# 《人工智能原理》课程实验报告

## 搜索算法实验



**姓名： 李金泽**

**学号： 18S103201**

**专业： 计算机技术**

**日期： 2018.12.19**

## 1 问题描述

### 1.1 待解决问题的解释

人工避障问题是当今研究的热点问题，成熟的人工避障算法可以广泛地应用于扫地机器人，无人驾驶以及智能物流之中。我们可以利用启发式算法进行路径搜索以实现物体的避障与自动行驶。在本次实验中，本人使用比较经典的启发式算法A\*算法进行搜索以实现目标的避障。启发式算法（heuristic algorithm)是相对于[最优化](https://baike.baidu.com/item/%E6%9C%80%E4%BC%98%E5%8C%96)算法提出的。一个问题的最优算法求得该问题每个实例的[最优解](https://baike.baidu.com/item/%E6%9C%80%E4%BC%98%E8%A7%A3/5208902)。启发式算法可以这样定义：一个基于直观或经验构造的算法，在可接受的花费（指计算时间和空间）下给出待解决组合优化问题每一个实例的一个[可行解](https://baike.baidu.com/item/%E5%8F%AF%E8%A1%8C%E8%A7%A3/962143)，该可行解与最优解的偏离程度一般不能被预计。

### 1.2 问题的搜索形式描述

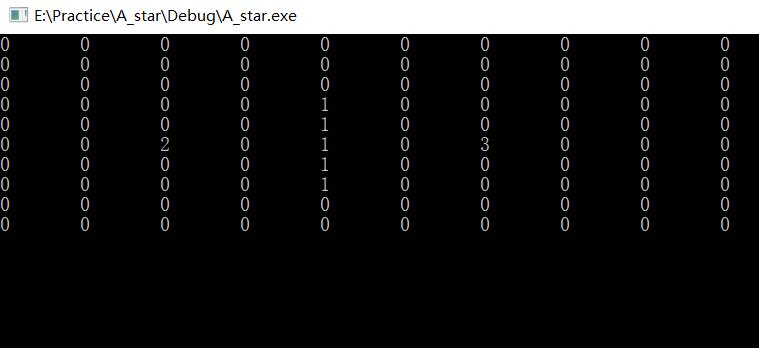


图1 整个图的初始状态

**1.初始状态：**

这里我们定义了一个10\*10的地图，地图上用0,1,2,3分别表示图上该点可以达到，该点是障碍物，该点是起始点以及该点是终点。从上图中我们可以看出，地图的起点在(5,2)这个位置，地图的终点在(5,6)这个位置，障碍物类似于一面竖直的墙。

**2.后继函数：**

移动规则：每一个点可以从八个方向进行移动，例如(5,2)→(4,3)，这是起点在向斜右上方移动。

**3.目标测试：**

判断向量是否是目标状态。当OpenList中F值最小的那个节点正好是目的节点时，我们认为已经找到了从起始节点到目的节点的最短路径。

**4.路径耗散函数：**

路径耗散函数可以表示为： 其中表示从开始点S到当前块n的最小移动量。该数值随着离起始点S越远而增大，并且每次增加的数值只有10和14两种。在这里我们定义为块n的父节点距离起始点S的最短移动距离。表示从当前块n到目标点D的移动量估算值。这个常常被称为试探，因为我们不能确定移动量是多少，仅仅是一个估算值。在本次实验中，移动量被定义为曼哈顿距离。

### 1.3 解决方案介绍

一般来说迪杰斯特拉算法，基于启发的最佳优先搜索(BFS)以及基于启发算法的A\*算法都可以很好的解决避障寻路问题。Dijkstra算法从物体所在的初始点开始，访问图中的结点。它迭代检查待检查结点集中的结点，并把和该结点最靠近的尚未检查的结点加入待检查结点集。该结点集从初始结点向外扩展，直到到达目标结点。Dijkstra算法保证能找到一条从初始点到目标点的最短路径，只要所有的边都有一个非负的代价值。在下图中，粉红色的结点是初始结点，蓝色的是目标点，而类菱形的有色区域则是Dijkstra算法扫描过的区域。颜色最淡的区域是那些离初始点最远的，因而形成探测过的边境

