est égal à *eventBound2*, la gestion des événements est transférée vers Bloc-2.

%}

activeEvent=2;

case 2

%{

Bloc-2. Préparation des tableaux nécessaires pour le travail dans le Bloc-3

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

Dans ce bloc, un travail préparatoire est effectué pour la transition vers le Bloc-3. Son essence est la suivante: que le nombre de lignes libres restantes soit *nFreeRow*. Nous formons un tableau *L*(1:nFreeRow, 1:nFreeRow) et collectons les indices de position libre de toutes les lignes restantes. Cela signifie ce qui suit: dans la matrice de solution d'origine, nous considérons la grille d'intersection de colonnes libres et de lignes libres. Nous transférons toutes ces cellules de la grille d'intersection à la projection dans un tableau plus petit *L*. Dans ce cas, nous prenons en compte la correspondance des indices du tableau *L* avec les indices correspondants de la matrice de solution d'origine.

%}

%{

Recherchez les indices initiaux des lignes libres restantes dans la matrice de solution et enregistrez les résultats dans le tableau *A*.

%}

A=find(A==0);

% Indique le nombre de lignes libres par *nFreeRow*

nFreeRow=length(A);

%{

Nous trouvons les indices initiaux des colonnes libres restantes dans la matrice de solution et enregistrons les résultats dans le tableau *B*.

%}

B=find(B==0);

%{

De toute évidence, le nombre de colonnes libres sera égal au nombre de lignes libres

%}

%{

Créer un tableau *L(1:nFreeRow, 1:nFreeRow)* et remplissez toutes les cellules avec un. De plus, si la cellule *L*(p, q) s'avère libre, alors nous écrivons zéro dans cette cellule au lieu d'un.

%}

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

%{

Créons des tableaux *rAr* et *tAr* pour sauvegarder les index de correspondance pour contrôler les tableaux.

%}

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

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

%{

Nous aurons besoin de ces tableaux pour rendre compte de manière équivalente des indices de positions libres dans le tableau *L*, avec les indices correspondants des tableaux de contrôle *D1* et *D2*.

Sur la base des informations sur les lignes et colonnes libres restantes, nous écrivons zéro dans les cellules libres correspondantes du tableau *L*. Dans le même cycle, nous formerons des tableaux de comptabilité *rAr* et *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

%{

Sauvegardez tous les tableaux principaux. Nous en aurons besoin pour le *Back Tracking*, s'il devient nécessaire de revenir au début du Bloc-2 pour des calculs répétés.

%}

Ay=A;

By=B;

D1y=D1;

D2y=D2;

Qy=Q;

Ly=L;

rAr\_y=rAr;

tAr\_y=tAr;

yTotPos=totPos;

%{

Nous avons fait le travail préparatoire. Nous pouvons maintenant passer au Bloc-3.

%}

activeEvent=3;

case 3

%{

Bloc-3. Utilisation de l'algorithme *rand & rand*

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

Dans ce bloc, nous poursuivons l'achèvement de la composition. Ici, un autre algorithme est utilisé, qui s'appelle *rand & rand*. Son essence est la suivante. Dans la liste des lignes libres restantes, un index de ligne aléatoire est sélectionné. Dans la ligne sélectionnée, dans la liste des positions libres, nous sélectionnons au hasard un index. S'il s'avère que la position est libre des restrictions diagonales imposées par toutes les reines précédemment placées, alors la position est considérée comme libre et la reine y est placée.

%}

% Augmentez le compteur du nombre de cas lorsque le Bloc-3 est utilisé.

simCount3=simCount3+1;

%{

S'il s'avère que le nombre de répétitions (*simCount3*) ne dépasse pas la valeur limite de *simBound3*, alors nous continuerons à former une solution sur la base des données collectées dans le tableau *L*.

