**摘要**

马的周游问题是个np问题，本文对特定的m的国际象棋棋盘的马的周游问题（m,n均为偶数且|m-n|<），分别给出了其回溯和分治的解法。其中回溯法除了Warnsdorff's rule，还给出了两个剪枝规则。在分治求解中，基于特殊的低纬棋盘结构，采用了勾连法进行棋盘的合并求解高维棋盘的马的Hamilton回路，使得求解高维的棋盘时，算法复杂度达到O（n）。

**关键字： 马的周游；hamilton 回溯；分治**

**目 录**

[一、马的Hamilton周游问题求解——基于递归回溯 1](#_Toc515867564)

[1.1回溯法的简介 1](#_Toc515867565)

[1.2马的Hamilton周游问题——递归回溯解法 4](#_Toc515867566)

[1.3回溯法求解实现结果 11](#_Toc515867567)

[二、马的Hamilton周游问题——分治解决 12](#_Toc515867568)

[2.1分治算法的简介 12](#_Toc515867569)

[2.2马的Hamilton周游问题的分治解法 13](#_Toc515867570)

[2.3算法实现——基于c++ 16](#_Toc515867571)

[2.4 16棋盘下马的Hamilton回路程序运行求解结果： 27](#_Toc515867572)

[参考文献 28](#_Toc515867573)

# 一、马的Hamilton周游问题求解——基于递归回溯

## 1.1回溯法的简介

回溯法有“通用的解题方法”之称。用它可以系统地搜索一个问题的所有解或任一解。回溯法是一个既带有系统性又带有跳跃性的搜索算法。它在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时，先判断该结点是否包含问题的解。如果肯定不好含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯。否则，进入该子树，继续按深度优先策略搜索。回溯法求问题的所有解时，要回溯到根，且根结点的所有子树都已被搜索遍才结束。回溯法求问题的一个解时，只要搜索到问题的一个解就可以结束。这种以深度优先方式系统搜索问题解的算法称为回溯法，它适用于解组合数较大的问题。

详细的描述则为：回溯法按深度优先策略搜索问题的解空间树。首先从根节点出发搜索解空间树，当算法搜索至解空间树的某一节点时，先利用剪枝函数判断该节点是否可行（即能得到问题的解）。如果不可行，则跳过对该节点为根的子树的搜索，逐层向其祖先节点回溯；否则，进入该子树，继续按深度优先策略搜索。

回溯法的基本行为是搜索，搜索过程使用剪枝函数来为了避免无效的搜索。剪枝函数包括两类：

（1）使用约束函数，剪去不满足约束条件的路径；

（2）使用限界函数，剪去不能得到最优解的路径。

问题的关键在于如何定义问题的解空间，转化成树（即解空间树）。解空间树分为两种：子集树和排列树。两种在算法结构和思路上大体相同。

**回溯法通常解题步骤：**

1. 针对所给问题，定义问题的解空间；
2. 确定易于搜索的解空间结构；
3. 以深度优先方式搜索解空间，并在搜索过程用剪枝函数避免无效搜索

1.1.1递归回溯

回溯法

void backtrack(int t)

{

if(t>n) output(x);

else

for(int i=f(n,t);i<g(n,t);i++){

x[t]=h(i);

if(constraint(t)&&bound(t))backtrack(t+1);

}}

其中形式参数t 表示递归深度，即当前扩展结点在解空间树中的深度。N用来控制递归深度。当t>n时，算法已搜索至叶节点，此时，由output（x）记录或输出得到的可行解x。算法backtrack的for循环中f（n，t）和g（n，t）分别表示在当前扩展点未搜索的子树的起始编号和终止编号，h（i）表示在当前扩展结点处x[t]的第i个可选值，constraint（t）和bound（t）是当前扩展结点处的约束函数和限界函数。constraint（1）返回的值为true时，在当前扩展结点处x[1:t]取值满足问题的约束条件，否则不满足问题的约束条件，可剪去相应的子树。bound（t）返回值为true时，在当前扩展结点处x[1:t]取值未使目标函数越界，还需由backtrack（t+1）对其相应的子树进一步搜索。否则，当前扩展节点处x[1:t]的取值使目标函数越界，可剪去相应的子树，执行算法for循环后，已搜索遍当前扩展结点的所有未被搜索过的子树。backtrack（1）执行完毕，返回t-1层继续执行，对还没测试过的x[t-1]的值继续搜索。当t=1时，若已经测试完x[1]的所有可选值，外层调用就全部结束。显然，这一搜索过程按深度优先方式进行。调用一次backtrack（1）即可完成整个回溯搜索过程。

迭代回溯：

采用树的非深度递归优先遍历算法，可将回溯法表示为一个非递归迭代过程。

void iterativeBacktrack()

