学号 P21714001 专业 计算机英才班 姓名 刘峰

实验日期 **2019.12.20**  教师签字 成绩

实验报告

【实验名称】  **启发式算法解决八数码问题**

【实验目的】

八数码问题也称为九宫问题。在3×3的棋盘，摆有八个棋子，每个棋子上标有1至8的某一数字，不同棋子上标的数字不相同。棋盘上还有一个空格，与空格相邻的棋子可以移到空格中。要求解决的问题是：给出一个初始状态和一个目标状态，找出一种从初始转变成目标状态的移动棋子步数最少的移动步骤。

所谓问题的一个状态就是棋子在棋盘上的一种摆法。棋子移动后，状态就会发生改变。解八数码问题实际上就是找出从初始状态到达目标状态所经过的一系列中间过渡状态。

八数码问题一般使用搜索法来解。搜索法有广度优先搜索法、深度优先搜索法、A\*算法等。这里通过用不同方法解八数码问题来比较一下不同搜索法的效果。

【实验原理】

广度优先搜索和双向广度优先搜索都属于盲目搜索，这在状态空间不大的情况下是很合适的算法，可是当状态空间十分庞大时，它们的效率实在太低，往往都是在搜索了大量无关的状态结点后才碰到解答，甚至更本不能碰到解答。

搜索是一种试探性的查寻过程，为了减少搜索的盲目性引，增加试探的准确性，就要采用启发式搜索了。所谓启发式搜索就是在搜索中要对每一个搜索的位置进行评估，从中选择最好、可能容易到达目标的位置，再从这个位置向前进行搜索，这样就可以在搜索中省略大量无关的结点，提高了效率。

A\*算法

A\*算法是一种常用的启发式搜索算法。在A\*算法中，一个结点位置的好坏用估价函数来对它进行评估。A\*算法的估价函数可表示为：

f'(n) = g'(n) + h'(n)

这里，f'(n)是估价函数，g'(n)是起点到终点的最短路径值（也称为最小耗费或最小代价），h'(n)是n到目标的最短路经的启发值。由于这个f'(n)其实是无法预先知道的，所以实际上使用的是下面的估价函数：

f(n) = g(n) + h(n)

其中g(n)是从初始结点到节点n的实际代价，h(n)是从结点n到目标结点的最佳路径的估计代价。在这里主要是h(n)体现了搜索的启发信息，因为g(n)是已知的。用f(n)作为f'(n)的近似，也就是用g(n)代替g'(n)，h(n)代替h'(n)。这样必须满足两个条件：（1）g(n)>=g'(n)（大多数情况下都是满足的，可以不用考虑），且f必须保持单调递增。（2）h必须小于等于实际的从当前节点到达目标节点的最小耗费h(n)<=h'(n)。第二点特别的重要。可以证明应用这样的估价函数是可以找到最短路径的。

【实验内容】

A算法的步骤

A\*算法基本上与广度优先算法相同，但是在扩展出一个结点后，要计算它的估价函数，并根据估价函数对待扩展的结点排序，从而保证每次扩展的结点都是估价函数最小的结点。

A算法的步骤如下：

1）建立一个队列，计算初始结点的估价函数f，并将初始结点入队，设置队列头和尾指针。

2）取出队列头（队列头指针所指）的结点，如果该结点是目标结点，则输出路径，程序结束。否则对结点进行扩展。

3）检查扩展出的新结点是否与队列中的结点重复，若与不能再扩展的结点重复（位于队列头指针之前），则将它抛弃；若新结点与待扩展的结点重复（位于队列头指针之后），则比较两个结点的估价函数中g的大小，保留较小g值的结点。跳至第五步。

4）如果扩展出的新结点与队列中的结点不重复，则按照它的估价函数f大小将它插入队列中的头结点后待扩展结点的适当位置，使它们按从小到大的顺序排列，最后更新队列尾指针。

5）如果队列头的结点还可以扩展，直接返回第二步。否则将队列头指针指向下一结点，再返回第二步。

实验代码：

#include <iostream>

using namespace std;

#define null 0

#define HEAD 499 //表示头结点

#define INVALID\_STATE\_INDEX 10 //无效的状态

#define INVALID\_SET\_INDEX 1000

#define WIDTH 3 //棋盘的维度

/\*状态图的状态节点\*/

struct Node

{

int State[9];//该节点的状态

int Parent;//状态图中该节点的父节点

int h; //节点到达目的的困难程度

int g; //节点的实际路径

int f; //节点的总路径

};

static int End[9]; //目标棋盘

static int Start[9]; //初始棋盘

static int Invalid[9] = { 0, 0, 0, 0, 0, 0, 0, 0, 0 }; //无效的状态

static Node m;

static Node n;

static Node OPEN[500];

static Node CLOSE[500];

static int O\_Num;

static int C\_Num;

//

int IndexOfZero(int\* input)

{

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

{

if (input[i] == 0)

return i;

}

return INVALID\_STATE\_INDEX;

