迷宫问题

#include "1.h"

A\_Star::A\_Star(const char pImportMap[RowNumber][ColumnNumber])

{

if (!initMap(pImportMap))

{

exit(0);

}

}

A\_Star::~A\_Star()

{

}

bool A\_Star::initMap( const char pImportMap[RowNumber][ColumnNumber] )

{

bool isHaveStart,isHaveEnd;

isHaveStart = isHaveEnd = false;//检测是否有起点和终点

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

{

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

{

m\_StarNode[i][j].m\_cSymbol = pImportMap[i][j];

m\_StarNode[i][j].m\_X=i;

m\_StarNode[i][j].m\_Y=j;

if(m\_StarNode[i][j].m\_cSymbol=='s')

{

m\_StarNode[i][j].m\_Type = Node\_Start;

m\_StarNode[i][j].m\_Gvalue=0;

m\_StarNode[i][j].m\_IsFind=true;

isHaveStart = true;

m\_StarStart=m\_StarNode[i][j];

}

else if(m\_StarNode[i][j].m\_cSymbol=='e')

{

m\_StarNode[i][j].m\_Type = Node\_End;

m\_StartEnd=m\_StarNode[i][j];

isHaveEnd = true;

}

else if(m\_StarNode[i][j].m\_cSymbol=='x')

{

m\_StarNode[i][j].m\_Type = Node\_Barrier;

}

else

{

m\_StarNode[i][j].m\_Type = Node\_Way;

}

}

}

if (!isHaveStart)

{

cout<<"Where will We Start"<<endl;

}

else if(!isHaveEnd)

{

cout<<"Where will We Go ?"<<endl;

}

return (isHaveStart&&isHaveEnd);

}

bool A\_Star::isOverBorder(int nX,int nY)

{

if (nX < 0 || nX >= RowNumber || nY < 0 || nY >= ColumnNumber)

{

return true;

}

return false;

}

//曼哈顿距离为：|x1 - x2| + |y1 - y2|

float A\_Star::manhattanDis(int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y)

{

float hVal;

hVal=(float)abs(nDisStart\_x-nDisEnd\_x)+abs(nDisStart\_y-nDisEnd\_y);

hVal\*=ManhattanSacle;

return hVal;

}

//加权曼哈顿 better

float A\_Star::jiaquan\_manhattanDis( int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y )

{

float hVal,dx,dy;

dx=(float)abs(nDisStart\_x-nDisEnd\_x);

dy=(float)abs(nDisStart\_y-nDisEnd\_y);

if(dx>dy)

hVal=10\*dx+6\*dy;

else

hVal=6\*dx+10\*dy;

return hVal;

}

//欧氏距离为： sqrt((x1-x2)^2+(y1-y2)^2 )

float A\_Star::euclideanDis( int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y )

{

float hVal;

hVal=(float)sqrt(pow((nDisStart\_x-nDisEnd\_x),2)+pow((nDisStart\_y-nDisEnd\_y),2));

return hVal;

}

//切比雪夫距离为：max(|x1 - x2| , |y1 - y2|)

float A\_Star::chebyshevDis( int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y )

{

float hVal;

hVal=(float)max(abs(nDisStart\_x-nDisEnd\_x),abs(nDisStart\_y-nDisEnd\_y));

return hVal;

}

void A\_Star::AstarAlgorithm()

{

priority\_queue<myNode> openQueue;//open表

openQueue.push(m\_StarStart);//将起始点压入open表

myNode currNode,\*neighborNode;

int nNeighborX,nNeighborY; //邻居的x，y坐标, 邻居相对于选定点的G值偏移

float offsetG;

while(!openQueue.empty())//oepn表不为空一直循环

{

currNode = openQueue.top();//每次有元素插入都会进行排序，myNode 重载了操作运算符 < ，每次队列头的元素的F值最小

if (currNode == m\_StartEnd)//当前节点为终点，退出

{

break;

}

openQueue.pop();//出队列

// printRoute();

for (int i=0;i<NeighborNum;++i)//遍历该节点的邻居

{

//根据所有偏移量找到邻居

nNeighborX = currNode.m\_X + offset[i][0];

nNeighborY = currNode.m\_Y + offset[i][1];

//邻居节点超出边界，障碍物，查找过了都跳过

if (isOverBorder(nNeighborX,nNeighborY))

{

continue;

}

neighborNode = &m\_StarNode[nNeighborX][nNeighborY];//找到邻居节点的信息

if (neighborNode->m\_IsFind || neighborNode->m\_Type == Node\_Barrier)

