八数码问题

这里需要说明的A能找到的解是局部最优解，但是独特的启发式函数可以使得解为全局最优解，八数码问题就是一个能通过A求得最优解的问题。

像下图所示，通过将数字位向空格位移动直至将棋盘从初始状态变化到目标状态。

问题分析

启发式函数的确定

h(n):已经移动的步数

g(n):此状态与目标状态九宫格中相异数字的个数

状态保存

A\*算法有个很大的问题就是消耗内存资源，我们可以用char型数据保存，这里我另一种保存策略：用一个long int数值表示，方法如下

0-8九个状态可以四位二进制数来表示0000B-1000B,所以九个状态就可以用36个二进制位来表示，然后这36位二进制数就可以用一个long int型数据来表示，这样增加编码和解码工作，不过操作很风骚，位运算很好实现，只是这是后来想到的，没有实现

算法优化

在找最小值的时候，我们可以用二分查找，o(n)优化到o(logn)，这就要求我们再插入时顺序插入，因为查询次数是要大于添加open\close表项的，所以这个方法是可以优化执行效率的

无解情况

将九宫格变成线性后，计算初始状态和目标状态的奇偶性是否一致，一致有解，否则无解。

代码实现：

#include <iostream>

#include <ctime>

#include <vector>

using namespace std;

const int ROW = 3;

const int COL = 3;

const int MAXDISTANCE = 10000;

const int MAXNUM = 10000;

typedef struct \_Node{

int digit[ROW][COL];

int dist; // distance between one state and the destination

int dep; // the depth of node

// So the comment function = dist + dep.

int index; // point to the location of parent

} Node;

Node src, dest;

vector<Node> node\_v; // store the nodes

bool isEmptyOfOPEN() {

for (int i = 0; i < node\_v.size(); i++) {

if (node\_v[i].dist != MAXNUM)

return false;

}

return true;

}

bool isEqual(int index, int digit[][COL]) {

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

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

if (node\_v[index].digit[i][j] != digit[i][j])

return false;

}

return true;

}

ostream& operator<<(ostream& os, Node& node) {

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

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

os << node.digit[i][j] << ' ';

os << endl;

}

return os;

}

void PrintSteps(int index, vector<Node>& rstep\_v) {

rstep\_v.push\_back(node\_v[index]);

index = node\_v[index].index;

while (index != 0) {

rstep\_v.push\_back(node\_v[index]);

index = node\_v[index].index;

}

for (int i = rstep\_v.size() - 1; i >= 0; i--)

cout << "Step " << rstep\_v.size() - i

<< endl << rstep\_v[i] << endl;

}

void Swap(int& a, int& b) {

int t;

t = a;

a = b;

b = t;

}

void Assign(Node& node, int index) {

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

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

node.digit[i][j] = node\_v[index].digit[i][j];

}

int GetMinNode() {

int dist = MAXNUM;

int loc; // the location of minimize node

for (int i = 0; i < node\_v.size(); i++) {

if (node\_v[i].dist == MAXNUM)

continue;

else if ((node\_v[i].dist + node\_v[i].dep) < dist) {

loc = i;

dist = node\_v[i].dist + node\_v[i].dep;

}

}

return loc;

}

bool isExpandable(Node& node) {

for (int i = 0; i < node\_v.size(); i++) {

if (isEqual(i, node.digit))

return false;

}

return true;

}

int Distance(Node& node, int digit[][COL]) {

int distance = 0;

bool flag = false;

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

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

for (int k = 0; k < ROW; k++) {

for (int l = 0; l < COL; l++) {

if (node.digit[i][j] == digit[k][l]) {

distance += abs(i - k) + abs(j - l);

flag = true;

break;

}

else

flag = false;

}

if (flag)

break;

}

return distance;

}

int MinDistance(int a, int b) {

return (a < b ? a : b);

}

void ProcessNode(int index) {

int x, y;

bool flag;

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

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

if (node\_v[index].digit[i][j] == 0) {

x =i; y = j;

flag = true;

break;

}

else flag = false;

}

if(flag)

break;

}

Node node\_up;

Assign(node\_up, index);

int dist\_up = MAXDISTANCE;

if (x > 0) {

Swap(node\_up.digit[x][y], node\_up.digit[x - 1][y]);

if (isExpandable(node\_up)) {

dist\_up = Distance(node\_up, dest.digit);

node\_up.index = index;

node\_up.dist = dist\_up;

node\_up.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_up);