%}

if simCount3 <= simBound3

while totPos < eventBound2

% Définir des indices de lignes libres dans le tableau *L* en fonction du tableau *A*

freeRowAr=find(A>0);

% Définissez le nombre de lignes libres (*nFreeRow*)

nFreeRow=length(freeRowAr);

% Choisissez un nombre aléatoire (*randNumb*) dans l'intervalle (*1, nFreeRow*).

randNumb=randi(nFreeRow);

%{

Dans la liste des lignes libres *freeRowAr*, nous sélectionnons au hasard l'index des lignes *selectRowInd*

%}

selectRowInd=freeRowAr(randNumb);

%{

Considérons un tableau *L*. Formons une liste d'indices de position libres (*freePosAr*) dans une rangée avec *selectRowInd*. Définissez la taille de cette liste (*nFreePos*)

%}

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

nFreePos=length(freePosAr);

if nFreePos>0

%{

S'il y a une position libre dans la ligne sélectionnée, nous continuons la solution. S'il n'y a pas de positions libres, cela signifie que la branche de recherche a conduit à un blocage. Dans ce cas, nous devons interrompre l'exécution de l'algorithme dans ce bloc et revenir au niveau de base précédent.

%}

%{

S'il y a une position libre dans la ligne, alors nous sélectionnons un nombre aléatoire (*randNumb*) dans l'intervalle (*1,nFreePos*)

%}

randNumb=randi(nFreePos);

%{

Après cela, dans la liste des positions libres (*freePosAr*), nous sélectionnons la position *selectPosInd* sur la base du nombre aléatoire sélectionné *randNumb*.

%}

selectPosInd=freePosAr(randNumb);

%{

Nous avons sélectionné au hasard l'index de ligne libre (*selectRowInd*) et sélectionné au hasard l'index de position libre (*selectPosInd*) dans cette ligne. Toutes ces actions ont été effectuées dans le tableau *L*. Maintenant, nous allons restaurer l'index d'origine de la position sélectionnée sur la base du tableau *B* (c'est l'indice qui correspond à la matrice de données d'origine).

%}

j=B(selectPosInd);

%{

Nous restaurerons également l'index d'origine de la ligne sélectionnée en fonction du tableau *A*.

%}

i=A(selectRowInd);

% Nous enregistrons le résultat (position reine dans la rangée) dans le tableau *Q*

Q(i)=j;

% On incrémente le compteur du nombre de positions occupées par la reine.

totPos=totPos+1;

%{

Nous écrivons 1 dans la cellule *selectRowInd* du tableau de contrôle des lignes libres *A* pour corriger que la ligne correspondante est fermée.

%}

A(selectRowInd)=0;

%{

Nous écrivons 1 dans la cellule *selectPosInd* du tableau *B* pour corriger que la colonne correspondante est occupée.

%}

B(selectPosInd)=0;

%{

Modifiez les cellules correspondantes des tableaux de contrôle diagonaux *D1* et *D2* en utilisant les valeurs réelles des indices (i,j) (qui correspondent à l'échiquier d'origine).

%}

rx=n+j-i;

tx=j+i;

D1(rx)=1;

D2(tx)=1;

%{

Dans toutes les lignes libres du tableau *L* dans la colonne *selectPosInd*, nous écrivons 1 (pour fermer les cellules correspondantes).

%}

L(freeRowAr,selectPosInd)=1;

%{

Important! Nous travaillons avec un tableau *L*, où sont projetées toutes les lignes libres et toutes les colonnes libres de la «*grande*» matrice de données d'origine. Lorsque nous plaçons la reine à la position (i, j) dans la matrice de données initiale, alors, en même temps, devraient être exclus de la suite: la row(i), la column(j) et toutes les cellules de la matrice de données qui se coucher sur les diagonales gauche et droite passant par le point (i, j). Ci-dessus, nous avons exclu la ligne correspondante et la colonne correspondante, en mettant à zéro les cellules correspondantes dans les tableaux *A* et *B*. Maintenant, nous devons par "*projection*" exclure les cellules du tableau *L* qui correspondent aux exceptions diagonales de la matrice de données d'origine. Pour ce faire, nous utilisons les index équivalents correspondants précédemment stockés dans les tableaux *rAr* et *tAr*.

%}

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

%{

Ainsi, nous avons effectué toutes les étapes procédurales associées à la sélection d'une position (i,j) dans la matrice de données d'origine pour l'emplacement de la reine.