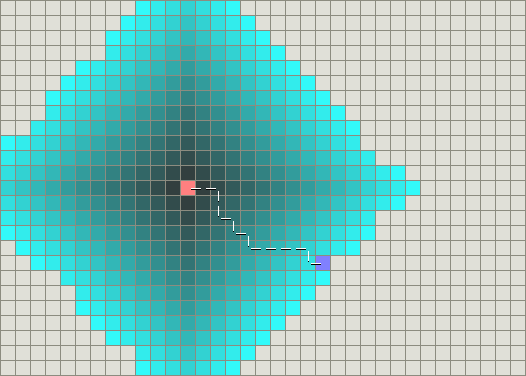


图2 迪杰斯特拉算法所有加入过OpenList的点的集合(无障碍)

最佳优先搜索（BFS）算法按照类似的流程运行，不同的是它能够评估（称为启发式的）任意结点到目标点的代价。与选择离初始结点最近的结点不同的是，它选择离目标最近的结点。BFS不能保证找到一条最短路径。然而，它比Dijkstra算法快的多，因为它用了一个启发式函数快速地导向目标结点。在下面的图中，越黄的结点代表越高的启发式值（移动到目标的代价高），而越黑的结点代表越低的启发式值（移动到目标的代价低）。这表明了与Dijkstra 算法相比，BFS运行得更快。

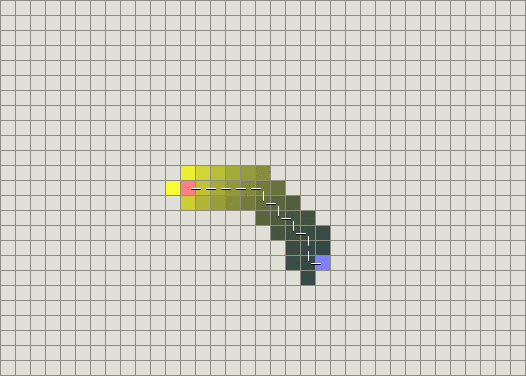


图3 BFS算法所有加入过OpenList的点的集合(无障碍)

从图2和图3中我们可以看出，当寻路问题中不存在障碍的情况下，迪杰斯特拉的计算量更大，加入OpenList的点更多，寻找出最短路径的时间越长。而BFS算法因为贪心的性质，算法的计算量很小，加入OpenList的点更少，寻优路径的时间比较短。

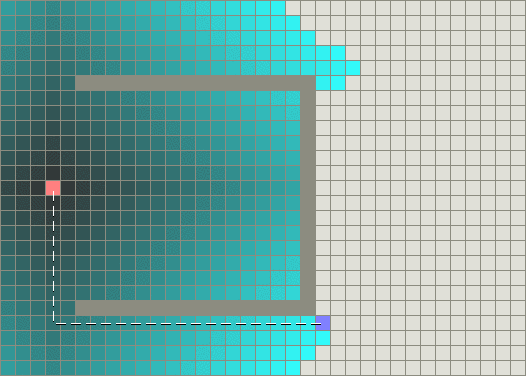


图4 Dijkstra在有障碍的情况下寻路的结果

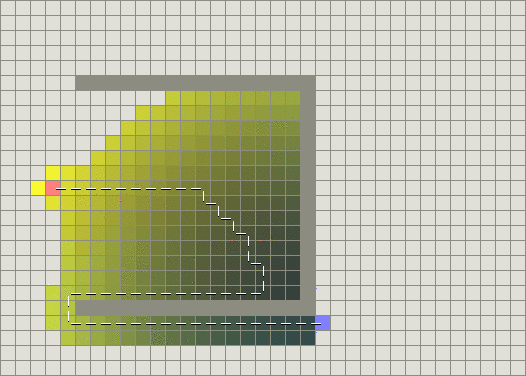


图5 BFS在有障碍的情况下寻路的结果

从图4和图5中我们可以看出虽然迪杰斯特拉算法运行的很慢，但是确实能找到一条最短的路径。BFS虽然比迪杰斯特拉算法运行的快，但是它找的路径不是一条好的路径。问题的原因在于BFS是基于贪心策略的，它试图向目标移动尽管这不是正确的路径。由于它仅仅考虑到达目标的代价，而忽略了当前已花费的代价。于是尽管路径变得很长，它仍然还要继续走下去。

