%Solution\_n\_Queens\_Completion\_Problem

%{

Программа предназначена для комплектации до полного решения произвольной композиции из *k* ферзей. Под композицией мы понимаем случайное распредление *k* ферзей на

произвольной шахматной доске с размером *n x n*, таким образом, что выполняются три условия данной задачи: *в каждой строке, в каждом столбце, а также на левой и правой диагоналях, проходящих черз позицию, где расположен ферзь, не располагается более одного ферзя.*

Нужно найти хотя бы одно решение и тем самым показать, что решение существует, либо вывести суждение, что с заданной вероятностью (*P*), данная композиция не может быть комплектована до полного решения.

%}

%{

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”.

%}

%{

Автор проекта и разработчик – Григорян Эрос (EricGrig), 2020

Буду рад, если какие-либо участки кода, или вся программа в целом, будут использоваться для научных целей, или для образования. При этом, буду благодарен, если сочтете возможным сослаться на мою публикацию. Это элемент культуры и знак взаимного уважения.

Для использования в коммерческих целях любого участка кода программы, или всей программы в целом, необходимо письменное согласие автора.

%}

%{

Результаты исследования, связанные с разработкой данного алгоритма, опубликованы в arxiv.org в статье: *Grigoryan E., Linear algorithm for solution n-Queens Completion problem,* [*https://arxiv.org/abs/1912.05935*](https://arxiv.org/abs/1912.05935) *.* Будет правильно, если вначале вы ознакомитесь с данной публикацией, прежде чем приступите к изучению исходного кода программы. Это сделает описание программы более прозрачным и уменьшит количество возможных вопросов.

Русскоязычная версия статьи опубликована на сайте сообщества программистов habr.com: [*https://habr.com/ru/post/483036/*](https://habr.com/ru/post/483036/)

%}

%{

1. Начало

---------

Как подготовлены исходные данные?

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

Обозначим через *n* размер стороны шахматной доски.

Пусть имеется одномерный обнуленный массив размера *n*. Если в i-ой строке

шахматной доски расположен ферзь в позиции j, то, соответственно,

в i-ую ячейку одномерного массива данных записывается значение j.

Далее, наряду с названием "*шахматная доска с размером n x n*" будем

использовать название "*матрица решения размером n x n*"

Проведем считывание файл данных исходной композиции и сохраним результат

в массиве *Q*. Здесь, в качестве примера используется имя файла данных

*kQueens\_Test\_Composition.mat* . Это имя необходимо заменить на то имя, которое соответсвует вашему файлу данных

%}

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

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

disp(iInfo);

% Input data file

Q=importdata(inputFileName);

n=length(Q);

%{

Вывод на экран первых 50-ти строк композиции (или всей композиции, если n < 50

%}

if n<50

nDisp=n;

else

nDisp=50;

end

disp(Q(1:nDisp));

%{

Определим количество нулевых ячеек в массиве *Q*, которую обозначим через

*nZero*. Тем самым определим количество свободных строк в матрице решения

%}

nZero=sum(Q==0);

% Обозначим размер композиции через *nComp*

nComp=n-nZero;

%{

Выведем для пользователя необходимую информацию по данной задаче: размер матрицы решения, размер композиции, число свободных позиций

%}

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)

%{

Вывод позиций ферзей в первых 50-ти строках композиции, либо позиции во всех строках, есди n <50

%}

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

disp(Q(1:nDisp));

%{

Если окажется, что размер композиции равен размеру шахматной доски, то

выведем соответсвующее сообщение и завершим выполнение программы.

%}

if nComp == n

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

disp(tStr);

pause

exit

end

%{

Если окажется, что размер композиции равен нулю, то есть нет композиции, то выведем соответсвующее сообщение и завершим выполнение программы.

%}

if nComp==0

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

disp(tStr);

exit

end

%{

Общее

-----

В процессе исследования, было разработано три основных алгоритма

решения задачи, которые различаются как по скорости решения задачи,

так и по эффективности. Также в программе реализованы участки кода,

которые выполняют подготовительные функции для основных алгоритмов.

Обозначим эти участки кода соответственно: Блок-0, Блок-1,..., Блок-5.

*Блок- 0* -- начальный блок. Проверка правильности композиции, подготовка контрольных

! массивов

!

*Блок- 1* -- формирование решения на основе алгоритма *rand\_set & rand\_set. Г*граничное

!значение числа правильно установленных ферзей равно *eventBound2*

!

*Блок- 2* -- подготовительные операции для перехода в Блок-3

!

*Блок- 3* -- формирование решения на основе алгоритма *rand & rand*

! Граничное значение числа правильно установленных ферзей равно *eventBound3*

!

*Блок- 4* -- подготовительные операции для перехода в Блок-5

!

*Блок- 5* -- формирование решения на основе правил «*минимального риска*» и

«*минимального ущерба*»

В процессе поиска решения, расчет передается от одного блока

алгоритма в другой как эстафетная палочка.

Алгоритм разрабатывался для достаточно широкого диапазона значений

размера шахматной доски: от значения *7* до *100 000 000*. Если позволяет

размер оперативной памяти, то можно вести расчеты и для больших значений

размера шахматной доски, например для *n = 800 000 000*. (На компьютере с *ОЗУ = 32 Gb* задача комлектации была решена для *n = 1 000 000 000*. Однако, из-за того, что слегка не хватило памяти, пришлось немного изменить программу и провести расчеты в два этапа).

%}

if n<5

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

disp(tStr);

exit

end

%{

Если *nComp <= eventBound2*, то расчеты начинаются с блока 1

Если *eventBound2 < nComp <= eventBound3*, то расчеты начинаются с блока 2

Если *nComp > eventBound3*, то управление передается в блок 4, где

выполняется подготовительная работа и после этого совершается переход

в блок 5 для выполнения основных расчетов

%}

%{

Как показали результаты исследования, на участке значений *n=(7,...,99)*

требуется более скурпулезный подход к формированию ветви поиска решения.