%}

else % if freePos>0

%{

S'il n'y a pas de positions libres dans la ligne, cela signifie que nous avons atteint une impasse, nous devons donc fermer cette branche de recherche et revenir au Bloc-3, puis répéter pour former une solution. Avant cela, nous devons restaurer tous les tableaux nécessaires en fonction des sauvegardes. Nous incrémentons le compteur du nombre total de calculs répétés, puisque nous revenons à recalculer.

%}

totSimCount= totSimCount+1;

%{

Sur la base des copies enregistrées, nous restaurerons les valeurs des tableaux nécessaires.

%}

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

% Passons au Block-3 pour le recomptage.

activeEvent=3;

end % if freePos>0

end %while totPos < simBound2

else

%{

S'il s'avère que le nombre de répétitions de *simCount3* dépasse la valeur limite *repeatBound3* et, en même temps, *eventInd == 1*, alors nous devons revenir au niveau de base 1 et reconstruire les branches de recherche. Avant cela, nous devons restaurer tous les tableaux nécessaires qui correspondent à ce point de retour.

%}

if eventInd==1

% On incrémente le compteur total du nombre de calculs répétés.

totSimCount= totSimCount+1;

% Restaurons les tableaux et transférons le contrôle au Bloc-1.

A=Ax;

B=Bx;

D1=D1x;

D2=D2x;

Q=Qx;

totPos=xTotPos;

% Remettez à zéro la valeur du compteur *simCount3*.

simCount3=0;

% Passons au Bloc-1.

activeEvent=1;

else

%{

Si, dans ce bloc, il y avait des répétitions *simBound3*, et dans chaque cas, à un moment donné, il s'est avéré que parmi les lignes libres restantes, il y a une ligne dans laquelle il n'y a pas de position libre, cela signifie que cette composition est négative, et il ne peut pas être complété. Pour cette raison, le programme doit être interrompu. Nous avons mis la variable *compositionInd* à zéro pour corriger que cette composition est négative. Nous avons également mis la variable *processInd* à zéro pour interrompre le programme.

%}

solutionInd=2;

processInd=0;

break

end

end % if simCount3 > simBound3

%{

Une fois les calculs du Bloc-3 réussis, le nombre de reines correctement situées dans la matrice de la solution sera égal à *eventBound3*.

Passons au Bloc-4.

%}

if totPos >= eventBound2

activeEvent=4;

end

case 4

%{

Bloc 4. Préparation des tableaux nécessaires pour travailler dans le bloc-5

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

Nous allons au Block-4 dans trois cas:

1) Immédiatement après la fin du bloc 3, c'est-à-dire si eventInd était égal à 1 ou 2,

2) Si la valeur n <= nFix1,

3) Si la valeur nComp> = eventBound2.

Ce bloc est préparatoire, où nous perpare les tableaux nécessaires avant la transition dans le Bloc-5. Dans une certaine mesure, le fonctionnement de l'algorithme dans ce bloc est similaire au fonctionnement de l'algorithme dans le Bloc-3. Son essence est la suivante. Soit *nRow* le nombre de lignes libres restantes dans la matrice de solution. Nous formons un tableau *L*(1:nRow, 1:nRow) et collectons les données de toutes les lignes et colonnes libres. L'algorithme de génération du tableau *L* est similaire à celui utilisé dans Bloc-2. Comme dans le Bloc-2, nous prendrons en compte la correspondance des indices du tableau *L* avec les indices correspondants de la matrice de solution d'origine. La traduction par projection d'une solution de la matrice d'origine vers une matrice *L* plus petite nous donne la possibilité à chaque étape de trouver efficacement une ligne avec un nombre minimum de positions libres et de réduire considérablement la quantité de calcul. Mais, non moins important est le fait que, sur la base du tableau *L*, nous gardons simultanément le statut de toutes les lignes libres restantes. Cela nous permet de contrôler toutes les lignes et de déterminer si une situation s'est produite lorsque dans l'une des lignes restantes le nombre de positions libres est nul. Dans ce cas, nous excluons la branche de recherche comme impasse. Cette approche nous permet de reporter les prévisions et c'est important. Nous arrêtons le calcul beaucoup plus tôt que le moment où il est "*soudainement*" constaté que cette branche de recherche est bloquée et doit être interrompue.

