八皇后问题

如果你去百度百科八皇后这个问题，你会发现人家也是历史上有头有脸的一个问题，最后一句“计算机发明后就有一万种方式解决这个问题”读起来也让程序猿们很快活。闲话少说，开始阐述我的思路：

最无脑的解法一定是八个for遍历，浪费了太多的计算资源在各种无用功上面，我们稍微构思一下：

首先如何决定下一个皇后能不能放这里可以有两种思路，第一种是尝试维护一个8\*8的二维矩阵，每次找到一个空位放下一个皇后就把对应行列对角线上的棋格做个标记，如果某行找不到可放皇后的格子就把上一个皇后拿走并把对应行列对角线的标记取消掉；第二种方法直接放弃构造矩阵来假装棋盘，我们把问题更加抽象化，八个皇后能放下一定是一行放一个，我们只需一个数组记录每个皇后的列数（默认第N个放第N行），那么问题就被抽象成了数组的第N个数和前N-1个数不存在几个和差关系即可（比如差不为零代表不在同一列）。

接着想想问题中存在着大量的循环怎么解决比较高效，我们知道递归和迭代一定程度上是可以很容易做到互相转化实现同样的思路的。递归是重复调用函数自身实现循环，迭代是函数内某段代码实现循环，使用递归的话我们应该要有一个能在第N行找到某一列的格子可以放皇后的函数，能找到把参数+1去调用自己去找下一行皇后能放的格子，找不到就算了。如果想用迭代，前面我们说过递归迭代是可以转化的，这种在函数最后调用自己的递归更是极易转化，我们按着迭代的套路在for循环的里按照刚刚递归的思路加几个判断判别循环是continue、break还是返回前一层循环即可。最后还有一种思路，准确来说还是和递归脱离不了关系，学习递归的时候我们我们知道，递归可以看做底层帮你维护的一个堆栈不断地push、pop，知道它的本质我们也可以通过手动维护一个堆栈来模拟这个递归调用的过程，只要构造两个函数backward(往后回溯)、refresh(向前刷新)来模拟堆栈进出即可。

最后我们来分析四个方法（矩阵维护法、递归法、迭代法、手动堆栈法）表现和改进，很明显在代码量上递归会是最短的，而需要运行的空间来看手动堆栈也会比较必要更大的运行内存（如果用VS运行手动堆栈的代码，很有可能会提示你stack溢出，那么你需要修改一下VS的配置给你的程序分配更大的内存）。八皇后问题有很多小细节可以改进（具体实现大家自己来，为了方便我就说一些我想到的点）：很明显棋盘是对称的，如果你得出了一个解法那么一定有行对称列对称对角线对称的另外三种对称的摆法，这样就可以减少一些计算量。

头脑风暴过后，结合代码和注释讲解具体实现过程：

1.矩阵维护法

这是第一个出现在我头脑中的方法，很桑心居然不是递归，看来脑子还不够抽象。上代码：

//八皇后维护矩阵法

#include<iostream>

using namespace std;

int cheese\_table[8][8];

int queen[8];//记录五个皇后的列数

int lastqueen=-1;

int solution=0;

int search\_line(int i,int j){//搜寻这一行有没可放的位置

for(;j<8;j++)

if(cheese\_table[i][j]==0)

return j;

return -1;

}

void set\_queen(int i,int j){//在可放的位置上放上皇后记录下来并对棋盘进行操作

cheese\_table[i][j]=-1;

queen[i]=j;

for(int temp=0;temp<8;temp++)//列操作

if(cheese\_table[temp][j]!=-1)

cheese\_table[temp][j]++;

for(int temp=0;temp<8;temp++)//行操作

if(cheese\_table[i][temp]!=-1)

cheese\_table[i][temp]++;

int tempj=j+1;

for(int tempi=i+1;tempi<8&&tempj<8;tempi++)//东南对角线操作

cheese\_table[tempi][tempj++]++;

tempj=j-1;

for(int tempi=i+1;tempi<8&&tempj>=0;tempi++)//东北对角线操作

cheese\_table[tempi][tempj--]++;

tempj=j+1;

for(int tempi=i-1;tempi>=0&&tempj<8;tempi--)//西南对角线操作

cheese\_table[tempi][tempj++]++;

tempj=j-1;

for(int tempi=i-1;tempi>=0&&tempj>=0;tempi--)//西北对角线操作

cheese\_table[tempi][tempj--]++;

return;

}

void uptake\_queen(int i){

int j=queen[i];

for(int temp=0;temp<8;temp++)//列操作

if(cheese\_table[temp][j]!=-1)

cheese\_table[temp][j]--;

for(int temp=0;temp<8;temp++)//行操作

if(cheese\_table[i][temp]!=-1)

cheese\_table[i][temp]--;

int tempj=j+1;

for(int tempi=i+1;tempi<8&&tempj<8;tempi++)//东南对角线操作

cheese\_table[tempi][tempj++]--;

tempj=j-1;

for(int tempi=i+1;tempi<8&&tempj>=0;tempi++)//东北对角线操作

cheese\_table[tempi][tempj--]--;

tempj=j+1;

for(int tempi=i-1;tempi>=0&&tempj<8;tempi--)//西南对角线操作

cheese\_table[tempi][tempj++]--;

tempj=j-1;

for(int tempi=i-1;tempi>=0&&tempj>=0;tempi--)//西北对角线操作

cheese\_table[tempi][tempj--]--;

cheese\_table[i][j]=0;

return;