对此我们提出了一个新的看法，我们可以将启发式方法例如BFS与传统的迪杰斯特拉算法结合在一起。

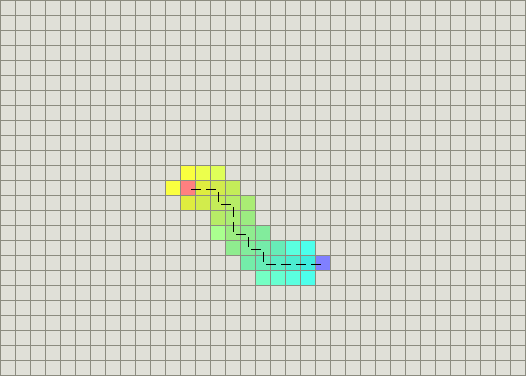


图6 A\*算法在没有障碍的情况下加入OpenList点的集合

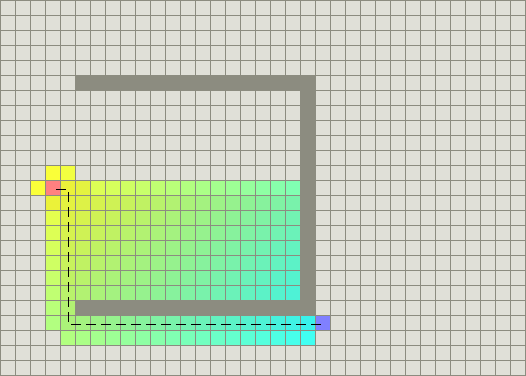


图7 A\*算法在有障碍的情况下加入OpenList点的集合

从图6中可以看出A\*和迪杰斯特拉算法一样能搜索最短路径，和BFS一样，能用启发函数引导自己。在没有障碍的情况下，A\*算法能做到和BFS一样快。从图7中我们可以看出，加入启发函数可以使得OpenList入栈次数更少，降低了运算量减少了运算时间。在评价函数中加入距离起始点距离的损失函数可以避免类似于图5的那种迂回路径。

## 2. 算法介绍

### 2.1 搜索算法一般介绍

通过Dijkstra计算图G中的最短路径时，需要指定起点vs(即从顶点vs开始计算)。此外，引进两个集合S和U。S的作用是记录已求出最短路径的顶点(以及相应的最短路径长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到起点vs的距离)。初始时，S中只有起点vs；U中是除vs之外的顶点，并且U中顶点的路径是"起点vs到该顶点的路径"。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点和顶点对应的路径。重复该操作，直到遍历完所有顶点。

2.2 算法伪代码

1. 把起点加入 open list 。

2. 重复如下过程：

　　a. 遍历open list ，查找F值最小的节点，把它作为当前要处理的节点，然后移到close list中

　　b. 对当前方格的 8 个相邻方格一一进行检查，如果它是不可抵达的或者它在close list中，忽略它。否则，做如下操作：

　　1)如果它不在open list中，把它加入open list，并且把当前方格设置为它的父亲

　　2)如果它已经在open list中，检查这条路径是否更近。如果更近，把它的父亲设置为当前方格，并重新计算它的G和F值。如果你的open list是按F值排序的话，改变后你可能需要重新排序。

　　c. 遇到下面情况停止搜索：

　　把终点加入到了 open list 中，此时路径已经找到了，或者查找终点失败，并且open list 是空的，此时没有路径。

3. 从终点开始，每个方格沿着父节点移动直至起点，形成路径。

## 3 算法实现

### 3.1 实验环境与问题规模

实验环境：Visiual Stdio2015

问题规模：空间消耗较大，搜索空间处于目标等值线内的节点数量是求解路径长度的指数级。时间复杂度也是求解路径的指数倍。

### 3.2 数据结构

**输入结构：**

int InputArray[10][10] = {

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 1, 1, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 2, 0, 1, 0, 3, 0, 0, 0 },

{ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 1, 1, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } };

**节点结构：**

typedef struct LNode {

int data; //对应数组中的数值

int F; //F = G + H;

int G; //G：从起点 A 移动到指定方格的移动代价，沿着到达该方格而生成的路径

int H; //H：从指定的方格移动到终点 B 的估算成本

int x, y; //对应数组中的坐标

bool OPen\_flag; //在开放列表中为1，不在为0

bool Close\_flag; //在关闭列表中为1，不在为0

struct LNode\* next; //用于链表排序

struct LNode\* path\_next; //用于最终找到的路径

}LNode, \*LinkList;

**两个表，open表和close表**

LinkList open\_List = InitList(); //定义并初始化一个开放列表

LinkList close\_List = InitList(); //一个封闭列表

### 3.3 实验结果

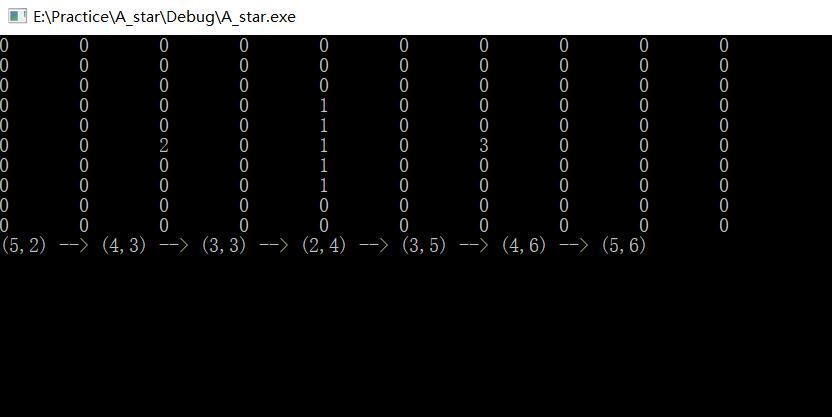


图8 竖直形障碍的A\*寻路结果

从图8中我们可以看出当图中的障碍为竖直形障碍的时候，A\*算法可以根据搜索到的目标状态回溯并且输出到最优路径。

### 3.4 系统中间及最终输出结果

**实例1 凹字形障碍**

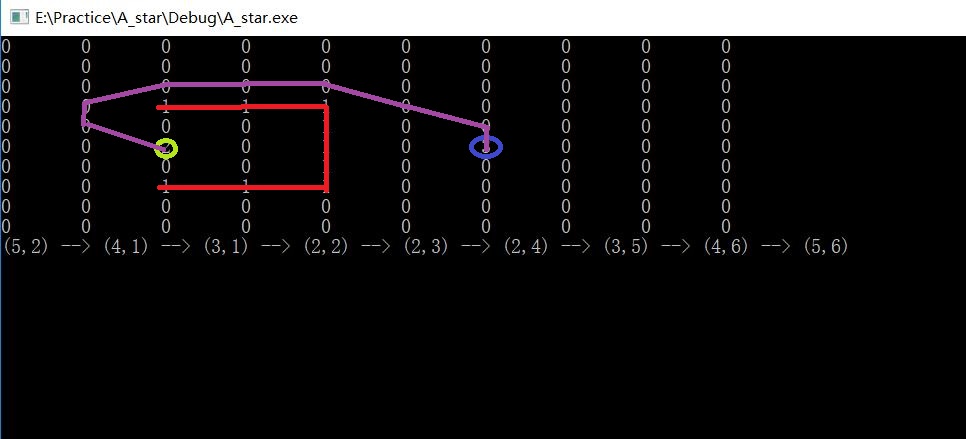


图9 凹字形障碍物的最短寻路结果

**实例2 只留一个出口的全包障碍**

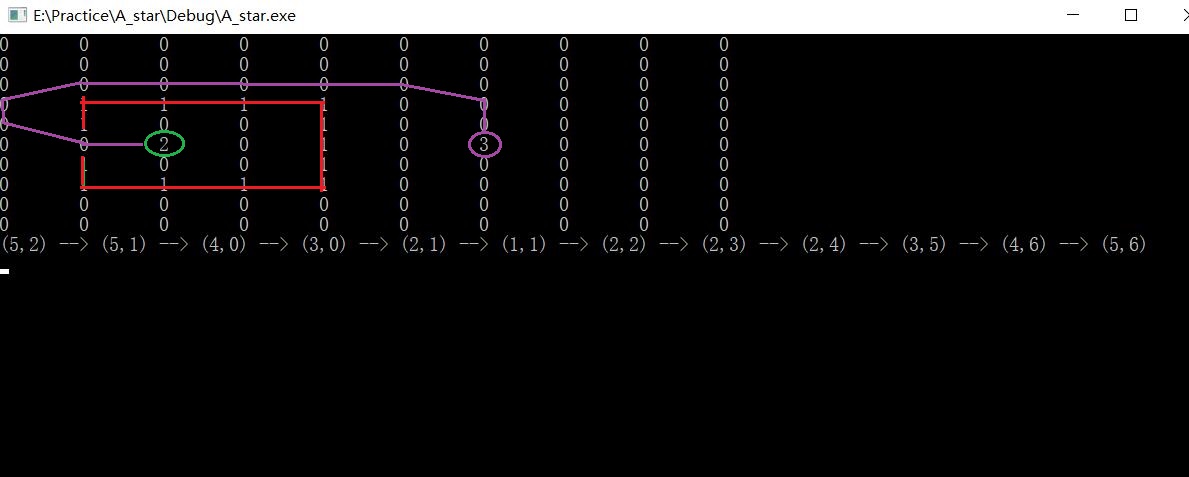


图10 只有一个出口的全包障碍的寻路结果

在图9和图10中，绿色代表起点，蓝色代表终点，紫色代表搜索得到的路径，红色代表障碍物的位置。可以看出A\*算法可以较为准确地找出最短路径解决现实中的避障问题。

### 附录一 源代码及其注释

程序中共三个文件。其中”Apath.h”包含各函数的定义以及数据结构的定义。”Apath.cpp”包含个函数的实现，”main.cpp”是主函数。源代码级注释如下：

**Main.cpp**

#include <stdio.h>

//#ifndef APATH\_H

#include "Apath.h"

#include <iostream>

#include <vector>

using namespace std;

//#endif

//为简单，干脆把把下面数组转为链表结构的数组

//约定：0是可走的，1表示障碍物不可走，2表示起点，3表示终点，4表示路径

int InputArray[10][10] = {

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 1, 1, 1, 1, 0, 0, 0, 0, 0 },

{ 0, 1, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 2, 0, 1, 0, 3, 0, 0, 0 },

{ 0, 1, 0, 0, 1, 0, 0, 0, 0, 0 },