%}

%{

La transition directe depuis le début du programme vers le Bloc-4 et la transition séquentielle le long de la chaîne Bloc-2 -> Bloc-3 -> Bloc-4 diffèrent par la forme de représentation des tableaux *A* et *B*. Ceci doit être pris en compte.

%}

if eventInd==4

%{

Nous trouvons les indices initiaux des lignes libres restantes dans la matrice de solution et les stockons dans le tableau *A*

%}

A=find(A==0);

% Notons *nRow* le nombre de lignes libres

nRow=length(A);

%{

Nous trouvons les indices initiaux des colonnes libres restantes dans la matrice de solution et les stockons dans le tableau *B*

%}

B=find(B==0);

else

T=find(A>0);

A=A(T);

nRow=length(T);

T=find(B>0);

B=B(T);

end

% Créez un tableau *L*(1:nRow,1:nRow) et remplissez toutes les cellules avec un.

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

%{

Créez des tableaux *rAr* et *tAr*. On y enregistre les indices cellulaires des tableaux de contrôle diagonaux qui correspondent aux positions libres dans le tableau *L.*

%}

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

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

%{

Créons des tableaux pour prendre en compte la liste cumulative des restrictions formées par la diagonale gauche (*D1s*), la diagonale droite (*D2s*) et les projections de colonne (*Bs*).

%}

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

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

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

%{

Sur la base des informations sur les lignes et colonnes libres restantes, nous écrivons zéro dans les cellules libres correspondantes du tableau *L*. Nous formons les tableaux *Cs, Ds, Bs*, ainsi que les tableaux de comptabilité *rAr, tAr*. Pour toutes les lignes (*nRow*) et, par conséquent, pour les positions libres restantes dans ces lignes, nous formons une liste cumulative de restrictions pour les projections diagonales gauche *D1* et droite *D2*, ainsi que pour les projections de la colonne *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

% Nous calculons la somme des éléments de chaque ligne du tableau *L*

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

%{

Nous trions les valeurs de somme dans l'ordre croissant du nombre de positions libres dans la ligne.

%}

[sumSort,rowRangInd]=sort(rowSum);

%{

Ici, dans le tableau *rowRangInd*, les indices de ligne sont stockés séquentiellement avec un nombre croissant de positions libres dans la ligne.

%}

if sumSort(1)>0

%{

Ici *sumSort*(1) est le nombre minimum de positions libres dans la liste de toutes les lignes du tableau *L(nRow,nRow)*.

%}

%{

Si le nombre minimum de positions libres est supérieur à zéro, alors nous continuons la solution et construisons la branche de la recherche.

%}

%{

Créez un tableau de contrôle comptable *E* de taille *nRow x nRow*, dans chaque cellule dont nous allons stocker la valeur totale des restrictions correspondantes.

%}

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

%{

Nous calculons et stockons dans le tableau *E* la valeur totale des contraintes des tableaux comptables de contrôle.

%}

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)=D1s(r)+D2s(t)+Bs(j);

end

end

end

% Supprimer les tableaux qui ne seront plus utilizes.

clear D1s D2s Bs

% Ensuite, au lieu des tableaux *D1, D2* et *Bs*, nous utiliserons le tableau *E.*

%{

Avant de passer à l'événement suivant, nous enregistrerons une copie de ces tableaux pour les réutiliser.

%}

Az=A;

Bz=B;

Qz=Q;

Lz=L;

Ez=E;

zPos=totPos;

%{

Nous avons terminé les travaux préparatoires du Bloc-4. Ensuite, nous allons au Bloc-5.

%}

activeEvent=5;

else % if sumSort(1)>0

%{

S'il s'avère que parmi les rangées restantes, il y a des œufs dans lesquels il n'y a pas de position libre, cela signifie:

a) Si *eventInd = 4*, la composition initiale ne peut pas être terminée, car dans la composition il y a au moins une ligne libre sans position libre. (On peut dire que cette composition est négative depuis la «*naissance*»).

b) Si *eventInd <3*, alors nous devons retourner au Bloc-2 et répéter la formation de la branche de recherche.

%}

if eventInd<3

%{

Si l'index d'événement est 1 ou 2, alors nous revenons au début du Bloc-3. Pour ce faire, avant de passer au Bloc-3, nous allons restaurer l'état initial des tableaux de contrôle que nous avions à la fin du Bloc-2.

