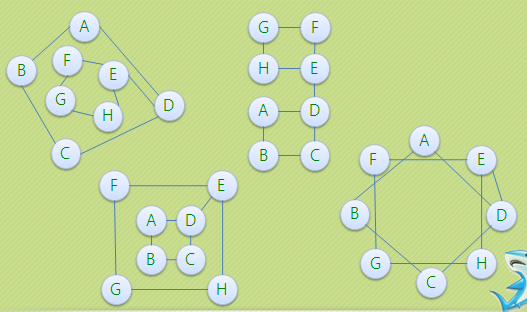
# 56图的存储结构

## 图的存储结构

* 图的存储结构相比较线性表与树来说就复杂很多。
* 我们回顾下，对于线性表来说，是一对一的关系，所以用数组或者链表均可简单存放。树结构是一对多的关系，所以我们要将数组和链表的特性结合在一起才能更好的存放。
* 那么我们的图，是多对多的情况，另外图上的任何一个顶点都可以被看作是第一个顶点，任一顶点的邻接点之间也不存在次序关系。
* 我们仔细观察以下几张图，然后深刻领悟一下：



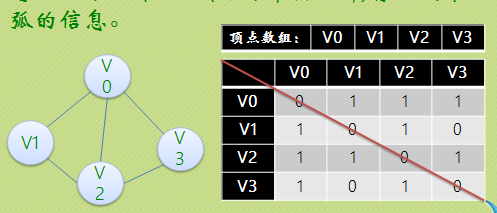
* 因为任意两个顶点之间都可能存在联系，因此无法以数据元素在内存中的物理位置来表示元素之间的关系（内存物理位置是线性的，图的元素关系是平面的）。
* 如果用多重链表来描述倒是可以做到，但在几节课前的树章节我们已经讨论过，纯粹用多重链表导致的浪费是无法想像的（如果各个顶点的度数相差太大，就会造成巨大的浪费）。
* 所幸，前辈们已经帮想好了出路，我们接下来会谈图的五种不同的存储结构，大家做好准备哦~

## 邻接矩阵（无向图）

* 考虑到图是由顶点和边或弧两部分组成，合在一起比较困难，那就很自然地考虑到分为两个结构来分别存储。
* 顶点因为不区分大小、主次，所以用一个一维数组来存储是很不错的选择。
* 而边或弧由于是顶点与顶点之间的关系，一维数组肯定就搞不定了，那我们不妨考虑用一个二维数组来存储。

于是我们的邻接矩阵方案就诞生了！

* 图的邻接矩阵(Adjacency Matrix)存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组(称为邻接矩阵)存储图中的边或弧的信息。



* 我们可以设置两个数组，顶点数组为vertex[4]={V0,V1,V2,V3}，边数组arc[4][4]为对称矩阵(0表示不存在顶点间的边，1表示顶点间存在边)。
* 对称矩阵：所谓对称矩阵就是n阶矩阵的元满足a[i][j]=a[j][i](0<=i,j<=n)。即从矩阵的左上角到右下角的主对角线为轴，右上角的元与左下角相对应的元全都是相等的。
* 有了这个二维数组组成的对称矩阵，我们就可以很容易地知道图中的信息：
  + 要判定任意两顶点是否有边无边就非常容易了；
  + 要知道某个顶点的度，其实就是这个顶点Vi在邻接矩阵中第i行(或第i列)的元素之和；
  + 求顶点Vi的所有邻接点就是将矩阵中第i行元素扫描一遍，arc[i][j]为1就是邻接点咯。

邻接矩阵（有向图）

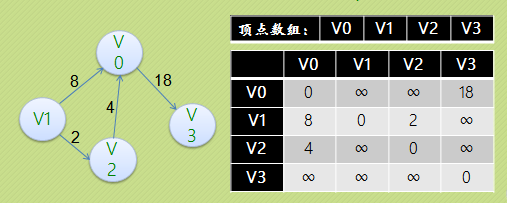
* 无向图的边构成了一个对称矩阵，貌似浪费了一半的空间，那如果是有向图来存放，会不会把资源都利用得很好呢？