}

int main(){

for(int i=0;i<8;i++)

for(int j=0;j<8;j++)

cheese\_table[i][j]=0;

//初始化棋盘

for(int i=0;;i++){//一行一行操作

int j=search\_line(i,lastqueen+1);

if(j==-1){//没有放皇后的位置了，回头

if(i==0)break;//真正结束位置

uptake\_queen(i-1);

lastqueen=queen[i-1];

//把上一行的queen的位置记录下来，便于回头的时候从这个位置之后寻找可放位置

i-=2;

}

else{

//把棋盘对应位置放上皇后，对这个皇后会影响的棋格进行操作

lastqueen=-1;

set\_queen(i,j);

if(i==7){

solution++;

uptake\_queen(7);

lastqueen=j;

i--;

}

}

}

cout<<solution<<endl;

return 0;

}

稍微讲解一下，cheese\_table为8\*8的棋盘，queen数组记录八个皇后各自的列数（前面说过，第N个皇后默认放在第N行，所以行数是隐式记录的），lastqueen记录着最后放置的那个皇后的列数（回溯时候很重要，保证回溯到上一行操作时候不会踏进同一个坑即不会再把皇后放到刚刚放过的地方），solution记录八皇后有几种放的方法。Search\_line（i，j）函数将会搜寻第i行从j列开始还有没可以放的格子，set\_queen（i，j）就是在可放皇后的(I,j)格子放下皇后，并且在棋盘上对放下的这个皇后的行列和主副对角线的格子进行标记，标记的方法是代表这些格子的数+1（这是本解法很关键的一点，并不是简简单单的对这些不可放置点从一个状态比如0置为1代表不可放置了，而是每次把某个皇后对应影响的这些格子的数都增加1，这么做极大的好处就是你回溯的时候只要逆着过去对拿走的皇后本会影响的格子减1即可，而不需要判断这些格子是否还会被其他在棋盘上的的皇后影响从而决定维持不可放的状态还是变为可放的状态，极大的减少了维护棋盘时候大量调用判断函数的时间，而只要简单的加减即可）。Uptate\_queen（i）函数就是拿起第i行的皇后，即本解法的回溯部分，对应set的过程你这做即可。最后看看主函数，初始化不说了，for循环中大致过程就是对每一行search出皇后可放位置，找到可放格子就放下皇后，如果八个皇后都放完了记一次数，并且在最后一行寻找是否有其他放皇后的位置，没有的话往前一行回溯；刚刚在某一行search不到放皇后的格子就只能回溯上一行。如果发现这一行就是第0行没有上一行了还要回溯，证明我们算法结束了，退出循环。这个for循环大概是假的for循环，没有限定i的大小，依靠的其实是想要回溯之前看看还能不能回溯来跳出。

2.递归法

//八皇后递归解法

#include<iostream>

using namespace std;

int queen[9]={-1,-1,-1,-1,-1,-1,-1,-1,-1};

int count=0;

bool available(int pointi,int pointj){//判断某个皇后是否与已有皇后冲突

for(int i=1;i<pointi;i++){

if(pointj==queen[i])return false;//同一列拒绝

if((pointi-i)==(pointj-queen[i]))return false;//同一主对角线拒绝

if((pointi-i)+(pointj-queen[i])==0)return false;//同一副对角线拒绝

}

return true;

}

void findSpace(int queenNumber){//在第queenNumber行找能放皇后的位置

for(int i=1;i<9;i++){//从1~8遍历这一行的八个空位

if(available(queenNumber,i)){

//如果可以放这个位置就记录下第queenNumber个皇后的位置

queen[queenNumber]=i;

if(queenNumber==8){//如果八个皇后都放满了统计一下

count++;

return;

}

int nextNumber=queenNumber+1;//还有皇后没放递归放下一个皇后

findSpace(nextNumber);

}

}

queen[--queenNumber]=-1;//如果这一行没有可放的位置说明上一行皇后放的位置不行，要为上一个皇后寻找新的可放位置

return;

}

int main(){

findSpace(1);//从（1，1）开始递归好理解

cout<<count<<endl;

return 0;

}

递归法不多说了，八皇后的最标准解法，我的注释也很详细，唯一我自己加的一个小技巧是把一开始设为1,1而不是0,0，毕竟人类都是习惯从1开始，当然我现在有点后悔了，写文章的时候再看代码感觉很脑残，毕竟本文对象程序猿好像已经习惯从下标0开始计数了哈。所以一开始数组设了9个元素，main函数调用递归函数从1,1开始都当是我的自作多情，大家开心就好。