}

}

Node node\_down;

Assign(node\_down, index);

int dist\_down = MAXDISTANCE;

if (x < 2) {

Swap(node\_down.digit[x][y], node\_down.digit[x + 1][y]);

if (isExpandable(node\_down)) {

dist\_down = Distance(node\_down, dest.digit);

node\_down.index = index;

node\_down.dist = dist\_down;

node\_down.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_down);

}

}

Node node\_left;

Assign(node\_left, index);

int dist\_left = MAXDISTANCE;

if (y > 0) {

Swap(node\_left.digit[x][y], node\_left.digit[x][y - 1]);

if (isExpandable(node\_left)) {

dist\_left = Distance(node\_left, dest.digit);

node\_left.index = index;

node\_left.dist = dist\_left;

node\_left.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_left);

}

}

Node node\_right;

Assign(node\_right, index);

int dist\_right = MAXDISTANCE;

if (y < 2) {

Swap(node\_right.digit[x][y], node\_right.digit[x][y + 1]);

if (isExpandable(node\_right)) {

dist\_right = Distance(node\_right, dest.digit);

node\_right.index = index;

node\_right.dist = dist\_right;

node\_right.dep = node\_v[index].dep + 1;

node\_v.push\_back(node\_right);

}

}

node\_v[index].dist = MAXNUM;

}

int main() {

int number;

cout << "Input source:" << endl;

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

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

cin >> number;

src.digit[i][j] = number;

}

src.index = 0;

src.dep = 1;

cout << "Input destination:" << endl;

for (int m = 0; m < ROW; m++)

for (int n = 0; n < COL; n++) {

cin >> number;

dest.digit[m][n] = number;

}

node\_v.push\_back(src);

cout << "Search..." << endl;

clock\_t start = clock();

while (1) {

if (isEmptyOfOPEN()) {

cout << "Cann't solve this statement!" << endl;

return -1;

}

else {

int loc; // the location of the minimize node

loc = GetMinNode();

if(isEqual(loc, dest.digit)) {

vector<Node> rstep\_v;

cout << "Source:" << endl;

cout << src << endl;

PrintSteps(loc, rstep\_v);

cout << "Successful!" << endl;

cout << "Using " << (clock() - start) / CLOCKS\_PER\_SEC

<< " seconds." << endl;

break;

}

else

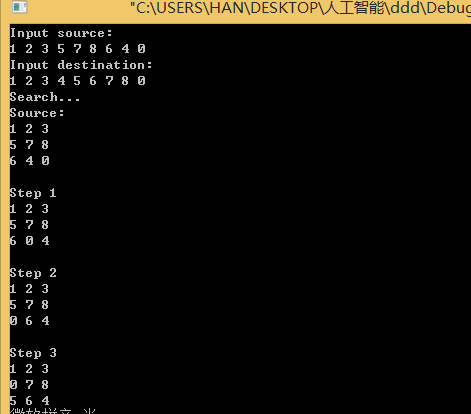
ProcessNode(loc);

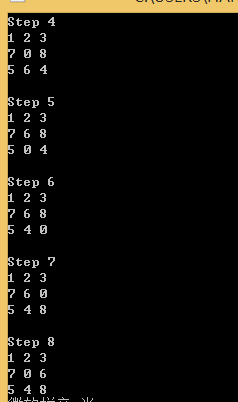
}

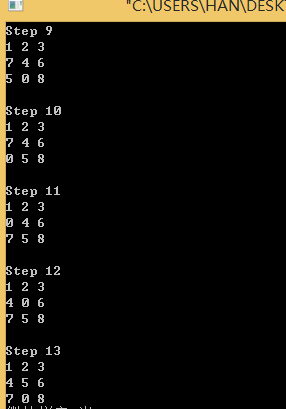
}

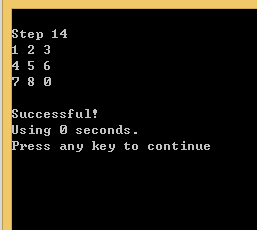
return 0;

}









程序二：

#include <stdio.h>  
  
#include <malloc.h>  
  
#include <stdlib.h>  
  
#define N 3                                //数码组大小  
  
#define Max\_Step 30                        //最大搜索深度  
  
#define MAX 50  
  
typedef struct node                        //八数码结构体  
  
{  
  
int form[N][N];                        //数码组  
  
int evalue;                            //评估值，差距  
  
int udirec;                    //所屏蔽方向,防止往回推到上一状态,1上2下3左4右  
  
struct node \*parent;                    //父节点  
  
}Graph;  
  