{

continue;

}

//这里 可以替换不同的算法，选择想用的h(x)

neighborNode->m\_Hvalue = jiaquan\_manhattanDis(nNeighborX,nNeighborY,m\_StartEnd.m\_X,m\_StartEnd.m\_Y);

offsetG = 1.0f;//不是对角线的邻居，默认 1 个距离，相当于一个格子

if (offset[i][0]!=0 && offset[i][1]!=0)//说明是对对角线上的邻居

{

offsetG = 1.4f;//取近似值,利于计算,1.4个格子

}

//还没压入open表 或者

//如果该邻居之前的G值(G值是重新选择父亲节点前计算得到的)小于从新计算的G值(以当前点为父节点计算的)

//该邻居节点的父节点要做更改，指向当前节点作为父节点, 因为这个估值更小，代价更低

if(neighborNode->m\_Gvalue == 0 || neighborNode->m\_Gvalue>currNode.m\_Gvalue+offsetG)

{

neighborNode->m\_ParentNode = &m\_StarNode[currNode.m\_X][currNode.m\_Y];

neighborNode->m\_Gvalue = currNode.m\_Gvalue+offsetG;

}

neighborNode->m\_IsFind = true;//被查找过了

openQueue.push(\*neighborNode);

}

}

}

bool A\_Star::findRoute()

{

myNode\* pNode = m\_StarNode[m\_StartEnd.m\_X][m\_StartEnd.m\_Y].m\_ParentNode;//最后节点，非终点

if (pNode == NULL)

{

cout<<"坑爹啊？根本没有路径可以到达终点"<<endl;

return false;

}

//根据终点，一个个往回找

while (pNode)

{

if (pNode->m\_ParentNode)

{

pNode->m\_Type = Node\_FindWay;

}

pNode = pNode->m\_ParentNode;

}

return true;

}

void A\_Star::printRoute()

{

int nFindNum = 0;//统计遍历的节点数

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

{

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

{

if (m\_StarNode[i][j].m\_IsFind)//统计遍历的节点数

{

nFindNum++;

}

if(m\_StarNode[i][j].m\_Type==Node\_FindWay)//找到的路

cout<<FindWay;//<<"F:"<<m\_StarNode[i][j].m\_Gvalue+m\_StarNode[i][j].m\_Hvalue;

else if(m\_StarNode[i][j].m\_IsFind)//找过的地方

{

if (m\_StarNode[i][j].m\_Type == Node\_Way)

{

cout<<HaveFindPlace;//<<"F:"<<m\_StarNode[i][j].m\_Gvalue+m\_StarNode[i][j].m\_Hvalue;

}

else

{

cout<<m\_StarNode[i][j].m\_cSymbol;

}

}

else

cout<<m\_StarNode[i][j].m\_cSymbol;

}

cout<<endl;

}

cout<<"共找了 "<<nFindNum<<" 个节点，终于找到了路"<<endl;

}

#pragma once

#include <stdio.h>

#include <iostream>

#include <queue>

#include <math.h>

#include <stdlib.h>

using namespace std;

//定义是否可以走对角线的宏

#define WALK\_DIAGONAL

#ifdef WALK\_DIAGONAL

#define NeighborNum 8 //邻居个数

static int offset[NeighborNum][2]={{1,0},{1,1},{1,-1},{-1,0},{-1,-1},{-1,1},{0,-1},{0,1}};//邻居的坐标选择

#else

#define NeighborNum 4 //邻居个数

static int offset[NeighborNum][2]={{1,0},{-1,0},{0,-1},{0,1}};//邻居的坐标选择

#endif

//定义二维数组的长度

#define RowNumber 6

#define ColumnNumber 8

//定义曼哈顿距离的扩大比例

#define ManhattanSacle 10 //10倍

enum NodeType

{

Node\_Start = 0, //起始点

Node\_Barrier, //障碍物，把所有不能通过的点都认为是障碍物

Node\_Way, //可通行的路

Node\_FindWay, //经过寻路后，会经过的点

Node\_End, //终点

Node\_Null, //未定义的点

};

//定义找到的路径的符号表示

#define FindWay '\*' //最终找到的路径的符号表示

#define HaveFindPlace 'o'//找过的地方的符号表示

//定义一个节点的数据结构

struct myNode

{

myNode()

{

m\_ParentNode = NULL;

m\_Type = Node\_Null;

m\_X = m\_Y = 0;

m\_Gvalue = 0;

m\_Hvalue = 0;

m\_ParentNode = NULL;

m\_IsFind = false;