}

void Swap(int\* input, int a, int b)

{

int temp = input[a];

input[a] = input[b];

input[b] = temp;

}

//设置状态信息

void SetState(Node& inc, int\* ina)

{

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

{

inc.State[i] = ina[i];

}

}

void CopyState(Node &from, Node &to)

{

SetState(to, from.State);

to.f = from.f;

to.g = from.g;

to.h = from.h;

to.Parent = from.Parent;

}

//按指定的方向进行状态转移

bool Move(Node& start, Node& end, int dirct)

{

CopyState(start, end);

int p0 = IndexOfZero(start.State);

switch (dirct)

{

case 0://left

if (p0 == INVALID\_STATE\_INDEX || p0 % 3 == 2)

return false;

else

Swap(end.State, p0, p0 + 1);

return true;

case 1:

if (p0 == INVALID\_STATE\_INDEX || p0 % 3 == 0)

return false;

else

Swap(end.State, p0, p0 - 1);

return true;

case 2:

if (p0 == INVALID\_STATE\_INDEX || p0 > 5)

return false;

else

Swap(end.State, p0, p0 + 3);

return true;

case 3:

if (p0 == INVALID\_STATE\_INDEX || p0 < 3)

return false;

else

Swap(end.State, p0, p0 - 3);

return true;

default:

return false;

}

return false;

}

//计算总的路径长度

void SethANDf(Node& input)

{

input.h = 0;

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

{

if (input.State[i] != End[i])

input.h++;

} //测试一下，目前九宫格状态与目标状态相差多少，用h衡量困难程度

if (input.State[4] != End[4])

input.h--;

input.f = input.h + input.g;

}

//按代价函数从小到大的顺序排序OPEN表

void SortOPEN(Node\* OPEN, int O\_Num)

{

Node temp;

for (int i = 0; i < O\_Num; i++)

{

for (int j = 0; j < O\_Num - 1 - i; j++)

{

if (OPEN[j].f > OPEN[j + 1].f)

{

CopyState(OPEN[j], temp);

CopyState(OPEN[j + 1], OPEN[j]);

CopyState(temp, OPEN[j + 1]);

}

}

}

}

//判断两个状态是否相等

bool EqualState(Node a, Node b)

{

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

{

if (a.State[i] != b.State[i])

return false;

}

return true;

}

//判断输入时否等于结束状态

bool IsEnd(Node input)

{

Node tar;

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

{

tar.State[i] = End[i];

}

return EqualState(tar, input);

}

//判断输入状态是否在状态集中

int InSet(Node input, const Node\* set, int index)

{

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

{

if (EqualState(input, set[i]))

return i;

}

return INVALID\_SET\_INDEX;

}

//打印状态和当前代价函数

void ShowState(Node input)

{

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

{

cout << input.State[i] << " ";

if (i % 3 == 2)

cout << endl;

}

cout << "代价函数为f = " << input.f << endl << endl;

}

void Solve()

{

/\*初始化信息\*/

cout << "@初始化信息：" << endl << endl;

cout << "请输入原棋盘（用空格隔开）：" << endl;

for (int i = 0; i < WIDTH\*WIDTH; i++){

cin >> Start[i]; //Start数组为初始矩阵，End数组为目标矩阵

}

cout << "请输入目标棋盘（用空格隔开）：" << endl;

for (int i = 0; i < WIDTH\*WIDTH; i++){

cin >> End[i];

}

cout << "您输入的原棋盘为：" << endl;

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

cout << Start[i \* 3] << " " << Start[i \* 3 + 1] << " " << Start[i \* 3 + 2] << endl;

}

cout << "您输入的目标棋盘为：" << endl;

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

cout << End[i \* 3] << " " << End[i \* 3 + 1] << " " << End[i \* 3 + 2] << endl;

}

cout << "@搜索过程及代价函数：" << endl << endl;

Node prob; //新建一个变量用来保存当前

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

{

prob.State[i] = Start[i];

}

prob.g = 0;

prob.Parent = HEAD;

SethANDf(prob); //该函数得到目前所有的数码状态

CopyState(prob, OPEN[O\_Num++]);

while (O\_Num>0)

{

/\*从OPEN取出f最小的状态\*/

SortOPEN(OPEN, O\_Num);

CopyState(OPEN[0], n);

CopyState(OPEN[O\_Num - 1], OPEN[0]);//用来覆盖原来最小的和保存最大的

O\_Num--;

CopyState(n, CLOSE[C\_Num++]);

if (IsEnd(n))

{

cout << "存在最佳路径：" << endl;

int index = C\_Num - 1;

while (index != HEAD)

{

ShowState(CLOSE[index]);

index = CLOSE[index].Parent;

}

return;

//success

}

else

{

if (Move(n, m, 0) && !EqualState(m, CLOSE[n.Parent]))

{//move left

m.g = n.g + 1;

m.Parent = C\_Num - 1;