3.迭代法

//八皇后迭代解法

#include<iostream>

using namespace std;

int count=0;

int queen[8]={-1,-1,-1,-1,-1,-1,-1,-1};

bool available(int pointi,int pointj){//判断某个皇后是否与已有皇后冲突

for(int i=0;i<pointi;i++){

if(pointi==i)return false;//同一行拒绝

if(pointj==queen[i])return false;//同一列拒绝

if((pointi-i)==(pointj-queen[i]))return false;//同一主对角线拒绝

if((pointi-i)+(pointj-queen[i])==0)return false;//同一副对角线拒绝

}

return true;

}

int main(){

int j=0;

for(int i=0;i<8;i++){//对于每一行

if(i==-1)break;//这才是真正退出循环的出口

for(;j<8;j++){

if(available(i,j)){

queen[i]=j;

if(i==7){

count++;

if(j==7){//如果最后一行最后一格试完就往前回溯

j=queen[--i];

j++;

queen[i]=-1;

i--;

break;

}

else

continue;

}

j=0;

break;

}

else

if(i==7&&j==7){

j=queen[--i];

j++;

queen[i]=-1;

i--;

break;

}

}

if(j==8){

j=queen[--i];

j++;

queen[i]=-1;

i--;

}

}

cout<<count<<endl;

return 0;

}

由于迭代法是我用迭代的套路来完成递归的思路的一个解法，所以直接看有点抽象，但是理解上边的递归以后再来看你就会发现：嗯，这里我好像见过。

4.手动维护堆栈法

//八皇后手动维护堆栈解法

#include<iostream>

using namespace std;

int QueenNumber=0;

int solutionCount=0;

int stopflag=0;

struct point{

int pointi;

int pointj;

}queenPoint[8];

bool available(int pointi,int pointj);

void backward();

void refresh(int pointi,int pointj);

int main(){

for(int i=0;i<8;i++)

queenPoint[i].pointi=queenPoint[i].pointj=-1;

//从（0，0）格子开始递归

refresh(0,0);

cout<<solutionCount<<endl;

return 0;

}

bool available(int pointi,int pointj){

for(int i=0;i<QueenNumber;i++){

if(pointi==queenPoint[i].pointi)return false;//同一行拒绝

if(pointj==queenPoint[i].pointj)return false;//同一列拒绝

if((pointi-queenPoint[i].pointi)==(pointj-queenPoint[i].pointj))return false;

//同一主对角线拒绝

if((pointi-queenPoint[i].pointi)+(pointj-queenPoint[i].pointj)==0)return false;

//同一副对角线拒绝

}

//都没问题返回可以

return true;

}

void backward(){

QueenNumber--;

int tempi=queenPoint[QueenNumber].pointi;

int tempj=queenPoint[QueenNumber].pointj;

queenPoint[QueenNumber].pointi=queenPoint[QueenNumber].pointj=-1;

if(QueenNumber<0)stopflag=1;;

refresh(tempi,++tempj);

}

void refresh(int pointi,int pointj){

//先是两种特殊情况的判断?

if(stopflag==1)return;

if(pointj==8)

backward();

//某一格可放就更新信息往下行递归

if(available(pointi,pointj)){

queenPoint[QueenNumber].pointi=pointi;

queenPoint[QueenNumber].pointj=pointj;

QueenNumber++;

//如果八个皇后都放完就计数回溯

if(QueenNumber==8){

solutionCount++;

backward();

}

//否则往下递归

else

refresh(++pointi,0);

}

//某一格不可放就往下一个格子递归

else{

//如果某一行都不行就回溯

if(pointj==7)

backward();

else

refresh(pointi,++pointj);

}

}

手动维护堆栈对于我们理解递归本质是很有好处的，当然这段代码飞出了一个大bug——stack overflow，大体思路上没啥问题啊bug老飞啊飞就很烦，时间紧迫最后我就简单粗暴把VS的运行内存调大了许多解决的，希望有志之士帮忙看看本质的解决方案是什么。

具体说说代码（这几个代码不是连贯写下来的，算是好几个晚上有空的时候码一码，所以我总感觉自己看的有点不连贯不知为啥），queennumber记录已经在棋盘上放的皇后数量，solutionnumber不说了结果数量，stopflag就是上面几个方法用来跳出循环的这里直接弄了一个flag，定义了一个结构point记录八个皇后的行列信息（大概就是这里和前面很反差吧），三个函数available用来判断某个点能不能放皇后，refresh用来往前推进的，函数中前三行后面说，第四行开始是主要工作，在这一行内调用available判断某个格子能不能放皇后，可以的话记录信息，并且判断是否把八个皇后都放完了，是的话回溯，否则从下一行的第一个格子开始递归，如果available判断某格子不能的话跳到本行下一个格子递归，如果某一行发现都无法放皇后，调用backward回溯。Backward函数主要做的就是，取消最近放的那个皇后的一切信息，在回溯过程中如果发现再回溯得回溯第-1行了（即第一歌皇后放在第一行最后一个格子的所有情况都尝试过了），把stopflag变为1,backward最后调用refresh对上一个皇后的下一个可能位子递归判断。所有回过头看refresh前三行，首先如果发现stopflag出现了，那么一层一层退回去结束这个逻辑上的循环，第二个if用来判断某一行是否找不到皇后的放置点了，是的话这个解法不行，上一个皇后得换地方。