{ 0, 1, 1, 1, 1, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 } };

int main()

{

vector<vector<int>> ResultPath;// 存储最后的路径

int row = MAP\_MAX\_X, col = MAP\_MAX\_Y;

//printf("hello world!\n");

LNode \*\*map = Translate\_array(InputArray, row, col); //这里将数组的地图转为节点map的地图

output(map, 10, 10);

LinkList open\_List = InitList(); //定义并初始化一个开放列表

LinkList close\_List = InitList(); //一个封闭列表

LNode\* startLNode = find\_start\_LNode(map, row, col);

LNode\* endLNode = find\_end\_LNode(map, row, col);

LNode\* curLNode = startLNode; //当前节点=开始节点

curLNode->G = 0; //计算节点的三个值

count\_LNode\_H(curLNode, endLNode);

count\_LNode\_F(curLNode);

push\_OpenList\_Node(open\_List, curLNode); //先将开始节点插入开放列表

while (curLNode->data != 3)

{

//LNode \*e = NULL;

curLNode = pop\_OpenList\_minNode(open\_List);

insert\_Into\_CloseList(curLNode, close\_List);

//2、查看起点周围的点是否在开放列表里，不在加入，在检测经过该点F值是否最小等；

check\_around\_curNode(curLNode, endLNode, open\_List, map);

}

while (endLNode->path\_next)

{

vector<int> temp;

temp.push\_back(endLNode->path\_next->x);

temp.push\_back(endLNode->path\_next->y);

//printf("x:%d---y:%d\n", endLNode->path\_next->x, endLNode->path\_next->y);

endLNode->path\_next = endLNode->path\_next->path\_next;

ResultPath.push\_back(temp);

}

for (vector<vector<int>>::iterator it = ResultPath.end() - 1; it >= ResultPath.begin(); it--)