Поэтому этот интервал был разделен на два подинтервала *(7,...,49)* и

*(50,...,99)*, в каждом из которых, используется соотвествующий алгорим

решения. (Здесь граничные значения можно незначительно увеличить, или

уменьшить. От этого суть решения не изменится)

%}

% *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.

%}

%{

О граничных значениях *eventBound2* и *eventBound3*:

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

Если процесc формирования ветви поиска решения приводит в тупик, то необходимо вернуться назад (*Back Tracking*), на какой-то из предыдущих уровней, и снова начать поиск решения. Для этого, мы должны знать на какой из предыдущих уровней следует возвращаться, чтобы заранее сохранить значения основных параметров этого уровня. Выбор оптимальной точки возврата является достаточной сложной и интересной задачей. В данном алгоритме мы используем следующее правило. Наряду с начальным уровнем, который соответствует завершению проверки композиции и формированию всех контрольных массивов, мы формируем и используем еще два основных базовых уровня для возврата назад: *eventBound2* и *eventBound3*. Здесь, *учетной мерой уровня решения, является количество ферзей, правильно расставленных на шахматной доске*.

%}

%{

О числе повторных вычислений на последнем базовом уровне

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

Самые большие трудности в работе алгоритма возникают на последнем этапе решения задачи. Все скрытые ошибки, которые были допущены при выборе индекса свободной строки, и (или) выборе свободной позиции в этой строке, "постепенно накапливаются", и на последнем этапе проявляются в том, что среди оставщихся свободных строк, находится хотя бы одна строка, в которой нет ни одной свободной позиции. Это означает тупик. Поэтому, алгоритм формирования ветви поиска решения на последнем этапе является болем скурпулезным. (В качестве аналогии, здесь уместно использовать сравнение микрохирургии,с обычной хирургией).

В стремлении учесть, возможные эффективные пути формирования ветви поиска решения, на последнем этапе, внутри двух вложенных циклов мы выполняем третий цикл, который повторяется несколько раз, с возвратом на начало этого же цикла, без изменения параметров двух внешних циклов. Это похоже на применение процедуры *Back Tracking* внутри системы вложенных циклов. Пороговое значение числа поаторений, который можно произвести внутр. этого цикла мы обозначаем через *repeatBound*. Далее, по тексту, об этом будет сказано чуть более подробно.

%}

%{

Вычислим значения *eventBound2* и *eventBound3*, а также пороговое значение

количества повторений *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

%{

Эмпирические значения парамтров *eventBound2* и *eventBound3* установлены на основе очень большого числа вычислительных экспериментов и оптимизированы для всего диапазона решения задачи. Для любого меньшего диапазона значений *n*, эти параметры можно незначительно изменить и получить такие значения, при которых программа будет работать чуть быстрее.

В процессе решения задачи, если возникает тупиковая ситуация, то некоторые блоки алгоритма выполняются повторно. Причем, в зависисмости от значений *n* и *nComp*

повторные вычисления начинаются с самого начала либо с некоторого достигнутого уровня. Если повторный поиск решения на верхних уровнях не приводит к успеху, то повторный поиск решения начинается на болем низких уровнях. Здесь, пременные

*simBound3*, *simBound5* определяют максимальное число повторных вичислений в пределах Blocks-3 и Blocks-5. *totSimBound* - определяет суммарное количество всех повторных вычислений на всех уровнях

%}

simBound3=5;

simBound5=100;

totSimBound=1000;

%{

Для работы алгоритма используются несколько рабочих массивов:

*A* - для контроля индексов строк,

*B* - для контроля индексов столбцов.

%}

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

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

%{

Также, для контроля ячеек диагональных проекций используются два

массива: *D1*(1:n2), *D2*(1:n2), где n2 - размер контрольных массивов

%}

n2=2\*n;

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

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

%{

Выбор индекса активного события *eventInd.*

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

Определим индекс блока с которого будет начата работа программы. Для этого присвоим соответствующее значение переменной *eventInd.* Также определим пороговое значение для числа повторных вычислений на последнем этапе (*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. Проверка правильности входных данных композиции

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

Проверятся правильность композиции и при этом последовательно заполняются соответсвующие ячейки контрольных массивов *A, B, C* и *D*.

В ячейках массива *Q*(i), сохраняются индексы позиций правильно установленных ферзей для соответствующих строк. Инкрементируется переменная *totPos* для учета количества правильно установленных ферзей.

%}

%{

Определим индексы занятых строк в массиве *Q* и сохраним результаты в массиве *qPosInd*

%}

qPosInd=find(Q>0);

% Запишем еденицу в те ячейки массива *B*, которые соответствут занятым столбцам.

B(Q(qPosInd))=1;

% Определим сумму единиц в массиве *B*

s=sum(B);

%{

Проверим, не расположены ли два разных ферзя в одном и том же столбце.

Если да, то существует ошибка в исходной композиции. В этом случае

выведем соответствующее сообщение и прервем дальнейшую работу программы.

%}

if s~=n-nZero

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

exit

end

%{

Алгоритм проверки работает следующим образом. Если ячейка (i,j)), где j=*Q*(i) свободна, с учетом диагональных ограничений, и ограничений по числу элементов в каждом столбце, то ферзь расположен в данной ячейке правильно. Правило «*не более одного ферзя в строке*» мы не проверяем, так как модель подготовки исходных данных исключает возможность появления в композиции более одного ферзя в строке. Каждая ячейка в 1-мерном массиве входных данных характеризует соответствующую строку в матрице решения.

%}

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

%{

Если в композиции будет обнаружена ошибка, т.е. расположение ферзей не будет

соответствовать условиям задачи, то будет выведено соответствующее сообщение, и работа программы будет прервана.

%}

if qError==0

A(qPosInd)=1;

totPos=nComp;

else

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

disp(tStr)

exit

end

% Удалим массив *qPosInd*, так как далее мы его не будем использовать.

clear qPosInd

%{

Сохранение копий сформированных массивов для повторного использования

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

Мы выполнили некоторую подготовительную работу. Организовали ввод исходных данных и проверили валидность рассматриваемой композиции. Сохраним копии сформированных массивов. Если будет необходимость снова вернуться на данный уровень и повторно обработать рассматриваемую композицию, то мы восстановим все необходимые массивы на основе этих резервных копий. Данный уровень - это начальный (нулевой) базовый уровень, откуда начинается формирование ветви поиска решения. Здесь количество правильно установленных ферзей равно размеру исходной композиции.