}

char m\_cSymbol; //该节点的符号

int m\_Type; //该节点的类型

int m\_X; //节点的X坐标

int m\_Y; //节点的Y坐标

float m\_Gvalue; //g(x) G值

float m\_Hvalue; //h(x) H值

myNode\* m\_ParentNode; //该节点指向的父节点

bool m\_IsFind; //该节点是否被找过

//按F值（= G + H） 从小到大排序，F值小的优先级高

friend bool operator<(myNode nNodeA,myNode nNodeB)

{

return (nNodeA.m\_Gvalue+nNodeA.m\_Hvalue > nNodeB.m\_Gvalue+nNodeB.m\_Hvalue);

}

//重载操作符 ==

friend bool operator==(myNode nNodeA,myNode nNodeB)

{

return (nNodeA.m\_X == nNodeB.m\_X && nNodeA.m\_Y == nNodeB.m\_Y);

}

};

class A\_Star

{

public:

A\_Star(const char pImportMap[RowNumber][ColumnNumber]);

~A\_Star();

protected:

//初始化地图

bool initMap(const char pImportMap[RowNumber][ColumnNumber]);

//曼哈顿距离为：|x1 - x2| + |y1 - y2|

float manhattanDis(int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y);

//加权曼哈顿距离 better 不大清楚为什么用这个

float jiaquan\_manhattanDis(int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y);

//欧氏距离为： sqrt((x1-x2)^2+(y1-y2)^2 )

float euclideanDis(int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y);

//切比雪夫距离为：max(|x1 - x2| , |y1 - y2|)

float chebyshevDis(int nDisStart\_x,int nDisStart\_y,int nDisEnd\_x,int nDisEnd\_y);

//大小比较函数

int max(int a,int b)

{

return a>b?a:b;

}

//对外接口

public:

//判断节点是否超过边界

//return true 超过边界，false：未超过

bool isOverBorder(int nX,int nY);

//A\*核心算法

void AstarAlgorithm();

//找到路线路线

bool findRoute();

//打印出路线

void printRoute();

private:

myNode m\_StarNode[RowNumber][ColumnNumber]; //存储所有节点数

myNode m\_StarStart; //开始节点

myNode m\_StartEnd; //终点节点

};

#include "1.h"

int main()

{

//自定义一个地图 （RowNumber \* ColumnNumber）大小

//'s':起始点字符

//'e':终点字符

//'x':障碍物字符

//'.':可通过点

char chMapDraw[RowNumber][ColumnNumber] =

{

{ '.','.','.','x','x','x','.','.'},

{ '.','.','.','.','.','x','x','.'},

{ '.','x','.','x','.','x','.','e'},

{ '.','x','.','x','.','x','.','.'},

{ 's','x','.','.','.','x','.','x'},

{ '.','x','.','.','.','.','.','.'},

};

//初始化地图

A\_Star Astar(chMapDraw);

//A\*算法

Astar.AstarAlgorithm();

//找到路径

if (Astar.findRoute())

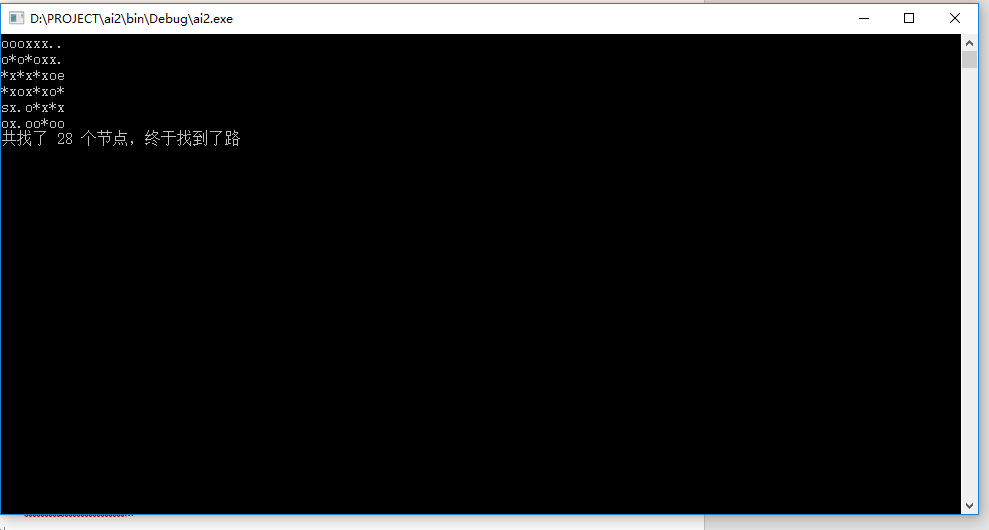
{

Astar.printRoute();//打印路径

}

getchar();