* 可见顶点数组vertex[4]={V0,V1,V2,V3}，弧数组arc[4][4]也是一个矩阵，但因为是有向图，所以这个矩阵并不对称，例如由V1到V0有弧，得到arc[1][0]=1，而V0到V1没有弧，因此arc[0][1]=0。
* 另外有向图是有讲究的，要考虑入度和出度，顶点V1的入度为1，正好是第V1列的各数之和，顶点V1的出度为2，正好是第V1行的各数之和。

邻接矩阵（网）

* 在图的术语中，我们提到了网这个概念，事实上也就是每条边上带有权的图就叫网。



* 这里“∞”表示一个计算机允许的、大于所有边上权值的值。

代码实现

* 作为一个课后作业给大家自己锻炼下，小甲鱼提供的参考答案仅供参考借鉴！

CreateMGraph.c

// 时间复杂度为O(n+n^2+e)

#define MAXVEX 100 // 最大顶点数

#define INFINITY 65535 // 用65535来代表无穷大

typedef struct

{

char vexs[MAXVEX]; // 顶点表

int arc[MAXVEX][MAXVEX]; // 邻接矩阵

int numVertexes, numEdges; // 图中当前的顶点数和边数

} MGraph;

// 建立无向网图的邻接矩阵

void CreateMGraph(MGraph \*G)

{

int i, j, k, w;

printf("请输入顶点数和边数：\n");

scanf("%d %d", &G->numVertexes, &G->numEdges);

for( i=0; i < G->numVertexes; i++ )

{

scanf("%c", &G->vexs[i]);

}

for( i=0; i < G->numVertexes; i++ )

{

for( j=0; j < G->numVertexes; j++ )

{

G->arc[i][j] = INFINITY; // 邻接矩阵初始化

}

}

for( k=0; k < G->numEdges; k++ )

{

printf("请输入边(Vi,Vj)上的下标i,下标j和对应的权w:\n"); // 这只是例子，提高用户体验需要进行改善

scanf("%d %d %d", &i, &j, &w);

G->arc[i][j] = w;

G->arc[j][i] = G->arc[i][j]; // 是无向网图，对称矩阵

}

}

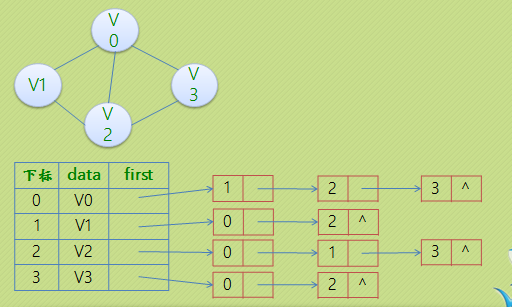
# 57图的存储结构（邻接表）

## 邻接表（无向图）

* 邻接矩阵看上去是个不错的选择，首先是容易理解，第二是索引和编排都很舒服~
* 但是我们也发现，对于边数相对顶点较少的图，这种结构无疑是存在对存储空间的极大浪费。



* 因此我们可以考虑另外一种存储结构方式，例如把数组与链表结合一起来存储，这种方式在图结构也适用，我们称为邻接表(AdjacencyList)。
* 邻接表的处理方法是这样：
  + 图中顶点用一个一维数组存储，当然，顶点也可以用单链表来存储，不过数组可以较容易地读取顶点信息，更加方便。
  + 图中每个顶点Vi的所有邻接点构成一个线性表，由于邻接点的个数不确定，所以我们选择用单链表来存储。