SethANDf(m);

if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX&&InSet(m, CLOSE, C\_Num) == INVALID\_SET\_INDEX)//neither in close nor in open

{

CopyState(m, OPEN[O\_Num++]);

}

else if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX)//in close

{

int C\_Num2 = InSet(m, CLOSE, C\_Num);

if (m.f < CLOSE[C\_Num2].f)

{

CopyState(m, OPEN[O\_Num++]);

SetState(CLOSE[C\_Num2], Invalid);

}

}

else

{ //in open

int C\_Num2 = InSet(m, OPEN, O\_Num);

if (m.f < OPEN[C\_Num2].f)

{

OPEN[C\_Num2].g = m.g;

OPEN[C\_Num2].Parent = C\_Num - 1;

SethANDf(OPEN[C\_Num2]);

}

}

}

if (Move(n, m, 1) && !EqualState(m, CLOSE[n.Parent]))

{//move right

m.g = n.g + 1;

m.Parent = C\_Num - 1;

SethANDf(m);

if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX&&InSet(m, CLOSE, C\_Num) == INVALID\_SET\_INDEX)//neither in close nor in open

{

CopyState(m, OPEN[O\_Num++]);

}

else if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX)//in close

{

int C\_Num2 = InSet(m, CLOSE, C\_Num);

if (m.f < CLOSE[C\_Num2].f)

{

CopyState(m, OPEN[O\_Num++]);

SetState(CLOSE[C\_Num2], Invalid);

}

}

else

{ //in open

int C\_Num2 = InSet(m, OPEN, O\_Num);

if (m.f < OPEN[C\_Num2].f)

{

OPEN[C\_Num2].g = m.g;

OPEN[C\_Num2].Parent = C\_Num - 1;

SethANDf(OPEN[C\_Num2]);

}

}

}

if (Move(n, m, 2) && !EqualState(m, CLOSE[n.Parent]))

{//move up

m.g = n.g + 1;

m.Parent = C\_Num - 1;

SethANDf(m);

if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX&&InSet(m, CLOSE, C\_Num) == INVALID\_SET\_INDEX)//neither in close nor in open

{

CopyState(m, OPEN[O\_Num++]);

}

else if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX)//in close

{

int C\_Num2 = InSet(m, CLOSE, C\_Num);

if (m.f < CLOSE[C\_Num2].f)

{

CopyState(m, OPEN[O\_Num++]);

SetState(CLOSE[C\_Num2], Invalid);

}

}

else

{ //in open

int C\_Num2 = InSet(m, OPEN, O\_Num);

if (m.f < OPEN[C\_Num2].f)

{

OPEN[C\_Num2].g = m.g;

OPEN[C\_Num2].Parent = C\_Num - 1;

SethANDf(OPEN[C\_Num2]);

}

}

}

if (Move(n, m, 3) && !EqualState(m, CLOSE[n.Parent]))

{//move down

m.g = n.g + 1;

m.Parent = C\_Num - 1;

SethANDf(m);

if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX&&InSet(m, CLOSE, C\_Num) == INVALID\_SET\_INDEX)//neither in close nor in open

{

CopyState(m, OPEN[O\_Num++]);

}

else if (InSet(m, OPEN, O\_Num) == INVALID\_SET\_INDEX)//in close

{

int C\_Num2 = InSet(m, CLOSE, C\_Num);

if (m.f < CLOSE[C\_Num2].f)

{

CopyState(m, OPEN[O\_Num++]);

SetState(CLOSE[C\_Num2], Invalid);

}

}

else

{ //in open

int C\_Num2 = InSet(m, OPEN, O\_Num);

if (m.f < OPEN[C\_Num2].f)

{

OPEN[C\_Num2].g = m.g;

OPEN[C\_Num2].Parent = C\_Num - 1;

SethANDf(OPEN[C\_Num2]);

}

}

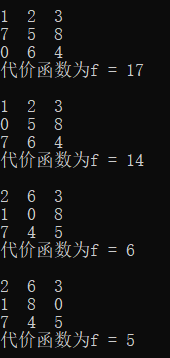
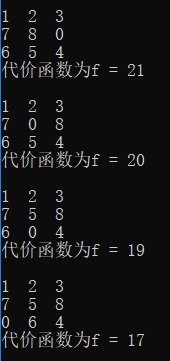
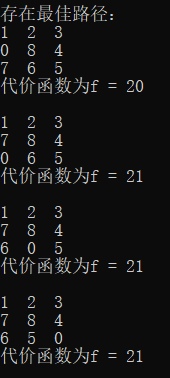
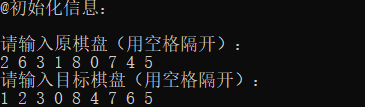
}

}

}

}

**实验结果：**



【实验总结】

利用启发式规则解决八数码问题，体现了启发式规则的智能性，理解了程序具有智能特性的原因---启发式算法。通过本次实验，基本了解了启发式算法的工作原理，并回顾了以前学过的数据结构和深度优先搜索算法，温故而知新。