%}

if eventInd==1

Ax=A;

Bx=B;

D1x=D1;

D2x=D2;

Qx=Q;

xTotPos=totPos;

end

%{

Обнулим счетчики числа повторений для третьего(*simCount3*) и пятого

(*simCount5*) уровней.

%}

simCount3=0;

simCount5=0;

%{

*simCount3* далее будет использоваться в качестве накопительного переключателя для формирования переходов в Блоке-3.

%}

%{

Обнулим *totSimCount* - счетчик совокупного числа всех повторений на различных уровнях.

%}

totSimCount=0;

%{

Все события разворачиваются внутри цикла *while processInd==1*, пока не будет получено решение для данной композиции, либо будет установлено, что решение не существует с вероятностью *P*. Главным критерием для такой оценки является общее число всех повторных вычислений(*totSimCount*). В статье, которая приведена в комментарии в начале данной программы, достаточно подробно об этом написано.

В результате проведения большого числа вычислительных экспериментов, для большого разнообразия случайных композиций произвольного размера *k* и для различных значений размера шахматной доски *n*, было установлено, что если суммарное число повторных вычислений(*totSimCount*) превышает пороговое значение *totSimBound*, и при этом решение не найдено, то рассматриваемая композиция не может быть комплектована до полного решения. Вероятность ошибки такого суждения составляет *0.0001*

%}

%{

Начало формирования ветви поиска решения

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

Как было сказано выше, мы рассматриваем различные блоки программы как отдельные события.

Есть пять таких событий. Три из них соответствуют основным блокам программы, а два события соответствуют программным блокам, которые выполняют подготовительные

функции. Присвоим переменной *activeEvent* индекс события, который в данный момент активен.

%}

activeEvent=eventInd;

%{

Введем в рассмотрение переменную *processInd* - в качестве "переключателя" для выхода из цикла. Цикл выполняется, если *processInd==1*, иначе выполнение цикла прерываетсяю.

%}

processInd=1;

%{

Введем в рассмотрение переменную *compositionInd.* Если *compositionInd==1*, то это будет означать, что рассматривая композиция положительная, т.е. может быть комплектована до полного решения. Иначе, композиция будет рассматриваться как отрицательная, т.е. она не может быть комплектована до полного решения.

Если *compositionInd==-1*, то рассматривая композиция является

отрицательной "*по рождению*". Это означает, что в массиве входных данных,

среди свободных строк этой композиции, изначально есть хотя бы одна строка,

в которой не осталось ни одной свободной позиции, все позиции закрыты за счет запретов, сформированных ранее установленными ферзями.

%}

solutionInd=0;

% Начало основного цикла

while processInd==1

% Переменная *active Event* служит в качестве переключателя между 5-ю событиями

switch activeEvent

case 1

%{

Блок-1. Использование алгоритма *rand\_set & rand\_set*

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

В данном блоке производится поиск свободной строки и свободной позиции в этой строке для расположения ферзя до тех пор, пока общее количество правильно установленных ферзей не будет равно пороговому значению *eventBound2*.

Алгоритм, который выполняется в данном блоке называется *rand\_set & rand\_set*. Суть ее состоит в следующем. Найдем индексы всех свободных строк. Проведем случайную перестановку этих индексов. Аналогично найдем индексы всех свободных столбцов. Также проведем случайную перестановку этих индексов. Последовательно рассмотрим пары индексов из этих двух списков (индекс случайной строки, индекс случайного столбца). Если ячейка матрицы решения, соответствующая этой паре индексов, не противоречит диагональным ограничениям, то в данню позицию устанавливается ферзь. В этом случае, в соответствующие ячейки контрольных массивов *A, B, D1* и *D2* записывается единица и счетчик общего числа правильно установленных ферзей(*totPos*) увеличивается на единицу.

%}

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

%{

В данном блоке позиции для расположения ферзей определяются стремительно быстро. Это своего рода разгонный блок. И, хотя все позиции определяются правильно, однако общая картина распределения ферзей в матрице решения является "грубым". Если вовремя не остановиться на каком-то определенном шаге, то дальнейшее построение ветви поиска решения с большой вероятностью приведет в тупик.

Учитывая высокую скорость работы алгоритма *rand\_set & rand\_set*, мы на основе данного блока проходим максимальный путь от значения *nComp* до значения *eventBound2*. После этого эстафета выполнения программы передается другому блоку.

%}

%{

Важно! Данный блок выполняется только в том случае, если *n >= 100* и при этом размер рассматриваемой композиции меньше *eventBound2*. Как показали результаты почти двух десятков миллионов вычислительных экспериментов, при заданном значения *eventBound2*, данный алгоритм всегда комплектует композицию до значения *eventBound2*. Ни разу не было ситуации, чтобы алгоритм зациклился. Это связано с тем, что значение *eventBound2* не является критическим, и есть много разных возможностей для того, чтобы достичь этот уровень. По этой причине, мы исключили из алгоритма контроль

завершения цикла, хотя в ранних версиях программы такая возможность учитывалась. *Жажда скорости оказалась выше логики охвата почти невозможных допустимых ситуаций*.

%}

%{

Когда количество правильно расположенных ферзей (*totPos*) оказывается равным значению *eventBound2*, управление событием передается в Блок-2.

%}

activeEvent=2;

case 2

%{

Блок-2. Подготовка необходимых массивов для работы в Блоке-3

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

В данном блоке выполняется подготовительная работа для перехода в Блок-3. Суть ее состоит в следующем: Пусть количество оставшихся свободных строк равно nFreeRow. Сформируем массив L(1:nFreeRow,1:nFreeRow) и соберем в ней индексы свободных позиций всех оставшихся строк. Это означает следующее: в исходной матрице решения мы рассматриваем сетку пересечения свободных столбцов и свободных строк.Такое пересечение назовем проекционным пересечением. Все такие ячейки на сетке пересечения мы переносим в массив меньшего размера L(1:nFreeRow,1:nFreeRow). При этом, мы ведем учет соответствия индексов массива L с соответствующими индексами исходной матрицы решения.