%}

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

activeEvent=3;

% On incrémente le compteur total du nombre de calculs répétés.

totSimCount= totSimCount+1;

elseif eventInd > 3

%{

Si *eventInd == 4*, cela signifie que la taille de la composition était telle que nous sommes immédiatement passés à ce niveau, en contournant les niveaux 1, 2 et 3. Et puisque parmi toutes les lignes libres restantes, il y a au moins une ligne dans laquelle il y a pas de position libre, alors cette composition ne peut initialement pas être complétée. Par conséquent, nous affichons le message approprié et interrompons le programme.

Nous avons défini la variable *compositionInd* sur -1 pour corriger que cette composition est initialement négative et ne peut pas être complétée. De plus, nous attribuons zéro à la variable *processInd* pour interrompre la poursuite du fonctionnement du programme

%}

solutionInd=2;

processInd=0;

break

end

end %if sumSort(1)>0

% Après les travaux préparatoires du Bloc-4, nous passons au Bloc-5.

case 5

%{

Bloc-5. L'étape finale de la résolution de problèmes.

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

Nous sommes au dernier niveau de solution de base. Il reste quelques lignes libres jusqu'à la fin de la solution. Si, à partir de ce niveau, en train de résoudre le problème, la branche de recherche mène à un blocage, alors nous reviendrons à ce niveau de base. À cette étape, nous devons choisir une seule position dans une rangée libre, pour l'emplacement de la reine. Dans cette étape, le nombre de possibilités d'un tel choix est égal à la somme des positions libres dans toutes les lignes libres restantes. Les deux boucles imbriquées utilisées dans le Bloc-5 n'ont qu'un seul objectif, sélectionner l'index d'une ligne libre à un niveau donné et sélectionner la position libre dans cette ligne. Toute la recherche supplémentaire des lignes libres restantes est effectuée uniquement dans la troisième boucle imbriquée. Par conséquent, d'abord dans le Bloc-5:

- nous sélectionnons la ligne avec le nombre minimum de positions libres;

- nous sélectionnons une position libre dans cette rangée et plaçons la reine.

Après cela, la séquence d'actions suivante est effectuée dans la troisième boucle:

*a)Parmi les lignes libres restantes, nous sélectionnons une ligne avec le nombre minimum de positions libres,*

b) *Parmi les positions libres de la ligne sélectionnée, nous sélectionnons cette position qui cause un dommage minimal à toutes les positions libres restantes.*

Ce cycle se poursuit jusqu'à l'obtention d'une solution complète. Si, à un moment donné, la branche de recherche aboutit à un blocage, le cycle est interrompu. Sur la base des copies de sauvegarde, tous les tableaux et variables correspondant au niveau de base actuel sont restaurés. Dans ce cas, la troisième boucle imbriquée se répète à nouveau, sans aucune modification des paramètres des première et deuxième boucles imbriquées. Le nombre de ces calculs répétés au niveau de la troisième boucle imbriquée ne doit pas dépasser la valeur limite de *repeatBound*. Si le nombre de répétitions dépasse la valeur de *repeatBound*, dans ce cas, après le retour au niveau de base, les paramètres des deux premières boucles imbriquées changent comme d'habitude. L'utilisation d'un tel modèle de trois boucles imbriquées n'est pas entièrement évidente à première vue. Le fait est que dans les cas où il y a plusieurs lignes avec la même valeur minimale du nombre total de positions libres, nous sélectionnons au hasard l'indice de l'une de ces deux lignes (ou un indice aléatoire de l'une des trois lignes si trois lignes ont la même valeur minimale). De même, une sélection aléatoire d'une position libre dans une rangée est effectuée si deux positions dans une rangée causent le même dommage minimal à toutes les positions libres restantes. (Ici, une sélection aléatoire est faite de seulement deux positions qui causent le même dommage minimal). Nous utilisons un tel algorithme avec un seul but afin de maximiser l'utilisation des "*ressources de tâches*" qui restent à cette étape. Plus la fin de la solution est proche, moins il est probable que la ligne libre sélectionnée aura une position libre. Selon la règle *du risque minimum*, nous devons d'abord placer la reine dans cette rangée libre, où le nombre de positions libres est minime. Voilà ce que nous faisons. Mais dans les situations où deux lignes ou deux positions libres ont les mêmes caractéristiques minimales, nous sélectionnons cet indice de manière aléatoire. Lorsque le troisième cycle imbriqué est répété plusieurs fois sans modifier les paramètres du cycle, cela nous donne la possibilité d'utiliser davantage de "*capacités de ressources*" de la tâche à ce niveau, car à certaines étapes de la formation de la branche de recherche, une sélection aléatoire est utilisée.

