/\*\*

\* knight.c 深度优先遍历 - 骑士周游问题|马踏棋算法

\*

\* @author Darbuly

\* @copy 2018-2019 DBL

\*

\*/

#include <stdio.h>

#include <time.h>

#define X 8

#define Y 8

int chess[X][Y];

int nextxy(int \*x,int \*y,int count)

{

switch(count)

{

case 0:

if(\*x+2<=X-1 && \*y-1>=0 && chess[\*x+2][\*y-1]==0)

{

\*x+=2;

\*y-=1;

return 1;

}

break;

case 1:

if(\*x+2<=X-1 && \*y+1<=Y-1 && chess[\*x+2][\*y+1]==0)

{

\*x+=2;

\*y+=1;

return 1;

}

break;

case 2:

if(\*x+1<=X-1 && \*y-2>=0 && chess[\*x+1][\*y-2]==0)

{

\*x+=1;

\*y-=2;

return 1;

}

break;

case 3:

if(\*x+1<=X-1 && \*y+2<=Y-1 && chess[\*x+1][\*y+2]==0)

{

\*x+=1;

\*y+=2;

return 1;

}

break;

case 4:

if(\*x-2>=0 && \*y-1>=0 && chess[\*x-2][\*y-1]==0)

{

\*x-=2;

\*y-=1;

return 1;

}

break;

case 5:

if(\*x-2>=0 && \*y+1<=Y-1 && chess[\*x-2][\*y+1]==0)

{

\*x-=2;

\*y+=1;

return 1;

}

break;

case 6:

if(\*x-1>=0 && \*y-2>=0 && chess[\*x-1][\*y-2]==0)

{

\*x-=1;

\*y-=2;

return 1;

}

break;

case 7:

if(\*x-1>=0 && \*y+2<=Y-1 && chess[\*x-1][\*y+2]==0)

{

\*x-=1;

\*y+=2;

return 1;

}

break;

default:

break;

}

return 0;

}

void print()

{

int i,j;

for(i=0;i<X;i++)

{

for(j=0;j<Y;j++)

{

printf("%2d\t",chess[i][j]);

}

printf("\n");

}

printf("\n");

}

int TravelChessBoard(int x,int y,int tag)

{

int x1=x,y1=y,flag=0,count=0;

chess[x][y]=tag;

if(X\*Y == tag)

{

print();

return 1;

}

flag = nextxy(&x1,&y1,count);

while(0==flag && count<7)

{

count++;

flag=nextxy(&x1,&y1,count);

}

while(flag)

{

if(TravelChessBoard(x1,y1,tag+1))

{

return 1;

}

x1=x;

y1=y;

count++;

flag = nextxy(&x1,&y1,count);

while(0==flag && count <7)

{

count++;

flag = nextxy(&x1,&y1,count);

}

}

if(0==flag)

{

chess[x][y]=0;//回溯，对应节点标记清零

}

return 0;

}

int main()

{

int i,j;

clock\_t start,stop;

start = clock();

for(i=0;i<X;i++)

{

for(j=0;j<Y;j++)

{

chess[i][j]=0;

}

}

if(!TravelChessBoard(2,0,1))

{

printf("sorry!Coun'ld founded!\n");

}

stop = clock();

printf("\nThis time had run for :%fsec\n\n",(double)(stop-start)/CLOCKS\_PER\_SEC);

return 0;

}

代码分析

时间复杂度：O(m\*n)

这个方法时间复杂度太高了，最多要搜8^64次（实际存在边界和已经访问的标记，虽然不会这么多次，也是很大），太过盲目

思维总结：

1.采用递归+回溯的方式，一个棋盘可以看成有64层深度的一棵树，每一个节点最多有8个子节点，采用深搜可以很方便的解决这个问题

优化思路：加入贪心算法

主要的思想没有变，还是用深度优先搜索，只是在选下一个结点的时候做了贪心算法优化，其思路如下：

从起始点开始，根据“马”的走法，它的下一步的可选择数是有0—8个的。

我们知道，当下一步的可选择数为0的时候，进行回溯。

当下一步的可选择数有1个的时候，我们直接取那一步就行了。

但是如果下一步的可选择数有多个的时候呢？

但其实我们怎么选下一步，对搜索的效率影响是非常大的！

贪心算法解释：

贪心算法（又称贪婪算法），是指在对问题求解时，总是做出在当前看来是最好的选择。

也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

它只考虑局部的最优，从而让总体也最优。

就我们这个马踏棋盘来说，我们每走一步都取最优的那个选择，从而让整体的算法也最优。

但是我们选择下一步的时候（假设有a、b、c、d四个点可以选择），怎样选才算是最优呢？

答案是：哪一个点的下一步少，就选哪一个。

我们选择a、b、c、d之中的某一个点作为下一步，选哪个比较好，就看哪个点的后续下一步比较少。例如：马走到a点后的下一步有3个选择；而b的下一步有2个；c有1个，d有2个。那么我们的最优选择是：c点！

为什么要这样选呢？网上的解释是：“选择最难走的路，才能走的远”呜。。。好像太抽象了。

我的理解是：有些选择的后续下一步很少，例如c点，如果不先遍历它的话以后可能会很难遍历到它。

甚至极端一点的情况是，如果现在不遍历它，以后都遍历不到了。遍历不成功的时候只能回溯，一直回溯到此刻的点，然后选了c点以后才能完成，这就浪费了大量的时间。

https://blog.csdn.net/Bone\_ACE/article/details/41579957