{

Int t=1;

while (t>0){

if(f(n,t)<=g(n,t))

for(int i=f(n,t);i<=g(n,t);i++){

x[t]=h(i);

if(constraint(t)&&bound(t)){

if(solution(t))output（x）;

else t++;}

}

else t--;

}

}

上述迭代回溯算法中，solution（t）判断在当前扩展结点处是否已得到问题的可行解。它返回的值为true时，在当前扩展结点处x[1:t]是问题的可行解。此时，由output（x）记录或输出得到的可行解，它返回的值为false时，在当前扩展结点处x[1:t]只是问题的部分解，还需向纵深方向继续搜索，算法f（n,t）和g（n,t）分别表示当前扩展结点处未搜索过的子树的起始编号和终止编号。H（i）表示在当前扩展结点处x[t]的第i个可选值。constrain（t）和bound（1）是当前扩展结点处的约束函数和限界函数。constrain(t)返回值为true时，在当前扩展结点处x[1:t]取值满足问题的约束条件，否则不满足问题的约束条件，可剪去相应的子树。bound（t）返回值为true时，在当前扩展结点处x[1:t]取值未使目标函数越界，还需对其相应的子树进一步搜索。否则，当前扩展结点处x[1:t]取值未使目标函数越界，可剪去相应的子树。算法while循环结束后，完成整个回溯搜索过程。

用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径，如果解空间树从根结点到叶节点的最长路径的长度为h（n），则回溯法所需的计算空间通常为O（h（n））。而显示地存储整个解空间则需要O（）或O（h（n）！）内存空间。

子集树和排列树

当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树为子集树。这类问题通常有个叶结点，其结点个数为-1.遍历自己数的任何算法均需的计算时间。

当所给的问题是确定n个元素满足某种性质的排列的排列时，相应的解空间树称为排列树。排列树通常有n！个叶节点。因此遍历排列树需要的计算时间。

## 1.2马的Hamilton周游问题——递归回溯解法

1.2.1 算法设计

根据马走日的特点，构建一维数组step[8]描述马的走法，从step[0]-step[7]分别表示马从当前位置的左上角位置逆时针方向可能走的8个步，如图1.1所示。由于马的行走位于棋盘中，因此可将这隐约束条件化为显约束条件。如果将nn格棋盘看做二维方阵，那么边可以构建一个二维数组的下标描述马的位置，对应元素值描述马的步数。

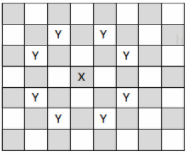


图1.1

用回溯法解决马的周游问题时，如果没有进行剪枝的话，那么算法的执行效率会很低（遍历完8，马的步数是2^4\*3^8\*4^20\*6^16\*8^20=25\*10^40）。因此根据问题的特点构建一下三点剪枝规则：

1、使用Warnsdorff's rule。在当前位置（Now）考虑下一个位置（Next）的时候，优先选择下一个的位置（Next）走法最少的那个。作为当前位置（Now）的下一位置（Next）。

2、在进行了第一点的剪枝后，如果可以优先选择的下一个位置不止一个，则优先选择离中心位置较远的位置作为下一步（即靠近棋盘边缘的位置）。

3、第三点的剪枝，每次都从棋盘的中间点开始出发，然后求出一条合法路径后再平移映射回待求路径。

最后，找到这个这个以中点为起点的哈密顿回路后，根据设定起点在这个回路中的序号，映射回以这个位置为起点的马周游路线即可。

1.2.2算法实现

#include <iostream>

#include <stdlib.h>

#include <iomanip>

#include <queue>

using namespace std;

typedef struct

{

int x;

int y;

} Step;

Step step[8] = { {-2, -1}, {-1, -2}, { 1, -2}, { 2, -1}, { 2, 1}, { 1, 2}, {-1, 2}, {-2,1} };

typedef struct NextPos

{

int nextPosSteps; //表示下一位置有多少种走法；走法少的优先考虑

int nextPosDirection; //下一位置相对于当前位置的方位

int nextPosToMidLength; //表示当前位置距中间点距离；距离中间点远的优先考虑

//

bool operator < (const NextPos &a) const

{

return nextPosSteps > a.nextPosSteps && nextPosToMidLength < a.nextPosToMidLength;

}

};

int board[100][100];

int M,N; //棋盘大小

//检测这个位置是否可以走

bool check(int x, int y)

{

if (x >= 0 && x < M && y >= 0 && y < N && board[x][y] == 0)

return true;

return false;

}

//下一位置有多少种走法

int nextPosHasSteps(int x, int y)

{

int steps = 0;

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

{

if (check(x + step[i].x, y + step[i].y))

steps++;

}

return steps;