%}

%{

Найдем исходные индексы оставшихся свободных строк в матрице решения и

сохраним результаты в массиве *A*.

%}

A=find(A==0);

% Обозначим количество свободных строк через *nFreeRow*

nFreeRow=length(A);

%{

Найдем исходные индексы оставшихся свободных столбцов в матрице решения

и сохраним результаты в массиве *B*.

%}

B=find(B==0);

%{

Очевидно, что число свободных столбцов будет равно числу свободных строк

Создадим массив *L*(1:nFreeRow,1:nFreeRow) и заполним все ячейки единицей.

Далее, если ячейка *L*(p,q) окажется свободной, то в эту ячейку вместо

единицы запишем ноль.

%}

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

%{

Создадим рабочие массивы *rAr* и *tAr* для сохранения индексов соответствия контрольным масивам.

%}

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

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

%{

Эти массивы нам будут нужны для эквивалентного учета индексов свободных позиций в массиве *L*, с соответсвующими индексами контрольных массивов *D1* и *D2*. На основе информации об оставшихся свободных строках и свободных столбцах, запишем нуль в соответствующие свободные ячейки массива *L*. В том же цикле сформируем массивы учета *rAr*, *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

%{

Создадим резервные копии всех основных массивов. Они нам будут нужны, если возникнет необходимость вернуться в начало Блока-2 для повторних расчетов

%}

Ay=A;

By=B;

D1y=D1;

D2y=D2;

Qy=Q;

Ly=L;

rAr\_y=rAr;

tAr\_y=tAr;

yTotPos=totPos;

% Мы выполнили подготовительную работу. Теперь можем перейти в Блок-3.

activeEvent=3;

case 3

%{

Блок-3. Использование алгоритма *rand & rand*

В данном блоке продолжается комплектация композиции. Здесь используется другой алгоритм формирования ветви поиска решения, который получил название *rand & rand*. Суть ее состоит в следующем. Из списка оставшихся свободных строк, отбирается случайный индекс строки. В пределах выбранной строки, из списка свободных позиций отбирается случайный индекс позиции. Если окажется, что рассматриваемая позиция свободна от диагональных ограничений, которые накладывают все ранее расположенные ферзи, то позиция считается свободной и в ней распологается ферзь.

%}

% Инкременируем счетчик учета числа случаев, когда используется Блок-3.

simCount3=simCount3+1;

%{

Если окажется, что количество повторений *simCount3* не превышает граничное значение *simBound3*, то продолжим формирование ветви поиска решения на основе данных, собранных в массиве *L.*

%}

if simCount3 <= simBound3

while totPos < eventBound2

% Определим индексы свободных строк в массиве *L* на основе массива *A*

freeRowAr=find(A>0);

% Определим количество свободных строк(*nFreeRow*)

nFreeRow=length(freeRowAr);

% Выберем случайное число(*randNumb*) в интервале (1, *nFreeRow*).

randNumb=randi(nFreeRow);

% Из списка свободных строк(freeRowAr) выберем индекс случайной строки *selectRowInd*

selectRowInd=freeRowAr(randNumb);

%{

Рассмотрим массив *L*. Сформируем список индексов свободных позиций (*freePosAr*) в случайно отобранной строке синдексом *selectRowInd*. Определим размер этого списка(*nFreePos*)

%}

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

nFreePos=length(freePosAr);

if nFreePos>0

%{

Если в выбранной строке имеются свободные позиции, то продолжаем решение.

Если свободных позиций нет, то это означает, что ветвь поиска привела в тупик. В этом случае, мы должны прервать выполнение алгоритма в данном блоке и вернуться на предыдущий базовый уровень.

%}

%{

Если в рассматриваемой строке имеются свободные позиции, то выбираем

случайное число(*randNumb*) в интервале (1, *nFreePos*)

%}

randNumb=randi(nFreePos);

%{

После этого, из списка свободных позицій *freePosAr* выбираем позицию *selectPosInd*, которая соответствует выбранному случайному числу*randNumb.*

%}

selectPosInd=freePosAr(randNumb);

%{

Мы случайным выбрали индекс свободной строки(*selectRowInd*) и случаймо выбрали индекс свободной позиции(*selectPosInd*) в этой строке. Все эти действия выполнялись в пределах массива *L*. Теперь, восстановим исходный индекс отобранной позиции на основе массива *B* (это тот индекс, который соответсвует исходной матрице данных).

%}

j=B(selectPosInd);

% Также восстановим исходный индекс отобранной строки на основе массива *A*

i=A(selectRowInd);

% Сохраним полученный результат(позицию ферзя в строке) в массиве решений *Q.*

Q(i)=j;

% Инкрементируем счетчик учета количества позиций, занятых ферзем.

totPos=totPos+1;

%{

Запишем 1 в ячейку *selectRowInd* массива контроля свободных строк *A*, чтобы зафиксировать, что соответствующая строка занята

%}

A(selectRowInd)=0;

%{

Запишем 1 в ячейку *selectPosInd* массива *B*, чтобы зафиксировать, что соответствующий столбец занят.

%}

B(selectPosInd)=0;

%{

Изменим соответствующие ячейки массивов диагонального контроля *D1* и *D2*,

используя реальные значения индексов (i,j) (которые соответсвуют рассматриваемой шахматной доске)

%}

rx=n+j-i;

tx=j+i;

D1(rx)=1;

D2(tx)=1;

%{

Во всех свободных строках массива *L* в столбце *selectPosInd* запишем 1 (чтобы закрыть соответствующие ячейки)

%}

L(freeRowAr,selectPosInd)=1;

%{

Важно! Мы работаем с массивом *L*, куда проекционно сведены все свободные строки и все свободные столбцы из исходной "большой" матрицы данных. Когда мы распологаем ферзь в найденную позицию (i,j) в исходной матрице данных, то, при этом, должны быть исключены из дальнейшего рассмотрения: строка(i), столбец(j) и все ячейки матрицы данных, которые лежат на левой и правой диагоналях, проходящих через точку (i,j). Выше мы исключили соответствующую строку и соответсвующий столбец, обнулив

соответсвующие ячейки в массивах *A* и *B*. Теперь, мы должны "*проекционно*" исключить те ячейки массива L, которые соответствуют диагональным исключениям в исходной матрице данных. Для этого используем соответствующие эквивалентные индексы, предварительно сохраненные в массивах *rAr* и *tAr.*

%}

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

%{

Тем самым мы выполнили все процедурные действия, связанные с выбором одной позиции (i,j) в исходной матрице данных для расположения ферзя.