邻接表（有向图）

* 若是有向图，邻接表结构也是类似的，我们先来看下把顶点当弧尾建立的邻接表，这样很容易就可以得到每个顶点的出度：



* 但也有时为了便于确定顶点的入度或以顶点为弧头的弧，我们可以建立一个有向图的逆邻接表：



* 此时我们很容易就可以算出某个顶点的入度或出度是多少，判断两顶点是否存在弧也很容易实现。

邻接表（网）

* 对于带权值的网图，可以在边表结点定义中再增加一个数据域来存储权值即可：



代码实现

* 作为一个课后作业给大家自己锻炼下，小甲鱼提供的参考答案仅供参考借鉴！

CreateALGraph.c

#define MAXVEX 100

typedef struct EdgeNode // 边表结点

{

int adjvex; // 邻接点域，存储该顶点对应的下标

int weight; // 用于存储权值，对于非网图可以不需要

struct EdgeNode \*next; // 链域，指向下一个邻接点

} EdgeNode;

typedef struct VertexNode // 顶点表结点

{

char data; // 顶点域，存储顶点信息

EdgeNode \*firstEdge; // 边表头指针

} VertexNode, AdjList[MAXVEX];

typedef struct

{

AdjList adjList;

int numVertexes, numEdges; // 图中当前顶点数和边数

} GraphAdjList;

// 建立图的邻接表结构

void CreateALGraph(GraphAdjList \*G)

{

int i, j, k;

EdgeNode \*e;

printf("请输入顶点数和边数：\n");

scanf("%d %d", &G->numVertexes, &G->numEdges);

// 读取顶点信息，建立顶点表

for( i=0; i < G->numVertexes; i++ )

{

scanf("%c", &G->adjList[i].data);

G->adjList[i].firstEdge = NULL; // 初始化置为空表

}

for( k=0; k < G->numEdges; k++ )

{

printf("请输入边(Vi,Vj)上的顶点序号：\n");

scanf("%d %d", &i, &j);

e = (EdgeNode \*)malloc(sizeof(EdgeNode));

e->adjvex = j; // 邻接序号为j

e->next = G->adjList[i].firstEdge;

G->adjList[i].firstEdge = e;

e = (EdgeNode \*)malloc(sizeof(EdgeNode));

e->adjvex = i; // 邻接序号为i

e->next = G->adjList[j].firstEdge;

G->adjList[j].firstEdge = e;

}

}

# 58图的存储结构（十字链表、邻接多重表、边集数组）

## 十字链表

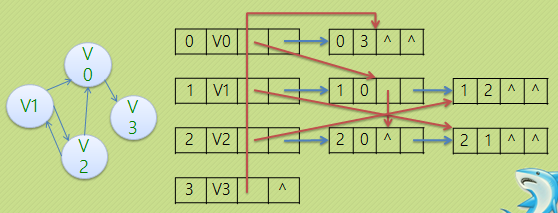
* 邻接表固然优秀，但也有不足，例如对有向图的处理上，有时候需要再建立一个逆邻接表~
* 那我们思考了：有没有可能把邻接表和逆邻接表结合起来呢？
* 答案是肯定的，这就是我们现在要谈的十字链表(Orthogonal List)

为此我们重新定义顶点表结点结构：

|  |  |  |
| --- | --- | --- |
| **data** | **firstIn** | **firstOut** |

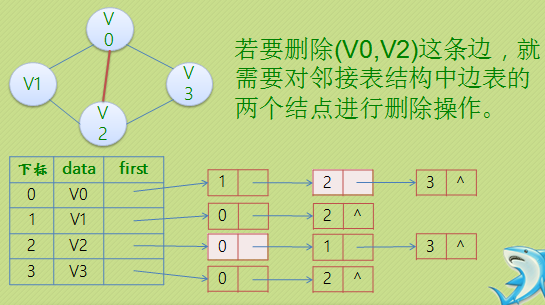
* 接着重新定义边表结点结构：

|  |  |  |  |
| --- | --- | --- | --- |
| **tailVex** | **headVex** | **headLink** | **tailLink** |

* 
* 十字链表的好处就是因为把邻接表和逆邻接表整合在了一起，这样既容易找到以Vi为尾的弧，也容易找到以Vi为头的弧，因而容易求得顶点的出度和入度。
* 十字链表除了结构复杂一点外，其实创建图算法的时间复杂度是和邻接表相同的，因此，在有向图的应用中，十字链表也是非常好的数据结构模型。