Graph \*Qu[MAX];                            //队列  
  
Graph \*St[MAX];                            //堆栈  
  
/\*打印数码组\*/  
  
void Print(Graph \*The\_graph)  
  
{  
  
int i,j;  
  
if(The\_graph==NULL)  
  
printf("图为空\n");  
  
else  
  
{  
  
printf("-----------------------------------------\n");  
  
for(i=0;i<N;i++)  
  
{  
  
printf("|\t");  
  
for(j=0;j<N;j++)  
  
{  
  
printf("%d\t",The\_graph->form[i][j]);      //遍历打印  
  
}  
  
printf("\t|\n");  
  
}  
  
printf("|\t\t\t\t差距:%d\t|\n",The\_graph->evalue);  //差距显示  
  
printf("-----------------------------------------\n");  
  
}  
  
}  
  
/\*估价函数\*/  
  
int Evaluate(Graph \* The\_graph,Graph \* End\_graph)  
  
{  
  
int value=0;    //差距数  
  
int i,j;  
  
for(i=0;i<N;i++)  
  
{  
  
for(j=0;j<N;j++)  
  
{  
  
if(The\_graph->form[i][j]!=End\_graph->form[i][j])  //遍历数码组比较差距  
  
value++;  
  
}  
  
}  
  
The\_graph->evalue=value;  
  
return value;  
  
}  
  
/\*移动数码组\*/  
  
Graph \*Move(Graph \*The\_graph,int Direct,int CreateNew\_graph)  
  
{  
  
Graph \*New\_graph;  
  
int HasGetBlank=0;              //是否获取空格位置  
  
int AbleMove=1;                //是否可移动  
  
int i,j,t\_i,t\_j,x,y;  
  
for(i=0;i<N;i++)                //获取空闲格位置  
  
{         
  
for(j=0;j<N;j++)  
  
{  
  
if(The\_graph->form[i][j]==0)  
  
{  
  
HasGetBlank=1;  
  
break;  
  
}  
  
}  
  
if(HasGetBlank)  
  
break;  
  
}  
  
//printf("空格位置:%d,%d\n",i,j);  
  
t\_i=i;t\_j=j;  
  
switch(Direct)  //移动空格  
  
{  
  
case 1:    //上  
  
t\_i--;  
  
if(t\_i<0)  
  
AbleMove=0;  
  
break;         
  
case 2:    //下  
  
t\_i++;  
  
if(t\_i>=N)  
  
AbleMove=0;  
  
break;         
  
case 3:    //左  
  
t\_j--;  
  
if(t\_j<0)  
  
AbleMove=0;  
  
break;       
  
case 4:    //右  
  
t\_j++;  
  
if(t\_j>=N)  
  
AbleMove=0;  
  
break;       
  
};  
  
if(!AbleMove)  //不可以移动则返回原节点  
  
{  
  
return The\_graph;  
  
}  
  
if(CreateNew\_graph)  
  
{  
  
New\_graph=(Graph \*)malloc(sizeof(Graph));        //生成节点  
  
for(x=0;x<N;x++)  
  
{  
  
for(y=0;y<N;y++)  
  
New\_graph->form[x][y]=The\_graph->form[x][y]; //复制数码组  
  
}  
  
}  
  
else  
  
{  
  
New\_graph=The\_graph;  
  
}  
  
/\*移动后\*/  
  
New\_graph->form[i][j]=New\_graph->form[t\_i][t\_j];  
  
New\_graph->form[t\_i][t\_j]=0;       
  
return New\_graph;  
  
}  
  
/\*搜索函数\*/  
  
Graph \*Search(Graph \*Begin,Graph \*End)  
  
{  
  
Graph \*g1,\*g2,\*g;  
  
int Step=0;          //深度  
  
int Direct=0;  
  
int i;  
  
int front,rear;  
  
front=rear=-1;      //队列初始化  
  
g=NULL;  
  
rear++;              //入队  
  
Qu[rear]=Begin;  
  
while(rear!=front)  //队不空

{  
  
front++;//出队  
  
g1=Qu[front];  
  
//printf("开始第%d个图:\n",front);  
  