%}

% Remettez à zéro le compteur du nombre de calculs répétés dans le Bloc-5

simCount5=0;

%{

Le cycle *for iRow = 1: nRow* sert à l'analyse séquentielle des lignes libres restantes, classées par ordre croissant du nombre total de positions libres dans la ligne. Les indices des lignes correspondantes sont stockés dans le tableau *rowRangInd(1:nRow)*. Ici *nRow* est le nombre de lignes libres restantes. Le tableau *rowRangInd* des calculs correspondants a été effectué dans le Bloc-4.

%}

for iRow=1:nRow % Première boucle imbriquée (externe)

% Choisissez une ligne dans la liste classéeю

selectRowInd=rowRangInd(iRow);

%{

*selectRowInd* est l'index de ligne dans le tableau *L*. Déterminons la valeur initiale de l'index de ligne sur l'échiquier, qui dans le tableau *L* correspond à l'index *selectRowInd*.

La valeur de *baseRowInd* sera nécessaire ultérieurement pour des calculs répétés.

%}

baseRowInd=A(selectRowInd);

% Copiez la ligne avec index selectRowInd du tableau *L* dans le tableau temporaire *T*

T=L(selectRowInd,:);

%{

Définissez les indices de position libre dans cette ligne et enregistrez le résultat dans le tableau *baseFreePosAr* (encore une fois, notez que les positions nulles dans le tableau *L* correspondent aux positions libres dans la matrice de solution d'origine)

%}

baseFreePosAr=find(T==0);

% Définissez le nombre total de positions libres (*nFreePos*) dans cette ligne

nFreePos=length(baseFreePosAr);

%{

La boucle *for jCol = 1: nFreePos* est utilisée pour les positions libres d'analyse séquentielle dans la ligne.

%}

for jCol=1:nFreePos % Boucle imbriquée-2

% Attribuer i l'indice réel de la ligne sélectionnée

i=baseRowInd;

%{

Dans le tableau *baseFreePosAr*, nous sélectionnons l'index de la colonne, qui est écrit dans la cellule avec le nombre *jCol*. Ici *jPos* est l'indice de colonne du tableau *L.*

%}

jPos=baseFreePosAr(jCol);

jPosBase=jPos;

%{

On détermine la valeur réelle de l'index de colonne (j), qui correspond à l'échiquier en question

%}

j=B(jPos);

% Enregistrez la valeur de j dans la variable *baseFreePos* pour des calculs répétés

baseFreePos=j;

%{

Attribuez à la variable *minRowInd* la valeur de l'index de ligne du tableau *L*, qui a le nombre minimum de positions libres dans la ligne.

%}

minRowInd=selectRowInd;

% Zéro le nombre de tentatives de la troisième boucle imbriquée.

repeatCount=0;

%{

La boucle while totPos <n est la troisième boucle imbriquée, où, à chaque étape, une position libre est recherchée pour que la reine soit située dans l'une des lignes libres restantes.

%}

while totPos < n % Boucle imbriquée-3

%{

La valeur d'index initiale de la ligne sélectionnée (i) et la valeur d'index de colonne (j) pour la première étape, nous avons déterminé ci-dessus (dans le Bloc 4), avant d'entrer dans le cycle. Dans le tableau *Q* de la ligne (i), nous enregistrons l'indice (j) de la position reine.