## 邻接多重表

* 讲了有向图的优化存储结构，对于无向图的邻接表，有没有问题呢？
* 如果我们在无向图的应用中，关注的重点是顶点的话，那么邻接表是不错的选择，但如果我们更关注的是边的操作，比如对已经访问过的边做标记，或者删除某一条边等操作，邻接表就显得不那么方便了。
* 到底有多烦？小甲鱼用图片告诉你：

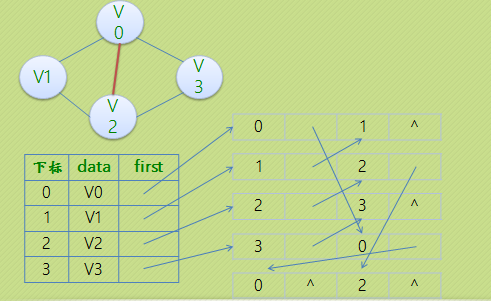


* 因此，我们也仿照十字链表的方式，对边表结构进行改装，重新定义的边表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **iVex** | **iLink** | **jVex** | **jLink** |

* 其中iVex和jVex是与某条边依附的两个顶点在顶点表中的下标。iLink指向依附顶点iVex的下一条边，jLink指向依附顶点jVex的下一条边。
* 也就是说在邻接多重表里边，边表存放的是一条边，而不是一个顶点。

不急，马上进入No pic you say a J8!环节~



## 边集数组

* 边集数组是由两个一维数组构成，一个是存储顶点的信息，另一个是存储边的信息，这个边数组每个数据元素由一条边的起点下标(begin)、终点下标(end)和权(weight)组成。

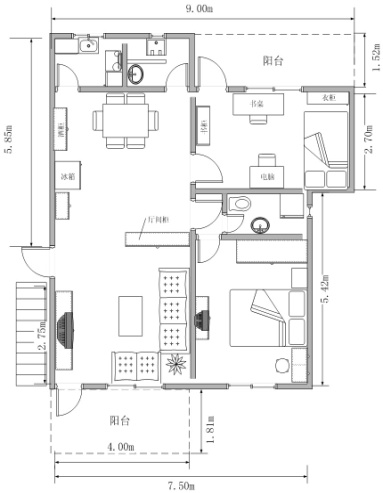


弗洛伊德的冰山理论



# 59图的遍历（深度优先遍历）

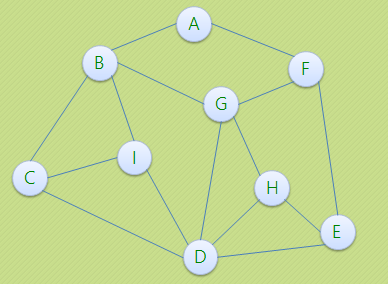
## 图的遍历



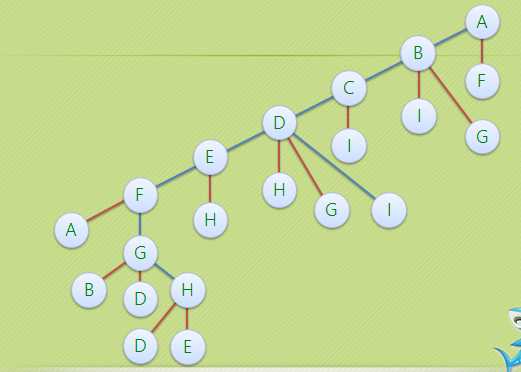
* 树的遍历我们谈了四种方式，大家回忆一下，树因为根结点只有一个，并且所有的结点都只有一个双亲，所以不是很难理解。
* 但是谈到图的遍历，那就复杂多了，因为它的任一顶点都可以和其余的所有顶点相邻接，因此极有可能存在重复走过某个顶点或漏了某个顶点的遍历过程。
* 对于图的遍历，如果要避免以上情况，那就需要科学地设计遍历方案，通常有两种遍历次序方案：它们是深度优先遍历和广度优先遍历。