%}

else % if freePos>0

%{

Если в рассматриваемой строке нет свободных позиций,то это означает, что мы достигли тупика, поэтому должны закрыть данную ветвь поиска и вернуться назад в блок-3, и повторить снова формирование ветви поиска. Перед этим, мы должны восстановить все необходимые массивы на основе резервных копий. Именно с этими массивами мы входили в событие 3.

Инкрементируем общий счетчик числа повторных вычислений, так как мы совершаем возврат назад для повторных вычислений.

%}

totSimCount= totSimCount+1;

% На основе сохраненных копий, восстановим значения необходимых массивов.

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

% Перейдем в Блок-3 для повторного счета.

activeEvent=3;

end % if freePos>0

end %while totPos < simBound2

else

%{

Если окажется, что количество повторений *simCount3* превышает граничное значение (*repeatBound3*) и, при этом *eventInd == 1*, то нужно вернуться на базовый уровень 1 и снова строить ветви поиска решений. Перед этим нужно восстановить все необходимые массивы, которые соответсвуют данной точке возврата

%}

if eventInd==1

% Инкрементируем общий счетчик числа повторных вичислений.

totSimCount= totSimCount+1;

% Восстановим массивы и передадим управление в Блок-1.

A=Ax;

B=Bx;

D1=D1x;

D2=D2x;

Q=Qx;

totPos=xTotPos;

% Обнулим значение счетчика *simCount3.*

simCount3=0;

% Перейдем в Блок-1.

activeEvent=1;

else

%{

Если в данном блоке было *simBound3* повторений, и в каждом случае, на каком-то шаге оказывалось, что среди оставшихся свободных строк, есть строка в которой не осталось ни одной свободной позиции, то это означает, что данная композиция отрицательная, и она не может быть комплектована. По этой причине работа программы должна быть прервана. Мы присваиваем переменной *compositionInd* значение ноль, чтобы

зафиксировать что рассматриваемая композиция отрицательная. Также присваиваем ноль переменной *processInd*, чтобы прервать дальнейшую работу программы.

%}

solutionInd=2;

processInd=0;

break

end

end % if simCount3 > simBound3

%{

При успешном завершиении вычислений в Блоке-3, количество ферзей, правильно

расположенных в матрице решения будет равно *eventBound3*.

Перейдем в Блок-4.

%}

if totPos >= eventBound2

activeEvent=4;

end

case 4

%{

Блок-4. Подготовка необходимых массивов для работы в блоке-5.

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

Мы переходим к событию 4 в трех случаях:

1)- Сразу, после завершения выполнения расчетов в Блке-3, т.е. в том случае,

если *eventInd* был равен 1 или 2

2)- Если значение *n <= nFix1*

3)- Если значение *nComp >= eventBound2*

В данном блоке выполняется подготовительная работы перед переходом в Блок-5. В определенной мере, работа алгоритма в данном блоке аналогична работе алгоритма в Блоке-3. Суть ее состоит в следующем:

Пусть количество оставшихся свободных строк в матрице решения равно *nRow*.

Сформируем массив *L*(1:nRow,1:nRow) и соберем в ней индексы свободных позиций всех оставшихся строк. Алгоритм формирования массива *L* аналогичен тому, что использовался в Блоке-2. Как и в Блоке-2 будем вести учет соответствия индексов массива *L* с соответствующими индексами исходной матрицы решения. Проекционный перевод решения из исходной матрицы в матрицу *L* меньшего размера, дает нам возможность на каждом шаге эффективно находить строку с минимальным значением числа свободных строк и значительно сократить объем вычислений. Но, не менее важное значение имеет тот факт, что на основе массива *L*, мы одновременно ведем учет состояния всех оставшихся свободных строк. Это позволяет контролировать все стрки и установить, наступила ли такая ситуация,когда в какой либо из оставшихся строк число свободных позиций равно нулю.В этом случае мы заранее исключаем рассматриваемую ветвь как тупиковую. Такой подход позволяет вести прогноз вперед и это важно. Мы прекращаем вычисления намного раньше того момента, когда "вдруг" будет установлено, что данная ветвь поиска тупиковая и ее нужно прервать.

%}

%{

Прямой переход от начала программы в событие-4, и последовательный

переход по цепочке Блок-2 --> Блок-3 --> Блок-4 различаются по форме

представления массивов *A* и *B*. Необходимо это учитывать.

%}

if eventInd==4

%{

Найдем исходные индексы оставшихся свободных строк в матрице решения

и сохраним их в массиве *A*

%}

A=find(A==0);

% Обозначим через nRow число свободных строк

nRow=length(A);

%{

Найдем исходные индексы оставшихся свободных столбцов в матрице решения и сохраним их в массиве *B*.

%}

B=find(B==0);

else

T=find(A>0);

A=A(T);

nRow=length(T);

T=find(B>0);

B=B(T);

end

% Создадим массив *L*(1:nRow,1:nRow) и заполним все ячейки единицей.