}

//判断是否回到起点

bool returnStart(int x, int y)

{

//校验最后是否可以回到起点，也就是棋盘的中间位置

int midx,midy;

midx = M / 2 - 1;

midy = N / 2 - 1;

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

if (x + step[i].x == midx && y + step[i].y == midy)

return true;

return false;

}

//输出结果

void outputResult(int xstart,int ystart)

{

int num = M \* N;

int k = num - board[xstart][ystart];

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

{

cout<<endl<<endl;

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

{

board[i][j] = (board[i][j] + k) % num + 1;

cout<<setw(5)<<board[i][j];

}

}

cout<<endl<<endl;

}

//某一位置距离棋盘中心的距离

int posToMidLength(int x,int y)

{

int midx = M / 2 - 1;

int midy = N / 2 - 1;

return (abs(x - midx) + abs(y - midy));

}

void BackTrace(int t, int x, int y,int xstart,int ystart)

{

//找到结果

if (t == M \* N && returnStart(x,y)) //遍历了棋盘的所以位置，并且最后可以回到起点，形成回路

{

outputResult(xstart,ystart);

exit(1);

}

else

{

priority\_queue<NextPos> nextPosQueue;

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

{

if (check(x + step[i].x, y + step[i].y))

{

NextPos aNextPos;

aNextPos.nextPosSteps = nextPosHasSteps(x + step[i].x, y + step[i].y);

aNextPos.nextPosDirection = i;

aNextPos.nextPosToMidLength = posToMidLength(x + step[i].x,y + step[i].y);

nextPosQueue.push(aNextPos);

}

}

while(nextPosQueue.size())

{

int d = nextPosQueue.top().nextPosDirection;

nextPosQueue.pop();

x += step[d].x;

y += step[d].y;

board[x][y] = t + 1;

BackTrace(t + 1, x, y,xstart,ystart);

//回溯

board[x][y] = 0;

x -= step[d].x;

y -= step[d].y;

}

}

}

void horseRun(int xstart,int ystart)

{

//初始化棋盘

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

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

board[i][j] = 0;

int midx = M / 2 -1;

int midy = N / 2 -1;

board[midx][midy] = 1; //从棋盘的中间的位置开始马周游

BackTrace(1, midx, midy,xstart,ystart);

}

int main(void)

{

//马周游起始位置

int x, y;

cout<<"请输入棋盘大小m\*n|m-n|<=2 且 m和n都为偶数 且 m,n < 20 :";

cin>>M>>N;

cout<<"请输入马周游起始位置--横纵坐标0 <= x < "<<M<<"和0 <= y < "<<N<<" :";

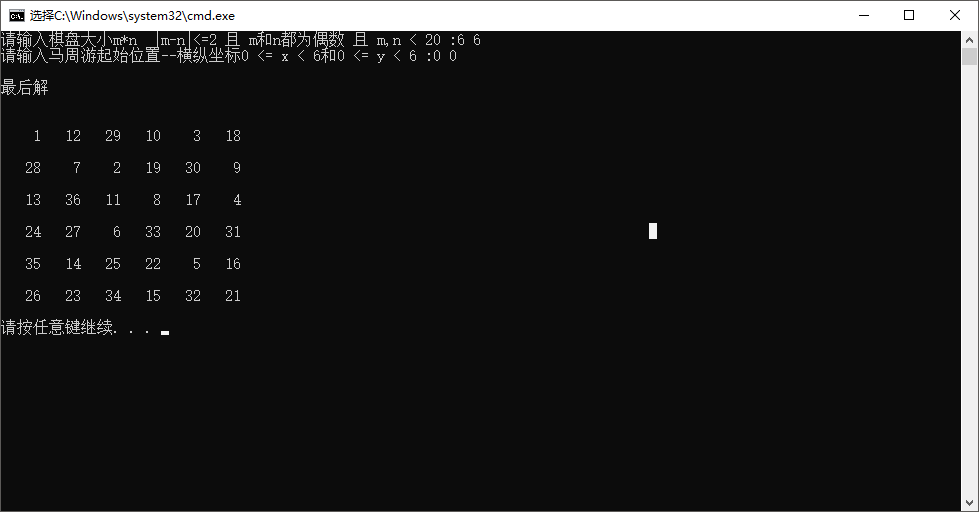
cin>>x>>y;

horseRun(x,y); //执行马周游

return 0;

}

## 1.3回溯法求解实现结果



# 二、马的Hamilton周游问题——分治解决

## 2.1分治算法的简介

2.1.1分治算法的思想

分治法的基本思想是将一个规模为n的问题分解为k个规模较小的子问题，这些子问题互相独立且与原问题相同。递归的解决这些子问题，然后将各个子问题的解合并得到原问题的解。它的一般算法设计模式如下:

其中，|P|表示问题P的规模，n0为一阀值，表示当问题P的规模不超过n0时，问题已容易解出，不必再继续分解。Adhoc（P）是该分治中的基本子算法，用于直接解小规模的问题P。当P的规模不超过n0时，直接用算法

divide-and-conquer(P)

{

if(|P|<=n0) adhoc(P);

divide P into smaller subinstances P1,P2,…,Pk;

for(i=1;i<=k;i++)

yi=divide-and-conquer(Pi);

Return merge(y1,y2,…,yk);

}

adhoc（P）求解。算法merge（y1，y2，…,yk）是分治法中的合并子算法，用于将P的子问题P1,P2,…,Pk的解y1，y2，…,yk合并为P的解。

Attention

在分治策略中，我们递归地求解一个问题，在每层递归中应用如下三个步骤：

分解（Divide）步骤将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小。

解决（Conquer）步骤递归地求解出子问题。如果子问题的规模足够小，则停止递归，直接求解。

合并（Combine）步骤将子问题的解组合成原问题的解

当子问题足够大的，需要递归求解时，我们称之为递归情况（recursive case）。当子问题变得足够小，不再需要递归时，我们说递归已经“触底”，进入了基本情况（base case）。有时，除了与原问题形式完全一样的规模更小的子问题外，还需要求解与原问题不完全一样的子问题。我们将这些子问题的求解看做合并步骤的一部分。

2.1.2分治算法的复杂性分析

从分治法的一般设计模式可以看出，用它设计出的程序一般是递归算法。因此，分治法的计算效率通常可以用递归方程来进行分析。一个分治算法将规模为n的问题分成k个规模为n/m的子问题去解。为方便起见，设分解阀值n0=1，且adhoc解规模为1的问题耗费1个单位时间。另外，再设将原问题分解为k个子问题及用merge将k个子问题的解合并为原问题的解需用f（n）个单位时间。如果用T（n）表示该分治法divide-and-conquer（P）解规模为|P|=n的问题所需的计算时间，则有

T（n）=

通常可以用展开递归式的方法来解这类递归方程，反复代入求解得

T（n）=

## 2.2马的Hamilton周游问题的分治解法

**2.2.1算法思想**

(1)马的Hamilton周游问题子问题的构造解决（Conquer）——基于回溯法

在nn的国际象棋棋盘上的一只马，可按8个不同方向移动。定义nn的国际象棋棋盘上的马步图为G（V,E），可参见图一。棋盘上的每个方格对应于图G中的一个顶点，V=。从一个顶点到另一个马步可以跳达的顶点之间有一条边E=。

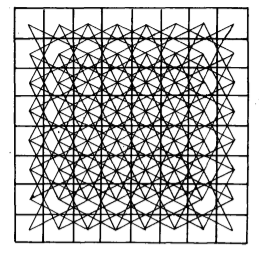


图1 国际象棋的马步图

图G有个顶点和4-12n+8条边。马的Hamilton周游路线问题即是图G的Hamilton回路问题。容易看出，当n为奇数时该问题无解。事实上，由于马在棋盘上移动的方格是黑白相间的，如果有解，则走到的黑白格子数相同，因此棋盘格子总数应该为偶数，然而为奇数，此为矛盾。下面给出的算法可以证明，当n6是偶数时，问题有解，而且可以用分治在线性的时间内构造出一个解。

考察一般的情况，即给定的国际象棋棋盘有m行和n列，且|m-n|2的情况。因此可能有mm,m和m）共三种不同规格的棋盘。为了采用分治策略，考察一类具有特殊结构的解，这类解在棋盘的4个角都包含两条

特殊的边，如图2所示，

图2 结构化的Hamilton回路

我们称具有这类特殊结构的Hamilton回路。用回溯法可在O(1)时间内找出6 ，68,88,810,10,1012棋盘上的结构化的Hamilton回路，如下所示