%}

Q(i)=j;

% On incrémente le compteur du nombre de reines installées.

totPos=totPos+1;

% Vérifiez si une solution complète est formée, puis arrêtez les calculs.

if totPos==n

solutionInd=1;

processInd=0;

break

end

%{

Nous avons utilisé le résultat préparé dans le Bloc-4 et placé la reine dans la cellule (i,j) de la matrice de solution. Ainsi, nous avons terminé le cycle suivant de détermination de la position sur l'échiquier pour l'emplacement de la reine. Après cela, nous devons changer les cellules correspondantes dans tous les tableaux de contrôle, étant donné les indices *(minRowInd, jPos)* du tableau *L*.

%}

A(minRowInd)=0;

B(jPos)=0;

%{

Modifier les cellules correspondantes du tableau *L* en utilisant les index équivalents stockés dans les tableaux *rAr* et *tAr*.

%}

rx=n+j-i;

tx=j+i;

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

%{

Nous décrémentons la valeur du tableau de contrôle cumulatif *E*, car nous avons placé la reine à la position (i,j).

%}

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

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

% Écrivez 1 à toutes les cellules actives dans la colonne *jPos* du tableau *L*

A1=find(A>0);

L(A1,jPos)=1;

%{

À cette étape, à l'intérieur de la boucle *while totPos <n*, nous avons effectué les actions suivantes:

- nous plaçons la reine dans la cellule (i,j), en utilisant les informations préparées précédemment;

- effectué les actions procédurales nécessaires avec des tableaux de contrôle, après que la reine est placée dans la cellule (i,j).

%}

%{

Sélection d'une rangée libre et d'une position libre dans la rangée

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

Maintenant, parmi les lignes libres restantes, nous trouvons la ligne avec le nombre minimum de positions libres, et parmi ces positions nous choisissons celle qui, dans le cas de la fermeture de la position, causera un dommage minimal à toutes les positions libres restantes dans les autres positions libres Lignes. Pour le faire, suivez ces étapes:

%}

%1.Définissez le nombre de positions libres dans chaque rangée libre restante

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

%{

2. Nous classons le tableau rowSum dans l'ordre croissant.

3. Enregistrez les valeurs classées des sommes dans le tableau *freePosAr* et les indices des lignes correspondantes dans le tableau *rowIndAr*.

%}

[freePosAr,rowIndAr]=sort(rowSum);

%{

Étant donné qu'à ce stade, nous suivons simultanément l'état de toutes les lignes libres restantes, cela nous donne l'opportunité d'établir si une telle situation s'est produite que dans l'une des lignes restantes, le nombre de positions libres est nul. Dans ce cas, nous considérons la branche de recherche générée comme un blocage et revenons au début du cycle.

Cette approche nous permet de reporter la prévision - nous arrêtons les calculs avant qu'il ne soit établi à l'étape suivante qu'il n'y a pas de position libre dans la ligne.

%}

%{

Voici le point de contrôle pour la branche de recherche générée. Si, dans chacune des lignes libres restantes, il y a au moins une position libre, alors la formation de la branche de recherche continue.

%}

if freePosAr(1)>0

%{

Il se peut que dans une liste classée, les deux premiers éléments de la liste ou les trois premiers éléments de la liste aient la même valeur minimale. Dans ce cas, nous sélectionnons au hasard l'indice de l'une des deux lignes avec la même valeur minimale (ou, l'indice de l'une des trois lignes, s'il y en a trois).

%}

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);

%{

Nous déterminons le nombre de positions libres dans la ligne sélectionnée et stockons les indices de ces lignes dans le tableau *freePosAr.*

%}

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

% Définir le nombre de positions libres (*nFreePos*)

nFreePos=length(freePosAr);

%{

Parmi ces positions, nous choisissons celle qui ferme le nombre minimum de positions libres dans les rangées restantes. Pour ce faire, nous utilisons le tableau E. Si deux lignes ont le même nombre minimum de positions libres, nous sélectionnons au hasard l'indice *jPos* de l'une d'entre elles. *Ici, nous introduisons un élément de hasard «sain» dans l'algorithme dans tous les cas où deux rangées ont le même nombre de positions libres*.

%}

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

%{

Sur la base du tableau d'indices source *A*, nous restaurons l'index réel i ligne donnée.