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

%{

Создадим рабочие массивы *rAr* и *tAr*. Сохраним в них индексы ячеек диагональных контрольных массивов, которые соответствуют свободным позициям в массиве *L.*

%}

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

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

%{

Создадим массивы для учета накопительного списка ограничений формируемых левой диагональю(*D1s*), правой диагональю(*D2s*) и проекций столбца(*Bs*)

%}

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

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

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

%{

На основе информации об оставшихся свободных строках и свободных столбцах, запишем zero в соответствующие свободные ячейки массива *L .* Формируем массивы *Cs, Ds, Bs,* а также массивы учета *rAr, tAr* . Для всех *nRow* строк и, соответственно, для оставшихся свободных позиций в этих строках сформируем накопительный список ограничений для

левой *D1s* и правой *D2s* диагональных проекций, а также для проекций столбца *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

% Вычислим сумму элементов каждой строки массива *L*

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

%{

Сортируем значения суммы в порядке возрастания числа свободных позиций в строке.

%}

[sumSort,rowRangInd]=sort(rowSum);

%{

Здесь, в массиве *rowRangInd* последовательно сохранены индексы строк с возрастающим числом свободных позиций в строке.

%}

if sumSort(1)>0

%{

Здесь *sumSort(1)* -минимальное число свободных позиций в списке всех строк массива *L*(nRow,nRow).

%}

%{

Если минимальное число свободных позиций больше нуля, то мы продолжаем построение ветви поиска решений

%}

%{

Создадим контрольный массив учета *E* размера *nRow x nRow*, в каждой ячейке которого сохраним суммарное значение соответствующих ограничений.

%}

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

%{

Вычислим и сохраним в массиве *E* совокупное значение накопительных массивов

ограничений

%}

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

%{

Далее, вместо массивов D1s,D2s и *Bs* мы будем использовать массив *E.*

Если нужно освободить память, то следует удалить массивы clear D1s D2s Bs, которые не будут использоваться.

%}

%clear D1s D2s Bs

%{

Прежде чем перейти к следующему событию, сохраним для повторного использования копии этих массивов.

%}

Az=A;

Bz=B;

Qz=Q;

Lz=L;

Ez=E;

zPos=totPos;

% Мы завершили подготовительную работу в Блоке-4. Далее, перейдем в Блок-5

activeEvent=5;

else % if sumSort(1)>0

%{

Если окажется, что среди оставшихся строк есть строка, в которой отсутствуют свободные позиции, то это означает:

a)Если *eventInd==4*, то рассматриваемая композиция изначально не может бать комплектована, так как в рассматриваемой композиции есть, по крайней мере одна свободная строка, в которой нет ни одной свободной позиции для расположения ферзя. (можно сказать, что данная композиция отрицательная с самого рождения).

b)Если *eventInd <3*, то это означает, что в предыдущем блоке продолжение формирования ветви поиска было неудачным, так как там не было проверки того, существует ли среди оставшихся свободных строк хотя бы одна строка, в которой все позиции закрыты и нет свободной позиции. В этом случае,нам нужно вернуться в Блок-2 и повторить формирование ветви поиска.

%}

if eventInd<3

%{

Если индекс событи равен 2 или 1, то возвращаемся назад в начало Блока-3. Для этого восстановим первоначальное состояние контрольних массивов, которые мы имели на выходе их Блока-2

%}

A=Ay;

B=By;

D1=D1y;

D2=D2y;

Q=Qy;

L=Ly;

rAr=rAr\_y;

tAr=tAr\_y;

totPos=yTotPos;

activeEvent=3;

% Инкрементируем общий счетчик числа повторных вичислений

totSimCount= totSimCount+1;

elseif eventInd > 3

%{

Если *eventInd==4*, то это означает, что размер композиции был таким, что мы сразу перешли на данный уровень, минуя блоки 1 и 2 и 3. А так как среди всех оставшихся строк есть хотя бы одна строка, в которой отсутствет свободная позиция, то данная композиция изначально такова, что не может быть комплектована. Поэтому выводим соответсвующее сообщение и прерываем работу программы.

Мы присваиваем переменной *compositionInd* значение -1, чтобы зафиксировать что рассматриваемая композиция изначально отрицательная, и гарантированно не имеет решения. Также, присваиваем ноль переменной *processInd*, чтобы прервать дальнейшую работу программы

%}

compositionInd=-1;

processInd=0;

break

end

end %if sumSort(1)>0

% После подготовительных работ в Блоке-4, мы совершаем переход в Блок-5.

case 5

%{

Блок-5. Завершающий этап решения задачи.

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

Мы находимся на последнем базовом уровне поиска решения. До конца решения осталось небольшое количество свободных строк. Если, начиння с данного уровня, в процессе решения задачи, ветвь поиска приведет в тупик, то мы вернемся обратно на данный базовый уровень.

На данном шаге мы должны выбрать только одну позицию в какой либо свободной строке. Количество возможностей такого выбора равно сумме свободных позиций во всех оставшихся свободных строк. Два вложенных цикла, которые используются здесь, служат лишь только для одной цели, чтобы выполнить только один шаг, т.е. выбрать индекс свободной строки на данном уровне, и индекс свободной позиции в этой строке. Весь дальнейший поиск решения для оставшихся свободных строк, производится только в пределах третьего вложенного цикла. Поэтому, на данном уровне выполняются следующие действия:

- выбирается строка с минимальным числом свободных позиций,

- выбирается свободная позиция в этой строке и там распологается ферзь.

После этого, в цикле выполняется следующая последовательность действий:

a)*Среди оставшихся свободных строк выбирается строка с минимальным числом свободных позиций,*

b)Среди свободных позиций в отобранной строке выбирается та позиция, которая наносит минимальный ущерб всем оставшимся свободным позициям.