1 30 33 16 3 24 1 10 31 40 21 14 29 38

32 17 2 23 34 15 32 41 2 11 30 39 22 13

29 36 31 14 25 4 9 48 33 20 15 12 37 28

18 9 6 35 22 13 42 3 44 47 6 25 18 23

7 28 11 20 5 26 45 8 5 34 19 16 27 36

10 19 8 27 12 21 4 43 46 7 26 35 24 17

（6 ） （ 6）

1 46 17 50 3 6 31 52 1 46 37 66 3 48 35 68 5 8

18 49 2 7 30 51 56 5 38 65 2 47 36 67 4 7 34 69

45 64 47 16 27 4 53 32 45 80 39 24 49 18 31 52 9 6

48 19 8 29 10 55 26 57 64 23 44 21 30 15 50 19 70 33

63 44 11 22 15 28 33 54 79 40 25 14 17 20 53 32 51 10

12 41 20 9 36 23 58 25 26 63 22 43 54 29 16 73 58 71

43 62 39 14 21 60 37 34 41 78 61 28 13 76 59 56 11 74

40 13 42 61 38 35 24 59 62 27 42 77 60 55 12 75 72 57

（8） （8）

1 54 69 66 3 56 39 64 5 8 1 4 119 100 65 6 69 102 71 8 75 104

68 71 2 55 38 65 4 7 88 63 118 99 2 5 68 101 42 7 28 103 72 9

53 100 67 70 57 26 35 40 9 6 3 120 97 64 41 66 25 70 39 74 105 76

72 75 52 27 42 37 58 87 62 89 98 117 48 67 62 43 40 27 60 29 10 73

99 30 73 44 25 34 41 36 59 10 93 96 63 44 47 26 61 24 33 38 77 106

74 51 76 31 28 43 86 81 90 61 116 51 94 49 20 23 46 37 30 59 34 11

77 98 29 24 45 80 33 60 11 92 95 92 115 52 45 54 21 32 35 80 107 78

50 23 48 79 32 85 82 91 14 17 114 89 50 19 22 85 36 55 58 31 12 81

97 78 21 84 95 46 19 16 93 12 91 18 87 112 53 16 57 110 83 14 79 108

22 49 96 47 20 83 94 13 18 15 88 113 90 17 86 111 84 15 56 109 82 13

（10） （10）

其中，6，8，10棋盘上的结构化Hamilton回路，旋转90°可以得到8，10和12棋盘上的结构化Hamilton回路。

**(**2)马的Hamilton周游问题分解（Divide）

对于m，n12时，采用分治策略求解问题。

将棋盘尽可能的平均分割为4块。当m,n=4k时，分割为2个2k；当m，n=4k+2时，分割为1个2k和1个2k+2。具体实现代码如下

// 将棋盘四分

int row1 = row / 2;

if (row % 4 > 0) row1--; // 处理不能四等分的情景

int row2 = row - row1;

int col1 = col / 2;

if (col % 4 > 0) col1--;

int col2 = col - col1;

(3)马的Hamilton周游问题合并（Combine）

4个子棋盘拼接后的结构如图3所示。

分别删除4个子棋盘中的结构化边A,B,C,D，添加新边E,F,G,H构成整个棋

图3 子棋盘的拼接

盘的结构化Hamilton回路，如图4所示。

图4 子盘中Hamilton回路的合并

H

G

F

E

1

4

3

5

7

6

2

0

D

C

B

A

7

6

5

4

3

2

1

0

通过上述步骤合并后的Hamilton回路任然是结构化Hamilton回路。

2.2.2算法复杂性分析

设上述分治法计算nn棋盘上的Hamilton回路所需计算时间为T（n），则T（n）满足如下递归式：

T（n）=

解此递归式可得T（n）=O（n²）。

## 2.3算法实现——基于c++

头文件Knights.h如下：

#ifndef KNIGHT\_H

#define KNIGHT\_H

#include <vector>

using namespace std;

typedef std::pair<int, int> grid;

class Knight

{

public:

Knight(int row, int col);

virtual ~Knight();

void output();

protected:

private:

int m\_row; // 棋盘的行数

int m\_col; // 棋盘的列数

vector<grid> m\_chessboard66; // 特定大小棋盘下的一种回路信息，属于分治法中的最小解，不同的最小解合并可达到任意大的解

vector<grid> m\_chessboard68;

vector<grid> m\_chessboard86;

vector<grid> m\_chessboard88;

vector<grid> m\_chessboard810;

vector<grid> m\_chessboard108;

vector<grid> m\_chessboard1010;

vector<grid> m\_chessboard1012;

vector<grid> m\_chessboard1210;

grid\*\* m\_link; // 保存m\_row \* m\_col 棋盘的回路结果,马是从(0,0)开始走，是分治法合并后的解,first成员数据表示上一步位置，second成员数据上一步位置

bool m\_bDataOK; // 标记基础结构化数据是否载入OK

int pos(int x, int y, int col);

void step(int m, int n, const vector<int>& path\_data, vector<grid>&);

void build(int m, int n, int offx, int offy, int col, const vector<grid>& board);

void base(int row, int col, int offx, int offy);

bool comp(int m, int n, int offx, int offy);

bool load\_path();

bool load\_data(int row, int col, vector<int>& path\_data, std::ifstream& ifs);

};

#endif // KNIGHT\_H

源文件Knights.cpp如下：

#include <iostream>

#include <fstream>

#include <vector>

#include <set>

#include "Knights.h"

using namespace std;

Knight::Knight(int row, int col)