%}

i=A(minRowInd);

%{

Sur la base du tableau des indices source *B*, nous restaurons l'indice réel j cette colonne

%}

j=B(jPos);

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

%{

S'il s'avère qu'il n'y a pas de positions libres dans la ligne, cela signifie que la branche de recherche a conduit à un blocage. Dans ce cas, nous fermons la branche de recherche et incrémentons les compteurs: *repeatCount, simCount5, totSimCount*.

%}

repeatCount=repeatCount+1;

simCount5=simCount5+1;

totSimCount=totSimCount+1;

%{

Nous allons restaurer les valeurs de tous les tableaux de contrôle (au début de l'exécution du cycle *while totPos <n*) et transférer le contrôle au début du cycle. Si le nombre d'utilisations répétées du cycle (alors *while totPos < n*) ne dépasse pas la valeur de seuil *repeatBound*, alors le contrôle est transféré au début du cycle *while totPos < n*, sans changer les paramètres de deux cycles externes

%}

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

i=baseRowInd;

j=baseFreePos;

minRowInd=selectRowInd;

jPos=jPosBase;

if repeatCount>repeatBound

%{

Si le nombre de réutilisation de boucle (*while totPos < n*) dépasse le seuil *repeatBound*, le contrôle est transféré vers la boucle externe *while jCol<=colPos*. Ci-dessus, nous avons restauré les paramètres correspondants pour la transition.

%}

repeatCount=0;

i=baseRowInd;

break

% Sortie de la boucle *while totPos <n*

% Exiting the loop *while totPos <n*

% Aller à la suite du cycle while jCol<=colPos

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

end

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

end %while totPos < n

if processInd==0

break

% Sortie de la boucle *while jCol<=colPos*

% Exiting the *while jCol <= colPos* loop

end

end %while jCol<=colPos

if processInd==0

break

% Sortie de la boucle for iRow=1:nRow

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

end

%{

Ici, à l'intérieur de la boucle *for iRow = 1: nRow*, se trouve le seul endroit où nous contrôlons *totSimCount*. Si la valeur de *totSimCount* dépasse la valeur de *totRepeatBound*, un message s'affiche indiquant que la probabilité que cette composition puisse être complétée jusqu'à ce que la solution complète soit inférieure à *0,0001*.

%}

if totSimCount > totSimBound

solutionInd=3;

break

%{

Soit *totSimCount <= totSimBound*. Ensuite, si, après avoir quitté le cycle *while jCol <= colPos*, il s'avère que le nombre de nouveaux tests à ce niveau (*simCount5*) dépasse les limites autorisées (*simBound5*), alors dans le cas *eventInd <3*, nous transférons le contrôle à l'événement 2. Si *eventInd == 4*, le programme continue, sous réserve de 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

%{

Si *solutionInd == 3*, alors nous incrémentons le compteur de cycle complet

*falseNegSimCount*.

Si la valeur du compteur ne dépasse pas le *falseNegSimBound*, nous retournons le contrôle au début du recomptage, selon la valeur de l'événement

%}

if solutionInd==3

if falseNegSimCount < falseNegSimBound

falseNegSimCount=falseNegSimCount+1;

switch eventInd

case 1

% Restaurer les baies et transférer le contrôle à l'événement 1

A=Ax;

B=Bx;

D1=D1x;

D2=D2x;

Q=Qx;

totPos=xTotPos;

activeEventInd=1;

case 2

% Restaurer les baies et transférer le contrôle à l'événement 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

% Restaurer les baies et transférer le contrôle à l'événement 5

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

activeEventInd=5;

end

% remise à zéro des compteurs pour les événements correspondants

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 queens:',nDisp); disp(tStr)

disp(Q(1:nDisp));

%{

Nous enregistrerons le résultat de l'achèvement dans le fichier *nQueens\_Test\_Completion\_Solution.mat*. Si à la suite de la solution, il n'était pas possible de compléter la composition à la solution complète, alors les valeurs nulles sont enregistrées dans les cellules correspondantes du tableau *Q*.

Le nom de fichier *nQueens\_Test\_Completion\_Solution.mat* est donné à titre d'exemple. Évidemment, vous pouvez utiliser n'importe quel autre nom.

%}

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

if solutionInd == 1

save(outputFileName,'Q');

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

disp(iInfo);

end