Данный цикл продолжается до тех пор, пока не будет получено полное решение. Если на каком-то шаге данного цикла, ветвь поиска решения приведет в тупик, то цикл прерывается. На основе резервных копий восстанавливаются все массивы и переменные, соответствующие базовому уровню, и выполняется процедура возврата на базовый уровень. При этом, третий вложенный цикл повторяется снова, без каких либо

изменений параметров первого и второго вложенных циклов. Число таких повторных вычислений на уровне третьего вложенного цикла не должно превышать граничное значение *repeatBound*. Если число повторений превышает значение *repeatBound*, то в этом случае, после выполнения процедуры возврата на базовый уровень, параметры первых двух вложенных циклов изменяются как обычно. Использование такой модели работы трех вложенных циклов, не совсем очевидно на первый взгляд. Дело в том, что в тех случаях, корда имеется несколько строк с одинаковым минимальным значением суммарного числа свободных позиций, мы случайно отбираем индекс одного из двух таких строк (или случайный индекс одной их трех строк, если три строки имеют одинаковое минимальное значение). Аналогично, производится случайный отбор свободной позиции в строке, если две позиции в строке, наносят одинаковый минимальный ущерб всем оставшимся свободным позициям. (Здесь производится случайный выбор только из двух позиций, которые наносят одинаковый минимальный ущерб). Такой алгоритм мы используем лишь с одной целью, чтобы с максимальной эффективностью использовать "ресурсы задачи", которые остались к данному шагу. Чем ближе к концу решения задачи, тем меньше вероятность того, что в отобранной свободной строке окажется свободная позиция для расположения ферзя. Согласно правилу минимального риска, мы должны в первую очередь расположить ферзь в той свободной строке, где количество свободных позиций минимально. Именно это мы и делаем. Но в ситуациях, когда две строки, или две свободные позиции имеют одинаковые минимальные характеристики, мы производим случайный отбор. Когда третий вложенный цикл повторяется несколько раз без изменения параметров цикла, то это дает нам возможность использовать больше "ресурсных возможностей" задачи на данном уровне, так как на некоторых шагах формирования ветви поиска используется случайный выбор.

%}

% Обнулим счетчик учета числа повторных вычислений в Блоке-5

simCount5=0;

%{

Цикл *for iRow=1:nRow* служит для последовательного анализа оставшихся свободных строк, ранжированных в порядке возрастания суммарного числа свободных позиций в строке. Индексы соответсвующих строк сохранены в массиве *rowRangInd(1:nRow)*. Здесь *nRow* - число оставшихся свободных строк. Соответствующие расчеты были проведены нами в четвертом Блоке-4.

%}

for iRow=1:nRow % Первый (внешний) вложенный цикл.

% Выберем строку из ранжированного списка

selectRowInd=rowRangInd(iRow);

%{

*selectRowInd* является индексом строки в массиве *L*. Определим исходное значение индекса строки на шахматной доске, которая в массиве *L* соответствует индексу *selectRowInd.*

Значение *baseRowInd* далее нам будет нужно для повторных расчетов

%}

baseRowInd=A(selectRowInd);

% Скопируем из массива *L* строку с индексом selectRowInd в рабочий массив *T*

T=L(selectRowInd,:);

%{

Определим индексы свободных позиций в этой строке и сохраним результат в массиве *baseFreePosAr* (еще раз отметим, что нулевые позиции в массиве *L* соответсвуют свободныь позициям в исходной матрице решения)

%}

baseFreePosAr=find(T==0);

% Определим общее число свободных позиций(*nFreePos*) в этой строке

nFreePos=length(baseFreePosAr);

%{

Цикл *for jCol=1:nFreePos* служит для последовательного анализа свободных позиций в рассматриваемой строке

%}

for jCol=1:nFreePos % Вложенный цикл-2

% Присвоим i реальный индекс выбранной строки

i=baseRowInd;

%{

Выберем из массива *baseFreePosAr* индекс столбца, который записан в ячейке под номером *jCol*. Здесь *jPos* -это индекс столбца массива *L*

%}

jPos=baseFreePosAr(jCol);

jPosBase=jPos;

%{

Определим реальное значение индекса столбца(j), который соответствует

рассматриваемой шахматной доске

%}

j=B(jPos);

% Сохраним значение j в переменной *baseFreePos* для повторных расчетов

baseFreePos=j;

%{

Присвоим переменной *minRowInd* значение индекса строки массива *L*,

который имеет минимальное число свободных позиций в строке.

%}

minRowInd=selectRowInd;

% Обнулим значение счетчика числа повторных запусков третьего вложенного цикла.

repeatCount=0;

%{

Цикл *while totPos < n* является третьим вложенным циклом, где, на каждом шаге ведется поиск свободной позиции для расположения ферзя в какой-либо из оставшихся свободных строк.

%}

while totPos < n % Вложенный цикл-3

%{

Исходное значение индекса отобранной строки(i) и значение индекса столбца(j) для первого шага, мы определили выше (в Блоке-4), до входа в рассматриваемый цикл

Сохраним j-индекс позиции ферзя в стоке(i), массива решений *Q*(1:n).

%}

Q(i)=j;

% Инкрементируем счетчик учета количества установленных ферзей.

totPos=totPos+1;

% Проверим, если сформировано полное решение, то завершаем расчеты.

if totPos==n

solutionInd=1;

processInd=0;

break

end

%{

Мы воспользовались результатом, подготовленным в Блоке-4, и расположили ферзь в ячейку (i,j) матрицы решения. Тем самым, мы завершили очередной цикл определения позиции на шахматной доске для расположения ферзя. После этого, мы должны изменить соответствующие ячейки во всех контрольных массивах, учитывая индексы *(minRowInd,jPos)* массива *L*

%}

A(minRowInd)=0;

B(jPos)=0;

%{

Изменим соответствующие ячейки массива L, используя эквивалентные индексы, сохраненные в массивах *rAr* и *tAr*

%}

rx=n+j-i;

tx=j+i;

rxInd=find(rAr==rx);

L(rxInd)=1;

txInd=find(tAr==tx);

L(txInd)=1;

%{

Декрементируем значение накопительного контрольного массива *E*, так как в позицию (i,j) мы расположили ферзь

%}

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

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

% Запишем 1 во все активные ячейки столбца *jPos* массива *L*

A1=find(A>0);

L(A1,jPos)=1;

%{

На данном шаге, внутри цикла *while totPos < n* мы выполнили следующие действия:

- установили ферзь в ячейку (i,j), воспользовавшись ранее подготовленной

информацией

- выполнили необходимые процедурные действия с контрольными массивами,

после расположения ферзя в ячейку (i,j).