{

m\_row = row;

m\_col = col;

m\_link = new grid\*[m\_row];

for (int i = 0; i < m\_row; i++)

{

m\_link[i] = new grid[m\_col];

}

m\_bDataOK = load\_path();

}

Knight::~Knight()

{

for (int i = 0; i < m\_row; i++)

{

delete[]m\_link[i];

m\_link[i] = NULL;

}

delete[]m\_link;

m\_link = NULL;

}

void Knight::step(int row, int col, const vector<int>& path\_data, vector<grid>& b)

{

b.resize(row \* col, std::make\_pair(0, 0));

if (row <= col)

{

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

{

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

{

int p = path\_data[pos(i, j, col)] - 1;

b[p].first = i;

b[p].second = j;

}

}

}

else

{

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

{

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

{

int p = path\_data[pos(i, j, col)] - 1;

b[p].first = i;

b[p].second = j;

}

}

}

}

bool Knight::comp(int row, int col, int offx, int offy)

{

if (0 != row % 2 || 0 != col % 2

|| row - col > 2

|| col - row > 2

|| row < 6

|| col < 6)

{

cout << "Invalid/unsupported parameters of the size of the board" << endl;

return 1;

}

if (row < 12 || col < 12)

{

base(row, col, offx, offy); // base solution

return 0;

}

// 将棋盘进了均匀四分

int row1 = row / 2;

if (row % 4 > 0) row1--; // 处理不能四等分的情景

int row2 = row - row1;

int col1 = col / 2;

if (col % 4 > 0) col1--;

int col2 = col - col1;

// 递归调用解决

comp(row1, col1, offx, offy);

comp(row1, col2, offx, offy + col1);

comp(row2, col1, offx + row1, offy);

comp(row2, col2, offx + row1, offy + col1);

// 合并步骤

int x[8], y[8];

x[0] = offx + row1 - 1; y[0] = offy + col1 - 3; // 见合并步骤中的图 对应点0 1 2 3 4 5 6 7坐标

x[1] = x[0] - 1; y[1] = y[0] + 2;

x[2] = x[1] - 1; y[2] = y[1] + 2;

x[3] = x[2] + 2; y[3] = y[2] - 1;

x[4] = x[3] + 1; y[4] = y[3] + 2;

x[5] = x[4] + 1; y[5] = y[4] - 2;

x[6] = x[5] + 1; y[6] = y[5] - 2;

x[7] = x[6] - 2; y[7] = y[6] + 1;

int p[8];

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

p[i] = pos(x[i], y[i], m\_col); // 求真实大棋盘中对应的位置编号

// 真正的合并子结果

for (int i = 1; i < 8; i += 2)

{

int j1 = (i + 1) % 8;

int j2 = (i + 2) % 8;

//采用grid型二维数组m\_link记录合并结果，其中成员first数据为下一步位置，成员second数据对应上一步位置。（typedef std::pair<int, int> grid;）

if (m\_link[x[i]][y[i]].first == p[i - 1])

{

m\_link[x[i]][y[i]].first = p[j1];

}

else

{

m\_link[x[i]][y[i]].second = p[j1];

}

if (m\_link[x[j1]][y[j1]].first == p[j2])

{

m\_link[x[j1]][y[j1]].first = p[i];

}

else

{

m\_link[x[j1]][y[j1]].second = p[i];

}

}

return 0;

}

void Knight::base(int row, int col, int offx, int offy)

{

if (row == 6 && col == 6) build(row, col, offx, offy, m\_col, m\_chessboard66);

if (row == 6 && col == 8) build(row, col, offx, offy, m\_col, m\_chessboard68);

if (row == 8 && col == 6) build(row, col, offx, offy, m\_col, m\_chessboard86);

if (row == 8 && col == 8) build(row, col, offx, offy, m\_col, m\_chessboard88);

if (row == 8 && col == 10) build(row, col, offx, offy, m\_col, m\_chessboard810);

if (row == 10 && col == 8) build(row, col, offx, offy, m\_col, m\_chessboard108);

if (row == 10 && col == 10) build(row, col, offx, offy, m\_col, m\_chessboard1010);

if (row == 10 && col == 12) build(row, col, offx, offy, m\_col, m\_chessboard1012);

if (row == 12 && col == 10) build(row, col, offx, offy, m\_col, m\_chessboard1210);

}

void Knight::build(int m, int n, int offx, int offy, int col, const vector<grid>& board)

{

int k = m \* n; // k 是棋盘走局board的大小

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

{

// first step, 获取小棋盘board上第i步的坐标

int x1 = offx + board[i].first;

int y1 = offy + board[i].second;