## 深度优先遍历

* 深度优先遍历(DepthFirstSearch)，也有称为深度优化搜索，简称为DFS。
* 它的具体思想类似于课程开头讲的找钥匙方案，无论从哪一间房间开始都可以，将房间内的墙角、床头柜、床上、床下、衣柜、电视柜等挨个寻找，做到不放过任何一个死角，当所有的抽屉、储藏柜中全部都找遍，接着再寻找下一个房间。
* 现在请大家一起来想办法走以下这个迷宫，要求



* 我们可以约定右手原则：在没有碰到重复顶点的情况下，分叉路口始终是向右手边走，每路过一个顶点就做一个记号。
* 接下来有情小甲鱼童鞋带我们走迷宫去。
* 迷宫走完了，所有的顶点也遍历过了，这就是深度优先遍历！
* 反应快的童鞋一定会感觉深度优先遍历其实就是一个递归的过程嘛~
* 如果再细心观察，你会发现整个遍历过程就像是一棵树的前序遍历！



看动画写代码

* 请观看图的深度优先遍历（邻接矩阵实现）的原理动画，结合经验自己先尝试完成代码部分。
* 小甲鱼将分别提供给大家邻接矩阵和邻接表的实现参考方案！

AdList.c

// 邻接表的深度有限递归算法

// 邻接表的创建代码见第五十七讲源代码部分

// 鱼C工作室(www.fishc.com)

#define TRUE 1

#define FALSE 0

#define MAX 256

typedef int Boolean; // 这里我们定义Boolean为布尔类型，其值为TRUE或FALSE

Boolean visited[MAX]; // 访问标志的数组

void DFS(GraphAdjList GL, int i)

{

EdgeNode \*p;

visited[i] = TRUE;

printf("%c " GL->adjList[i].data);

p = GL->adjList[i].firstEdge;

while(p)

{

if( !visited[p->adjvex] )

{

DFS(GL, p->adjvex);

}

p = p->next;

}

}

// 邻接表的深度遍历操作

void DFSTraverse(GraphAdjList GL)

{

int i;

for( i=0; i < GL->numVertexes; i++ )

{

visited[i] = FALSE; // 初始化所有顶点状态都是未访问过状态

}

for( i=0; i < GL->numVertexes; i++ )

{

if( !visited[i] ) // 若是连通图，只会执行一次

{

DFS(GL, i);

}

}

}

AdMatrix\_DFS.c

// 邻接矩阵的深度有限递归算法

// 邻接矩阵的创建代码见第五十六讲源代码部分

// 鱼C工作室(www.fishc.com)

#define TRUE 1

#define FALSE 0

#define MAX 256

typedef int Boolean; // 这里我们定义Boolean为布尔类型，其值为TRUE或FALSE

Boolean visited[MAX]; // 访问标志的数组

void DFS(MGraph G, int i)

{

int j;

visited[j] = TRUE; // 访问过的顶点设置为TRUE

printf("%c ", G.vexs[i]); // 打印顶点

for( j=0; j < G.numVertexes; j++ )

{

if( G.arc[i][j]==1 && !visited[j] )

{

DFS(G, j); // 对为访问的邻接顶点递归调用

}

}

}

// 邻接矩阵的深度遍历操作

void DFSTraverse(MGraph G)

{

int i;

for( i=0; i < G.numVertexes; i++ )

{

visited[i] = FALSE; // 初始化所有顶点状态都是未访问过状态

}

for( i=0; i < G.numVertexes; i++ )

{

if( !visited[i] ) // 若是连通图，只会执行一次

{

DFS(G, i);