}



八数码问题

#include<stdio.h>

#include<stdlib.h>

#define TIME 100 //只搜索100次

#define MAXSIZE 100

int n=1;

int result[9]={1,2,3,8,0,4,7,6,5};//0是空格

typedef struct Node{

int num[9];

char expension;//记录是否可以拓展，Y可以，N不可以

char banOperate;/\*表示不能执行的操作，L代表不能左移，

R代表不能右移，U代表不能上移，D代表不能下移，

C代表可以随意移动\*/

int father;//父节点的下标

}Node;

Node store[MAXSIZE];//将搜索过的状态存储于数组中

int same(int temp)//判断是否达到了目标状态

{

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

if(store[temp].num[j]!=result[j])

return 0;

return 1;

}

void printResult()//输出搜索结果

{

for(int j=1;j<=n;j++)

{

printf("第%d步搜索后:\n",j);

printf("\t%d\t%d\t%d\n",

store[j].num[0],store[j].num[1],store[j].num[2]);

printf("\t%d\t%d\t%d\n",

store[j].num[3],store[j].num[4],store[j].num[5]);

printf("\t%d\t%d\t%d\n",

store[j].num[6],store[j].num[7],store[j].num[8]);

printf("\n\n");

}

}

int left(int temp)//将空格左移

{

int o;

for(int j=0;j<9;j++)//判断空格位置

{

if(store[temp].num[j]==0)

o=j;

break;

}

if(o==0||o==3||o==6)

return 0;

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

store[n].num[k]=store[temp].num[k];

int tempNum=store[n].num[o-1];//移动后的状态

store[n].num[o-1]=0;

store[n].num[o]=tempNum;

store[temp].expension='N';

store[n].banOperate='L';

store[n].expension='Y';

store[n].father=temp;

if(same(n))

{

printResult();

exit(1);

}

n++;

return 1;

}

int right(int temp)//将空格右移

{

int o;

for(int j=0;j<9;j++)//判断空格位置

{

if(store[temp].num[j]==0)

o=j;

break;

}

if(o==2||o==5||o==8)

return 0;

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

store[n].num[k]=store[temp].num[k];

int tempNum=store[n].num[o+1];//移动后的状态

store[n].num[o+1]=0;

store[n].num[o]=tempNum;

store[temp].expension='N';

store[n].banOperate='R';

store[n].expension='Y';

store[n].father=temp;

if(same(n))

{

printResult();

exit(1);

}

n++;

return 1;

}

int up(int temp)//将空格上移

{

int o;

for(int j=0;j<9;j++)//判断空格位置

{

if(store[temp].num[j]==0)

o=j;

break;

}

if(o==0||o==1||o==2)

return 0;

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

store[n].num[k]=store[temp].num[k];

int tempNum=store[n].num[o-3];//移动后的状态

store[n].num[o-3]=0;

store[n].num[o]=tempNum;

store[temp].expension='N';

store[n].banOperate='U';

store[n].expension='Y';

store[n].father=temp;

if(same(n))

{

printResult();

exit(1);

}

n++;

return 1;

}

int down(int temp)//将空格下移

{

int o;

for(int j=0;j<9;j++)//判断空格位置

{

if(store[temp].num[j]==0)

o=j;

break;

}

if(o==6||o==7||o==8)

return 0;

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

store[n].num[k]=store[temp].num[k];

int tempNum=store[n].num[o+3];//移动后的状态

store[n].num[o+3]=0;

store[n].num[o]=tempNum;

store[temp].expension='N';

store[n].banOperate='D';

store[n].expension='Y';

store[n].father=temp;

if(same(n))

{

printResult();

exit(1);

}

n++;

return 1;

}

void init()

{

Node start;

printf("请输入初始状态，用空格分开，0代表空格:\n");//输入初始状态

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

scanf("%d",&start.num[i]);

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

if(start.num[k]==0)

break;

start.banOperate='C';

start.expension='Y';

start.father=-1;

store[0]=start;

}//将初始状态赋予store[0]

int main()

{

init();

if(same(0))

{

printf("没有必要进行搜索");

exit(1);