// next step, 第i+1步的坐标

int x2 = offx + board[(i + 1) % k].first;

int y2 = offy + board[(i + 1) % k].second;

int p = pos(x1, y1, col);// 小棋盘下的坐标对应的大棋盘的第p个位置

int q = pos(x2, y2, col);// 对应第q个位置

m\_link[x1][y1].first = q;

m\_link[x2][y2].second = p;

}

}

int Knight::pos(int x, int y, int col)

{

return col \* x + y;

}

void Knight::output()

{

if (!m\_bDataOK)

return;

if (comp(m\_row, m\_col, 0, 0))

{

cout << "Unable to find a tour. " << endl;

return;

}

// 按照马走的顺序，依次打印每一步在棋盘中的x，y坐标，坐标从0开始计数

int \*\*a = new int\*[m\_row];

for (int i = 0; i < m\_row; i++)

{

a[i] = new int[m\_col];

}

for (int i = 0; i < m\_row; i++)

{

for (int j = 0; j < m\_col; j++)

{

a[i][j] = 0;

}

}

int i = 0;

int j = 0;

int k = 2;

a[0][0] = 1; // 表示第一步，马周游的起点

cout << "(0,0)" << " ";

vector<grid> finalboard;

finalboard.resize(m\_row\*m\_col, make\_pair(0, 0));

for (int p = 1; p < m\_row \* m\_col; p++)

{

int x = m\_link[i][j].first;

int y = m\_link[i][j].second;

i = x / m\_col;

j = x % m\_col;

if (a[i][j] > 0)

{

i = y / m\_col;

j = y % m\_col;

}

finalboard[k - 1] = make\_pair(i, j);

a[i][j] = k++; // 棋盘的位置p是第k步

cout << "(" << i << "," << j << ") ";

if ((k - 1) % m\_col == 0)

cout << endl;

}

cout << endl;

for (int i = 0; i < m\_row; i++)

{

for (int j = 0; j < m\_col; j++)

{

cout << a[i][j] << " ";

}

cout << endl;

}

for (int i = 0; i < m\_row; i++)

{

delete[]a[i];

a[i] = NULL;

}

delete[]a;

a = NULL;

}

bool Knight::load\_path()

{

int iterator1[9] = { 6,6,8,8,8,10,10,10,12 };

int iterator2[9] = { 6,8,6,8,10,8,10,12,10 };

vector<vector<grid>\* > grids;

grids.push\_back(&m\_chessboard66);

grids.push\_back(&m\_chessboard68);

grids.push\_back(&m\_chessboard86);

grids.push\_back(&m\_chessboard88);

grids.push\_back(&m\_chessboard810);

grids.push\_back(&m\_chessboard108);

grids.push\_back(&m\_chessboard1010);

grids.push\_back(&m\_chessboard1012);

grids.push\_back(&m\_chessboard1210);

std::ifstream ifs("board\_path.txt");

if (!ifs.is\_open())

{

cout << "Unable to load base data from board\_path.txt" << endl;

return false;

}

vector<int> a;

a.resize(12 \* 10); // 最大只需要这么多

bool bOK = true;

for (int i = 0; i < 9; i++)

{

if (!load\_data(iterator1[i], iterator2[i], a, ifs))

{

bOK = false;

break;

}

step(iterator1[i], iterator2[i], a, \*grids[i]);

}

return bOK;

}

bool Knight::load\_data(int row, int col, vector<int>& data, std::ifstream& ifs)

{

if (row \* col > data.size())

{

cout << "overflow" << endl;

return false;

}

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

{

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

{

ifs >> data[pos(i, j, col)];

}

cout << endl;

}

return true;

}

main.cpp文件如下

#include <iostream>

#include <cstdlib>

#include "Knights.h"

using namespace std;

int main()

{

Knight knight(16,16);

knight.output();

return 0;

}

board\_path.txt文件如下:

1 30 33 16 3 24

32 17 2 23 34 15

29 36 31 14 25 4

18 9 6 35 22 13

7 28 11 20 5 26

10 19 8 27 12 21

1 10 31 40 21 14 29 38

32 41 2 11 30 39 22 13

9 48 33 20 15 12 37 28

42 3 44 47 6 25 18 23

45 8 5 34 19 16 27 36

4 43 46 7 26 35 24 17

1 32 9 42 45 4

10 41 48 3 8 43

31 2 33 44 5 46

40 11 20 47 34 7

21 30 15 6 19 26

14 39 12 25 16 35

29 22 37 18 27 24

38 13 28 23 36 17

1 46 17 50 3 6 31 52

18 49 2 7 30 51 56 5

45 64 47 16 27 4 53 32

48 19 8 29 10 55 26 57

63 44 11 22 15 28 33 54

12 41 20 9 36 23 58 25

43 62 39 14 21 60 37 34

40 13 42 61 38 35 24 59

1 46 37 66 3 48 35 68 5 8

38 65 2 47 36 67 4 7 34 69

45 80 39 24 49 18 31 52 9 6

64 23 44 21 30 15 50 19 70 33

79 40 25 14 17 20 53 32 51 10

26 63 22 43 54 29 16 73 58 71

41 78 61 28 13 76 59 56 11 74

62 27 42 77 60 55 12 75 72 57

1 38 45 64 79 26 41 62

46 65 80 23 40 63 78 27

37 2 39 44 25 22 61 42

66 47 24 21 14 43 28 77

3 36 49 30 17 54 13 60

48 67 18 15 20 29 76 55

35 4 31 50 53 16 59 12

68 7 52 19 32 73 56 75

5 34 9 70 51 58 11 72

8 69 6 33 10 71 74 57

1 54 69 66 3 56 39 64 5 8

68 71 2 55 38 65 4 7 88 63

53 100 67 70 57 26 35 40 9 6

72 75 52 27 42 37 58 87 62 89

99 30 73 44 25 34 41 36 59 10

74 51 76 31 28 43 86 81 90 61

77 98 29 24 45 80 33 60 11 92

50 23 48 79 32 85 82 91 14 17

97 78 21 84 95 46 19 16 93 12

22 49 96 47 20 83 94 13 18 15

1 4 119 100 65 6 69 102 71 8 75 104

118 99 2 5 68 101 42 7 28 103 72 9

3 120 97 64 41 66 25 70 39 74 105 76

98 117 48 67 62 43 40 27 60 29 10 73

93 96 63 44 47 26 61 24 33 38 77 106

116 51 94 49 20 23 46 37 30 59 34 11

95 92 115 52 45 54 21 32 35 80 107 78

114 89 50 19 22 85 36 55 58 31 12 81

91 18 87 112 53 16 57 110 83 14 79 108

88 113 90 17 86 111 84 15 56 109 82 13

1 118 3 98 93 116 95 114 91 88

4 99 120 117 96 51 92 89 18 113

119 2 97 48 63 94 115 50 87 90

100 5 64 67 44 49 52 19 112 17

65 68 41 62 47 20 45 22 53 86

6 101 66 43 26 23 54 85 16 111

69 42 25 40 61 46 21 36 57 84

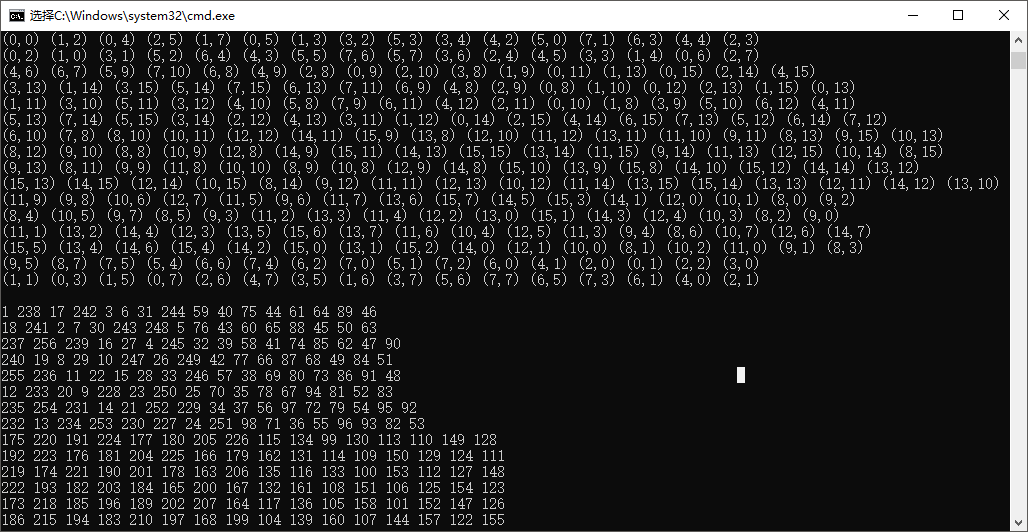
102 7 70 27 24 37 32 55 110 15

71 28 39 60 33 30 35 58 83 56

8 103 74 29 38 59 80 31 14 109

75 72 105 10 77 34 107 12 79 82

104 9 76 73 106 11 78 81 108 13

2.4 16棋盘下马的Hamilton回路程序运行求解结果：

# 参考文献

[1]. 曹新谱,肖宝麟; 国际象棋棋盘上马的周游路线问题[J];重庆大学学报(自然

科学版);1998年04期

[2].肖金声;骑士巡游问题的解[J];中山大学学报（自然科学版）；1994年03期

[3] .王晓东. 算法设计与分析习题解答（第3版）[M]. 北京:王晓东, 2014.