%}

%{

Выбор свободной строки и свободной позиции в строке

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

Теперь, среди оставшихся свободных строк, найдем строку с минимальным числом свободных позиций, и из этих позиций выберем ту, которая, в случае закрытия позиции, нанесет минимальный ущерб всем оставшимся свободным позициям в осташихся строках. Для этого выполним следующие действия:

%}

%1.Определим сумму свободных позиций в каждой из оставшихся свободных строк

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

%{

2. Ранжируем полученные значения сумм в порядке возрастания значений.

3. Сохраним в массиве *freePosAr* ранжированные значения сумм, а в массиве

*rowIndAr* - индексы соответствующих строк.

%}

[freePosAr,rowIndAr]=sort(rowSum);

%{

Так как на данном этапе мы одновременно ведем учет состояния всех оставшихся свободных строк, то это дает нам возможность установить, наступила ли такая ситуация, что в какой либо из оставшихся строк число свободных позиций равно нулю. В этом случае мы рассматриваем формируемую ветвь поиска как тупиковую и возвращаемся на начало цикла. Такой подход позволяет вести прогноз вперед - мы прекращаем вычиления до того, как будет установлено на очередном шаге, что в рассматриваемой строке не осталось ни одной свободной позиции.

%}

%{

Здесь контрольная точка для формируемой ветви поиска. Если, в каждой из оставшихся свободных строк, есть хотя бы одна свободная позиция, то продолжается формирование ветви поиска.

%}

if freePosAr(1)>0

%{

Может быть такая ситуация, что в ранжированном списке, первые два элемента списка, или первые три элемента списка, имеют одинаковое минимальное значение. В этом случае мы случайно отбираем индекс одной из двух строк с одинаковым минимальным значением (либо, индекс одной из трех строк, если таковых три).

%}

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);

%{

Определим число свободных позиций в выбранной строке и сохраним индексы этих строк в массиве *freePosAr*

%}

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

% Определим количество свободных позиций (*nFreePos*)

nFreePos=length(freePosAr);

%{

Выберем среди этих позиций ту, которая закрывает минимальное число свободных позиций в оставшихся строках. Для этого воспользуемся массивом *E*. Если две строки имеют одинаковое минимальное число свободных позиций, то случайно выбираем индекс *jPos* одной из них. *Здесь мы вносим в алгоритм элемент "здоровой" случайности во всех тех случаях, когда две строки имеют одинаковое число свободных позиций.*

%}

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

%{

На основе массива исходных индексов *A*, восстановим реальный индекс i данной строки

%}

i=A(minRowInd);

%{

На основе массива исходных индексов *B*, восстановим реальный индекс j данного столбца

%}

j=B(jPos);

else % if freePosAr(1)>0

%{

Если окажется, что в строке нет свободных позиций, то это значит, что ветвь поиска привела в тупик. В этом случае, закроем ветвь поиска и инкрементируем значение счетчиков: *repeCount, simCount5, totSimCount*

%}

repeatCount=repeatCount+1;

simCount5=simCount5+1;

totSimCount=totSimCount+1;

%{

Восстановим значения всех контрольных массивов (по состоянию на начало выполнения цикла *while totPos < n*) и передадим управление на начало цикла. Если количество повторного использования цикла ( *while totPos < n* ) не превышает пороговое значение *repeatBound*, то управление передается на начало цикла *while totPos < n*, без изменения параметров двух внешних циклов

%}

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

i=baseRowInd;

j=baseFreePos;

minRowInd=selectRowInd;

jPos=jPosBase;

if repeatCount>repeatBound

%{

Если количество повторного использования цикла ( *while totPos < n* ) превышает порогового значени *repeatBound*, то управление передается во внешний цикл *while jCol<=colPos* Выше, мы восстанавили соответствующие параметры для перехода

%}

repeatCount=0;

i=baseRowInd;

break

% Выход из цикла *while totPos <n*

% Переход на продолжение цикла while jCol<=colPos

end

end % freePosAr(1)>0

end %while totPos < n

if processInd==0

break

%Выход из цикла *while jCol<=colPos*

end

end %while jCol<=colPos

if processInd==0

break

% Выход из циклаfor iRow=1:nRow

end

%{

Здесь, внутри цикла *for iRow=1:nRow*, единственное место, где мы проводим контроль значения totSimCount. Если значение totSimCount превышает значение *totRepeatBound*, выводится сообщение о том, что вероятность того, что данная композиция может быть комплектована до полного решения меньше *0.0001 .*

%}

if totSimCount > totSimBound

solutionInd=3;

break

%{

Пусть totSimCount <= totSimBound. Тогда, если, после выхода из цикла *while jCol<=colPos* окажется, что число повторных испытаний на данном уровне(*simCount5*) превышает допустиме границы(*simBound5*), то в случае, если *eventInd<3*, мы передаем управление в Блок-2. Если eventInd = 4, то работа программы продолжается, с учетом принятых ограничений

%}

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

%{

Если solutionInd == 3, то мы увеличиваем счетчик полного цикла комплектования.

falseNegSimCount.

Если значение счетчика не превышает falseNegSimBound, то мы возвращаем управление

в начале пересчета в соответствии со значением eventInd

%}

if solutionInd==3

if falseNegSimCount < falseNegSimBound

falseNegSimCount=falseNegSimCount+1;

switch eventInd

case 1

% Восстановим массивы и передадим управление событию 1

A=Ax;

B=Bx;

D1=D1x;

D2=D2x;

Q=Qx;

totPos=xTotPos;

activeEventInd=1;

case 2

% Восстановим массивы и передадим управление событию 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

% Восстановим массивы и передадим управление событию 5

A=Az;

B=Bz;

Q=Qz;

L=Lz;

E=Ez;

totPos=zPos;

activeEventInd=5;

end

%Обнулим счетчки учета соответствующих событий

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));

disp(Q(1:nDisp));

%{

Сохраним результат комплектации в файле *nQueens\_Test\_Solution.mat* . Если в результате решения не удалось комплектовать композицию до полного решения, то, в соответствующих ячейках массива Q сохраняются нулевые значения.

Имя файла nQueens\_Test\_Solution.mat приведено в качестве примера. Очевидно, что можно использовать любое другое имя.

%}

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

if solutionInd == 1

save(outputFileName,'Q');

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

disp(iInfo);

end