}

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

{

if(store[i].expension=='Y')

{

if(store[i].banOperate=='L')

{

up(i);

right(i);

down(i);

}

if(store[i].banOperate=='R')

{

left(i);

up(i);

down(i);

}

if(store[i].banOperate=='U')

{

left(i);

right(i);

down(i);

}

if(store[i].banOperate=='D')

{

left(i);

up(i);

right(i);

}

if(store[i].banOperate=='C')

{

left(i);

up(i);

right(i);

down(i);

}

}

if(n>=TIME)

{

n--;

printResult();

printf("在搜索范围内没有结果");

exit(1);

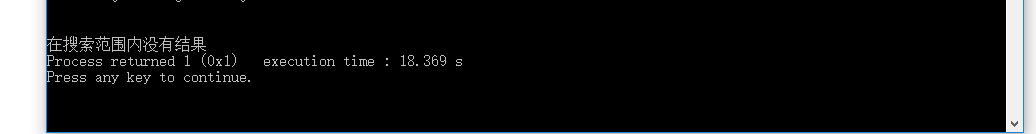
}

}

return 0;

}

\*\*\*未能搜索出结果，换了一种方式，上种方式待改进\*\*\*



#include <iostream>

#include <queue>

#include <stack>

#include <vector>

#include <algorithm>

#include <memory.h>

using namespace std;

// 八数码状态

typedef struct \_Status{

int status[3][3];

\_Status \*parent;

\_Status \*next;

}Status;

// AStar排序依据

bool decComparator(const Status &s1, const Status &s2){

int gn1 = 0, gn2 = 0;

int dn1 = 0, dn2 = 0;

const Status \*ptr1 = &s1;

const Status \*ptr2 = &s2;

int status[3][3] = {1,2,3,8,0,4,7,6,5};

while(ptr1 != NULL){

gn1 += 1;

ptr1 = ptr1->parent;

}

while(ptr2 != NULL){

gn2 += 1;

ptr2 = ptr2->parent;

}

for(int i = 0; i < 3; i++){

for(int j = 0; j < 3; j++){

if(s1.status[i][j] != status[i][j]){

dn1 += 1;

}

if(s2.status[i][j] != status[i][j]){

dn2 += 1;

}

}

}

return (gn1+dn1) > (gn2+dn2);

}

// 八数码搜索

class EightPuzzle{

private:

unsigned char allHash[362880];

Status root;

Status goal;

private:

int nextNumber;

Status next[4];

public:

EightPuzzle(Status \*root, Status \*goal){

memcpy(&this->root.status, &root->status, sizeof(int)\*9);

this->root.parent = NULL;

this->root.next = NULL;

memcpy(&this->goal.status, &goal->status, sizeof(int)\*9);

this->goal.parent = NULL;

this->goal.next = NULL;

}

private:

// 判断是否是目标状态

inline int IsGoal(Status \*tmp){

return memcmp(&tmp->status, &goal.status, sizeof(int)\*9);

}

// 下一个可行的状态

int NextStatus(Status \*tmp){

nextNumber = 0;

int posi, posj;

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

posi = i/3, posj = i - i/3\*3;

if(tmp->status[posi][posj] == 0){

break;

}

}

if(posi-1 >= 0){

Status left = \*tmp;

left.status[posi][posj] = left.status[posi-1][posj];

left.status[posi-1][posj] = 0;

if(allHash[Cantor(left.status)] == 0){

next[nextNumber] = left;

next[nextNumber].parent = tmp;

nextNumber++;

}

}

if(posi+1 <= 2){

Status right = \*tmp;

right.status[posi][posj] = right.status[posi+1][posj];

right.status[posi+1][posj] = 0;

if(allHash[Cantor(right.status)] == 0){

next[nextNumber] = right;

next[nextNumber].parent = tmp;

nextNumber++;

}

}

if(posj-1 >= 0){

Status up = \*tmp;

up.status[posi][posj] = up.status[posi][posj-1];

up.status[posi][posj-1] = 0;

if(allHash[Cantor(up.status)] == 0){

next[nextNumber] = up;

next[nextNumber].parent = tmp;

nextNumber++;

}

}

if(posj+1 <= 2){

Status down = \*tmp;

down.status[posi][posj] = down.status[posi][posj+1];

down.status[posi][posj+1] = 0;

if(allHash[Cantor(down.status)] == 0){

next[nextNumber] = down;

next[nextNumber].parent = tmp;

nextNumber++;

}

}

return nextNumber;