{

cout << "(" << (\*it)[0]<<","<<(\*it)[1] << ") --> ";

if (it == ResultPath.begin()) { break; }

}

cout << "(" << endLNode->x << ","<< endLNode->y <<")" << endl;

getchar();

return 0;

}

**Apath.h**

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stddef.h>

#include <stdbool.h>

#ifndef APATH\_H

#define APATH\_H

#endif

#define TURE 1

#define FAULT 0

//约定：0是可走的，1表示障碍物不可走，2表示起点，3表示终点，4表示路径

#define int\_0 0

#define int\_1 1

#define int\_2 2

#define int\_3 3

#define int\_4 4

#define MAP\_MAX\_X 10 //地图边界，二维数组大小

#define MAP\_MAX\_Y 10

typedef struct LNode {

int data; //对应数组中的数值

int F; //F = G + H;

int G; //G：从起点 A 移动到指定方格的移动代价，沿着到达该方格而生成的路径

int H; //H：从指定的方格移动到终点 B 的估算成本

int x, y; //对应数组中的坐标

bool OPen\_flag; //在开放列表中为1，不在为0

bool Close\_flag; //在关闭列表中为1，不在为0

struct LNode\* next; //用于链表排序

struct LNode\* path\_next; //用于最终找到的路径

}LNode, \*LinkList;

LinkList InitList(); //返回一个初始化的链表

LNode\*\* malloc\_array2D(int row, int col);

void free\_array2D(LNode \*\*arr);

LNode\*\* Translate\_array(int array[10][10], int row, int col); //将一个普通数组翻译为单链表节点的数组

void output(LNode \*\*array, int row, int col);

LNode\* find\_start\_LNode(LNode\*\* Arr, int row, int col); //从数组中找到始点

LNode\* find\_end\_LNode(LNode\*\* Arr, int row, int col); //从数组中找到终点

//忘记这些要干嘛了，重写吧

bool isExist\_ALNode\_in\_List(LNode\* curLNode, LinkList L\_OpenList); //查看节点是否在链表中，在返回ture,不在返回fault

//对关闭列表中的当前节点进行检查，看它周围的节点是否在OpenList链表里，不在:添加进去；在：检查经过它到达起点的G是否最小，是：修改，不是：不修改

//LNode\* check\_CloseList\_curLNode(LNode\* curLNode, LNode\* endLNode, LinkList L\_OpenList, LinkList L\_CloseList, LNode\*\* Arr);

LNode\* pop\_OpenList\_minNode(LinkList L\_OpenList); //返回开放列表中F值最小的节点

void push\_OpenList\_Node(LinkList L, LNode \*elem); //插入一个节点并排序

bool insert\_Into\_CloseList(LNode\* min\_Open, LinkList L\_CloseList);//插入OpenList中F值最小的节点到CloseList中去

int count\_LNode\_G(LNode\* curLNode, LNode\* aheadLNode); //计算节点的G值

int count\_LNode\_H(LNode\* curLNode, LNode\* endLNode); //计算节点的H值

int count\_LNode\_F(LNode\* curLNode); //计算节点的F值

bool isExist\_openList(LNode\* curLNode); //查看节点是否在链表中，在返回ture,不在返回fault

bool isExist\_closeList(LNode\* curLNode);

bool isobstacle(LNode\* curLNode);

void check\_around\_curNode(LNode\* cur, LNode\* endLNode, LinkList open\_list, LNode\*\* Arr); //检查周围的节点，是否合适加入开放列表

**Apath.cpp**

#include "Apath.h"

LinkList InitList()

{

LinkList L = (LinkList)malloc(sizeof(LNode));

if (L == NULL)

{

printf("Defeat!");

exit(1);

}

/\*

memset函数使用方法：

将L所指向的某一块内存中的每个字节的内容全部设置为0，块的大小等于

sizeof(LNode),这个函数通常为新申请的内存做初始化工作，其返回指向L的指针

\*/

memset(L, 0, sizeof(LNode));

return L;

}//LinkList()

LNode\*\* malloc\_array2D(int row, int col)

{

LNode\*\* map = (LNode\*\*)malloc(row \* sizeof(LNode\*) + row\*col \* sizeof(LNode));