//Print(g1);  
  
for(i=1;i<=4;i++)              //分别从四个方向推导出新子节点  
  
{  
  
Direct=i;  
  
if(Direct==g1->udirec)    //跳过屏蔽方向  
  
continue;         
  
g2=Move(g1,Direct,1);      //移动数码  
  
if(g2!=g1)                //数码是否可以移动  
  
{  
  
Evaluate(g2,End);      //对新节点评估  
  
//printf("开始产生的第%d个图:\n",i);  
  
//Print(g2);  
  
if(g2->evalue<=g1->evalue+1)    //是否为优越节点  
  
{  
  
g2->parent=g1;  
  
switch(Direct)              //设置屏蔽方向,防止往回推  
  
{  
  
case 1:  
  
g2->udirec=2;break;  
  
case 2:  
  
g2->udirec=1;break;  
  
case 3:  
  
g2->udirec=4;break;  
  
case 4:  
  
g2->udirec=3;break;  
  
}  
  
rear++;  
  
Qu[rear]=g2;                //存储节点到待处理队列  
  
if(g2->evalue==0)          //为0则,搜索完成  
  
{  
  
g=g2;  
  
i=5;  
  
}  
  
}  
  
else  
  
{  
  
free(g2);          //劣质节点抛弃  
  
g2=NULL;  
  
}  
  
}  
  
}  
  
Step++;                    //统计深度  
  
if(Step>Max\_Step)  
  
return NULL;  
  
if(g!=NULL)  
  
break;  
  
}  
  
return g;  
  
}  
  
/\*初始化一个八数码结构体\*/  
  
Graph \*CR\_BeginGraph(Graph \*The\_graph)  
  
{  
  
int M=10;            //随机移动步数  
  
int Direct;  
  
int i,x,y;  
  
Graph \*New\_graph;  
  
New\_graph=(Graph \*)malloc(sizeof(Graph));          //生成节点  
  
for(x=0;x<N;x++)  
  
{  
  
for(y=0;y<N;y++)  
  
New\_graph->form[x][y]=The\_graph->form[x][y];  //复制数码组  
  
}  
  
for(i=0;i<M;i++)  
  
{     
  
Direct=rand()%4;  
  
//printf("%d\n",Direct);  
  
New\_graph=Move(New\_graph,Direct,0);         
  
}  
  
New\_graph->evalue=0;  
  
New\_graph->udirec=0;  
  
New\_graph->parent=NULL;  
  
//Print(New\_graph);  
  
return New\_graph;  
  
}  
  
/\*主函数\*/  
  
void main()  
  
{     
  
/\*Graph Begin\_graph={  
  
{{2,8,3},{1,0,4},{7,6,5}},0,0,NULL  
  
};\*/  
  
/\*Graph Begin\_graph={  
  
{{2,0,1},{4,6,5},{3,7,8}},0,0,NULL  
  
};\*/  
  
/\*目标数码组\*/  
  
Graph End\_graph={  
  
{{1,2,3},{8,0,4},{7,6,5}},0,0,NULL  
  
};  
  
Graph \*G,\*P;  
  
/\*初始数码组\*/  
  
Graph \*Begin\_graph;  
  
int top=-1;  
  
Begin\_graph=CR\_BeginGraph(&End\_graph);  //随机产生初始数码组  
  
Evaluate(Begin\_graph,&End\_graph);        //对初始数码组评估  
  
printf("初始数码组:\n");  
  
//printf("udirec:%d\tparent:%d\n",Begin\_graph->udirec,Begin\_graph->parent);  
  
Print(Begin\_graph);  
  
printf("目标数码组:\n");  
  
Print(&End\_graph);   
  
/\*图搜索\*/  
  
G=Search(Begin\_graph,&End\_graph);  
  
//Print(G);  
  
/\*打印\*/   
  
if(G)  
  
{  
  
//把路径倒序  
  
P=G;  
  
//压栈  
  
while(P!=NULL)  
  
{  
  
top++;  
  
St[top]=P;  
  
P=P->parent;  
  
}  
  
printf("\n<<<<<<<<<<<<<<<<<搜索结果>>>>>>>>>>>>>>>>\n");       
  
/\*弹栈打印\*/  
  
while(top>-1)  
  
{  
  
P=St[top];  
  
top--;  
  
Print(P);  
  
}  
  
printf("\n<<<<<<<<<<<<<<<<<<完成!>>>>>>>>>>>>>>>>>>\n");  
  
}  
  
else  
  
printf("搜索不到结果.深度为%d\n",Max\_Step);  
  
}            //设计了搜索深度范围，防止队列内存越界