}

// 康托展开

int Cantor(int arr[][3]){

int fac[10] = {1,1,2,6,24,120,720,5040,40320,362880};

int index = 0;

for(int i = 7; i >= 0; i--){

int irow = i/3, icol = i - i/3\*3;

int count = 0;

for(int j = 8; j > i; j--){

int jrow = j/3, jcol = j - j/3\*3;

if(arr[jrow][jcol] < arr[irow][icol]){

count++;

}

}

index += (count\*fac[8-i]);

}

return index;

}

public:

// 广度优先搜索

int BFS(){

int step = 0;

memset(allHash, 0, 362880);

queue<Status> openTable;

Status \*closeTable = new Status;;

Status \*current = closeTable;

Status \*tmp;

openTable.push(root);

allHash[Cantor(root.status)] == 1;

while(!openTable.empty()){

tmp = new Status;

\*tmp = openTable.front();

openTable.pop();

step++;

current->next = tmp;

current = current->next;

if(IsGoal(tmp) == 0){

PrintPath(tmp);

freeCloseTable(closeTable);

return step;

}

int nextNumber = NextStatus(tmp);

if(nextNumber == 0){

continue;

}

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

openTable.push(next[i]);

allHash[Cantor(next[i].status)] == 1;

}

}

cout << "BFS failed." << endl;

freeCloseTable(closeTable);

return -1;

}

// 深度优先搜索

int DFS(){

int depth = 0;

int step = 0;

stack<Status> openTable;

Status \*closeTable = new Status;;

Status \*current = closeTable;

Status \*last;

Status \*tmp;

openTable.push(root);

while(!openTable.empty()){

tmp = new Status;

\*tmp = openTable.top();

openTable.pop();

step++;

current->next = tmp;

current = current->next;

if(IsGoal(tmp) == 0){

PrintPath(tmp);

freeCloseTable(closeTable);

return step;

}

memset(allHash, 0, 362880);

last = tmp;

depth = 0;

while(last != NULL){

allHash[Cantor(last->status)] = 1;

last = last->parent;

depth++;

}

if(depth > 8){

continue;

}

int nextNumber = NextStatus(tmp);

if(nextNumber == 0){

continue;

}

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

openTable.push(next[i]);

}

}

cout << "DFS failed." << endl;

freeCloseTable(closeTable);

return -1;

}

// 启发式搜索

int AStar(){

int step = 0;

memset(allHash, 0, 362880);

vector<Status> openTable;

Status \*closeTable = new Status;;

Status \*current = closeTable;

Status \*tmp;

openTable.push\_back(root);

allHash[Cantor(root.status)] == 1;

while(!openTable.empty()){

tmp = new Status;

\*tmp = openTable[openTable.size()-1];

openTable.pop\_back();

step++;

current->next = tmp;

current = current->next;

if(IsGoal(tmp) == 0){

PrintPath(tmp);

freeCloseTable(closeTable);

return step;

}

int nextNumber = NextStatus(tmp);

if(nextNumber == 0){

continue;

}

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

openTable.push\_back(next[i]);

allHash[Cantor(next[i].status)] == 1;

}

sort(openTable.begin(), openTable.end(), decComparator);

}

cout << "AStar failed." << endl;

freeCloseTable(closeTable);

return -1;

}

private:

// 打印路径

void PrintPath(Status \*head){

if(head == NULL){

return;

}

else{

PrintPath(head->parent);

for(int i = 0; i < 3; i++){

for(int j = 0; j < 3; j++){

cout << head->status[i][j];

}

cout << endl;

}

cout <<endl;

}

}

// 释放close表

void freeCloseTable(Status \*closeTable){

Status \*current;

while(closeTable != NULL){

current = closeTable->next;

free(closeTable);

closeTable = current;

}

}

};

int main()

{

Status init = {2,8,3,1,6,4,7,0,5,0,NULL};

Status goal = {1,2,3,8,0,4,7,6,5,0,NULL};

EightPuzzle ep = EightPuzzle(&init, &goal);

cout << "BFS\*\*\*\*\*\*\*\*\n" << endl;

cout << "step: " << ep.BFS() << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\n" << endl;

cout << "DFS\*\*\*\*\*\*\*\*\n" << endl;

cout << "step: " << ep.DFS() << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\n" << endl;

cout << "AStar\*\*\*\*\*\*\n" << endl;

cout << "step: " << ep.AStar() << endl;

cout << "\*\*\*\*\*\*\*\*\*\*\*\n" << endl;

return 0;

}