LNode\* head = (LNode\*)(map + row);

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

map[i] = head + i\*col;

return map;

}

LNode\*\* Translate\_array(int array[][10], int row, int col)

{

LNode \*\*map = malloc\_array2D(10, 10);

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

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

{

(map[i] + j)->data = array[i][j];

(map[i] + j)->G = 0;

(map[i] + j)->H = 0;

(map[i] + j)->F = 0; //(map[i] + j)->G + (map[i] + j)->H;

(map[i] + j)->x = i;

(map[i] + j)->y = j;

(map[i] + j)->Close\_flag = 0;

(map[i] + j)->OPen\_flag = 0;

(map[i] + j)->next = NULL;

(map[i] + j)->path\_next = NULL;

}

return map;

}//Translate\_array()

void free\_array2D(LNode \*\*arr)

{

free(arr);

}

void output(LNode\*\* array, int row, int col) //二维数组的访问必须指明位数，否则编译器不能解析

{

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

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

// {

// (array[i] + j)->F = j;

// }

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

{

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

{

printf("%d\t", (array[i] + j)->data);

}

printf("\n");

}

}

LNode\* find\_start\_LNode(LNode\*\* Arr, int row, int col) //从数组中找到始点

{

LNode\* start\_LNode = NULL;

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

{

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

{

if (2 == (Arr[i] + j)->data)

{

start\_LNode = (Arr[i] + j);

//起点H=0,G=0,F=0

start\_LNode->G = 0;

start\_LNode->H = 0;

start\_LNode->F = 0; //起点，则默认所有值为0

return start\_LNode; //返回节点

}

}

}

return NULL;

}

LNode\* find\_end\_LNode(LNode\*\* Arr, int row, int col) //从数组中找到终点

{

LNode\* end\_LNode = NULL;

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

{

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

{

if (3 == (Arr[i] + j)->data)

{

end\_LNode = (\*(Arr + i) + j);

end\_LNode->F = 0;

end\_LNode->G = 0;

end\_LNode->H = 0;

return end\_LNode; //返回节点

}

}

}

return NULL;

}

int count\_LNode\_G(LNode\* curLNode, LNode\* aheadLNode) //计算节点的G值

{

if (curLNode->x == aheadLNode->x && curLNode->y == aheadLNode->y)

return 0;

if (aheadLNode->x - curLNode->x != 0 && aheadLNode->y - curLNode->y != 0)

curLNode->G = aheadLNode->G + 14;

else

curLNode->G = aheadLNode->G + 10;

return curLNode->G;

}

int count\_LNode\_H(LNode\* curLNode, LNode\* endLNode) //计算节点的H值

{

curLNode->H = abs(endLNode->x - curLNode->x) \* 10 + abs(endLNode->y - curLNode->y) \* 10;

return curLNode->H;

}

int count\_LNode\_F(LNode\* curLNode) //计算节点的F值

{

curLNode->F = curLNode->G + curLNode->H;

return curLNode->F;

}

void push\_OpenList\_Node(LinkList L, LNode \*elem) //按从小到大的顺序

{

/\*

p和q的作用总结：p和q初始化都是指向链表的头结点(head)的指针,\*q的成员F值永远比\*elem的成员F值小

\*p的成员F值永远比\*elem的成员F值大，也就是说elem节点在q后p前插入，可以参考141行到142行代码

\*/

LNode \*p, \*q;

p = q = L;

while (p->next != NULL && p->F < elem->F)

{

q = p;

p = p->next;

}

if (p->F < elem->F) q = p;//这个编程规范很好，当p->next==NULL时，也是在指针q的后面插入elem节点

elem->next = q->next;

q->next = elem;

//插入成功，更改属性值OPen\_flag = 1

elem->OPen\_flag = 1;

}

LNode\* pop\_OpenList\_minNode(LinkList L\_OpenList) //返回开放列表中F值最小的节点

{

LNode \*elem = NULL;

if (L\_OpenList->next) //为了安全，防止访问空指针

{

L\_OpenList->next->OPen\_flag = 0;

elem = L\_OpenList->next;

L\_OpenList->next = L\_OpenList->next->next;

elem->next = NULL;

}

else

printf("have a NULL point in pop\_OpenList\_mimNode()");

return elem;