}

}

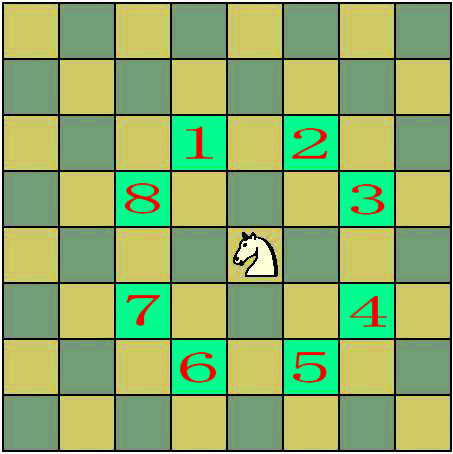
}

# 60马踏棋盘算法（骑士周游问题）

## 马踏棋盘算法（骑士周游问题）

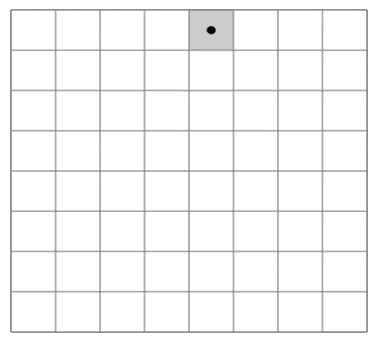
* 题目渊源：
  + 马踏棋盘问题（又称骑士周游或骑士漫游问题）是算法设计的经典问题之一。
* 题目要求：
  + 国际象棋的棋盘为8\*8的方格棋盘，现将“马”放在任意指定的方格中，按照“马”走棋的规则将“马”进行移动。要求每个方格只能进入一次，最终使得“马”走遍棋盘64个方格。
  + 编写代码，实现马踏棋盘的操作，要求用1~64来标注“马”移动的路径（看演示）。

关于国际象棋“马”的走法



马踏棋盘的一个解

* 对于在n\*n的棋盘上，当n>=5且为偶数的时候，以任意点作点都有解。



## 一些相关的知识点

* 回溯法：
  + 之前我们谈过回溯法，还是那句话，指导思想很简单，就是一条路走到黑，碰壁了再回来一条路走到黑......一般和递归可以很好的搭配使用，还有深度优先搜索（DFS）。
* 哈密尔顿路径：

图G中的哈密尔顿路径指的是经过图G中每个顶点，且只经过一次的一条轨迹。如果这条轨迹是一条闭合的路径（从起点出发不重复地遍历所有点后仍能回到起始点），那么这条路径称为哈密尔顿回路

算法描述

* 那么就让我们愉快的开始今天的代码之旅吧！

TravelChessBoard.c

#include <stdio.h>

#include <time.h>

#define X 8

#define Y 8

int chess[X][Y];

// 找到基于(x,y)位置的下一个可走的位置

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 = \*x + 2;

\*y = \*y - 1;

return 1;

}

break;

case 1:

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

{

\*x = \*x + 2;

\*y = \*y + 1;

return 1;

}

break;

case 2:

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

{

\*x = \*x + 1;

\*y = \*y - 2;

return 1;

}

break;

case 3:

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

{

\*x = \*x + 1;

\*y = \*y + 2;

return 1;

}

break;

case 4:

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

{

\*x = \*x - 2;

\*y = \*y - 1;

return 1;

}

break;

case 5:

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

{

\*x = \*x - 2;

\*y = \*y + 1;

return 1;

}

break;

case 6:

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

{

\*x = \*x - 1;

\*y = \*y - 2;

return 1;

}

break;

case 7:

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

{

\*x = \*x - 1;

\*y = \*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");

}

// 深度优先遍历棋盘

// (x,y)为位置坐标

// tag是标记变量，每走一步，tag+1

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

{

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

chess[x][y] = tag;

// 如果tag==X\*Y，则完成整个棋盘的遍历

if( tag == X\*Y )

{

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;

}

return 0;

}

int main()

{

int i, j;

clock\_t start, finish;

start = clock();

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

{

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

{

chess[i][j] = 0;

}

}

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

{

printf("抱歉，马踏棋盘失败鸟~\n");

}

finish = clock();

printf("\n本次计算一共耗时: %f秒\n\n", (double)(finish-start)/CLOCKS\_PER\_SEC);

return 0;

}