}

bool insert\_Into\_CloseList(LNode\* min\_Open, LinkList L\_CloseList)//插入OpenList中F值最小的节点到CloseList中去

{

//对于CloseList中的节点并不需要排序,采用头插法

min\_Open->next = L\_CloseList->next;

L\_CloseList->next = min\_Open;

min\_Open->Close\_flag = 1;

return TURE;

}

bool isExist\_openList(LNode\* curLNode)

{

return curLNode->OPen\_flag;

}

bool isExist\_closeList(LNode\* curLNode)

{

return curLNode->Close\_flag;

}

bool isobstacle(LNode\* curLNode)

{

if (curLNode->data == 1)

return TURE;

else

return FAULT;

}

bool isJoin(LNode\* cur) //该节点是否可以加入开放列表

{

if (cur->x > -1 && cur->y > -1) //边界检测

{

if (!isExist\_closeList(cur) && !isobstacle(cur)) //既不在关闭列表里，也不是障碍物

{

return TURE;

}

else

return FAULT;

}

return FAULT;

}

void insert\_open(LNode \*Node, LNode\* ahead, LNode\* endLNode, LinkList open\_list, LNode\*\* Arr)

{

if (isJoin(Node))

{

if (isExist\_openList(Node))

{

if (Node->x - ahead->x != 0 && Node->y - ahead->y != 0) {

/\*

if (Node->F > (ahead->F + 14))

{

count\_LNode\_G(Node, ahead);

count\_LNode\_F(Node); //H值没有改变，所以还是原来的值

Node->path\_next = ahead; //也不用再插入

}

\*/

if (Node->G >= (ahead->G + 14))

{

count\_LNode\_G(Node, ahead);

count\_LNode\_F(Node); //H值没有改变，所以还是原来的值

Node->path\_next = ahead; //也不用再插入

}

}

else {

/\*

if (Node->F > (ahead->F + 10))

{

count\_LNode\_G(Node, ahead);

count\_LNode\_F(Node); //H值没有改变，所以还是原来的值

Node->path\_next = ahead; //也不用再插入

}

\*/

if (Node->G >= (ahead->G + 10))

{

count\_LNode\_G(Node, ahead);

count\_LNode\_F(Node); //H值没有改变，所以还是原来的值

Node->path\_next = ahead; //也不用再插入

}

}

}

else {

count\_LNode\_G(Node, ahead);

count\_LNode\_H(Node, endLNode);

count\_LNode\_F(Node);

Node->path\_next = ahead;

push\_OpenList\_Node(open\_list, Node);

}

}

}

void check\_around\_curNode(LNode\* cur, LNode\* endLNode, LinkList open\_list, LNode\*\* Arr)

{

/\*

关于为什么加入x与y的限制，因为访问数组Arr是通过指针访问的。当x==5,y==0时，Arr[x]+y-1等价于Arr[4][9]

\*/

int x = cur->x;

int y = cur->y;

if (y > 0) { // y最小取1

insert\_open(Arr[x] + y - 1, cur, endLNode, open\_list, Arr);

}

if (y < MAP\_MAX\_Y - 1) { // y最大取8

insert\_open(Arr[x] + y + 1, cur, endLNode, open\_list, Arr);

}

if (x < MAP\_MAX\_X - 1) { // x最大取8

insert\_open(Arr[x + 1] + y, cur, endLNode, open\_list, Arr);

}

if (x < MAP\_MAX\_X - 1 && y > 0) {

insert\_open(Arr[x + 1] + y - 1, cur, endLNode, open\_list, Arr);

}

if (x < MAP\_MAX\_X - 1 && y < MAP\_MAX\_Y - 1) {

insert\_open(Arr[x + 1] + y + 1, cur, endLNode, open\_list, Arr);

}

if (x > 0) {

insert\_open(Arr[x - 1] + y, cur, endLNode, open\_list, Arr);

}

if (x > 0 && y < MAP\_MAX\_Y - 1) {

insert\_open(Arr[x - 1] + y + 1, cur, endLNode, open\_list, Arr);

}

if (x > 0 && y > 0) {

insert\_open(Arr[x - 1] + y - 1, cur, endLNode, open\_list, Arr);

}

}