****



**实验一：A算法求解8数码**

**学 院 智能与计算学部**

**专 业 网络空间安全**

**年 级 2020**

**姓 名 石子跃**

**学 号 3020244294**

**指导教师**

**实验一：A算法求解8数码**

1. **实验目的**

学习A算法，熟练掌握其基本原理，用启发式搜索解决8数码问题

1. **实验内容**

**八数码问题：**

状态：描述8个棋子和空位在棋盘的9个方格上的分布情况。其中，任何状态都可以被指定为初始状态。

操作符：产生4个行动，即上下左右移动

目标测试：用来检测状态是否能匹配上给定的目标状态。

路径费用函数：每一步的费用为1，因此整个路径的费用是路径中的步数。

问题描述：给定任意一个初始状态，要求找到一种搜索策略，用尽可能少的步数得到上图的目标状态。

(a)初始状态

(b)目标状态

**A算法：**

A算法特点在于对**估价函数f**的定义上。对于一般的启发式图搜索，总是**选择估价函数f值最小**的节点作为扩展节点。

估价函数：

f(n)＝ g(n)＋ h(n)

* + g(n)为初始状态到状态n是已付出的实际代价；
  + h(n)是从状态n到目标状态的最优路径的估计代价，而搜索的启发式信息主要由h(n)决定。

1. **实验设计方案**
2. **确定A算法中的估价函数f**

f(x)＝ g()＋ h()

代价函数值即结点深度 + 每有一位与目标节点不一致，启发函数值加一

1. **算法设计**

OPEN表保存所有已生成而未考察的节点

CLOSED表中记录已访问过的节点。

1.将起始点加入open表

2.当open表不为空时：

3. 寻找open表中f值**最小的点**current

4. 如果current是终止点，则找到结果，程序结束。

5. 否则，open表移出current，对current表中的每一个临近点：

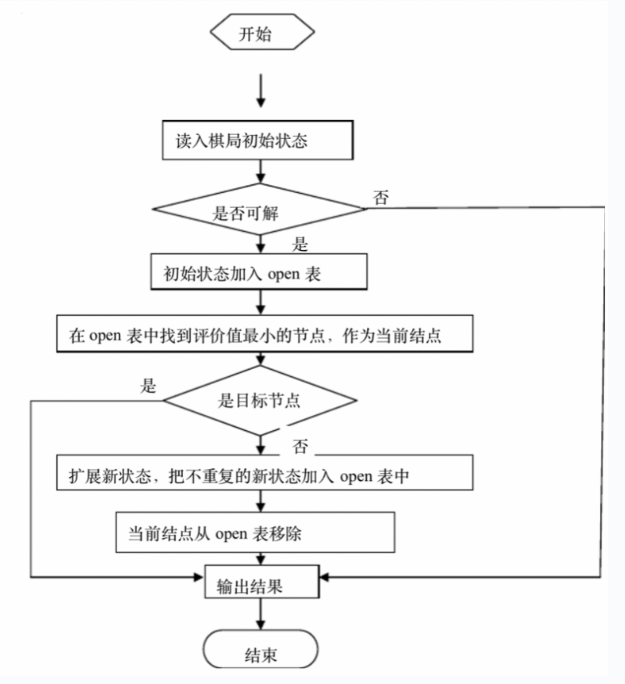
6. 若它不可走或在close表中，略过

7. 若它不在open表中，加入。

8. 若它在open表中，计算g值，若g值更小，替换其父节点为current，更新它的g值。

9..若open表为空，则路径不存在。

1. **实验流程图**



1. **实验代码**

#include <iostream>

#include <stdlib.h>

#include <windows.h>

#include <time.h>

#include <type\_traits>

using namespace std;

#define MAXNUM 100

#define SUCESS 1

#define ERROR 0

int Initial\_arrary[9] = {0};                      // 初始结点，待输入

int Goal\_arrary[9] = {1, 2, 3, 8, 0, 4, 7, 6, 5}; // 目标节点

class Table;

class Node

{ // 节点类

public:

    Node();

    Node(int s[]);

    Node &operator=(const Node ss);

    void showstate();               // 显示结点状态

    bool operator==(Node &N);       // 运算符重载，用于判断是否相等

    int g();                        //实际代价

    int h();                        //估计代价

    int fx();                       // 估价函数

    int expand(Node kid[]);         // 扩展子节点

    friend class Table;

    friend int A(Node S0, Node Sg);

    int \*state = new int[9]; // 结点状态

    int depth;               // 节点深度

    int f;                   // 结点fx值 ֵ

    Node \*father;            // ָ父节点指针

};

Node deleted; // 表示被删除的结点，用于比较与赋值

Node::Node()  // 默认构造函数

{

    for (int i = 0; i < 9; i++)

    {

        state[i] = 0;

    }

    f = depth = 0;

    father = NULL;

}

Node::Node(int s[]) // 用数组初始化的构造函数

{

    for (int i = 0; i < 9; i++)

    {

        state[i] = s[i];

    }

    depth = 0;

    f = fx();

    father = NULL;

}

Node &Node::operator=(const Node ss) //运算符重载，用于赋值

{

    for (int i = 0; i < 9; i++)

    {

        state[i] = ss.state[i];

    }

    depth = ss.depth;

    f = ss.f;

    father = ss.father;

    return \*this;

}

void Node::showstate()

{

    cout << "┏━━━━━━━━━━┓" << endl;

    for (int i = 0; i < 3; i++)

    {

        cout << "┃";

        for (int j = 0; j < 3; j++)

        {

            if (state[i \* 3 + j] == 0)

            {

                cout << "   ";

            }

            else

            {

                cout << "  " << state[i \* 3 + j];

            }

        }

        cout << " ┃" << endl;

    }

    cout << "┗━━━━━━━━━━┛" << endl;

}

bool Node::operator==(Node &N)

{

    for (int i = 0; i < 9; i++)

    {

        if (state[i] != N.state[i])

        {

            return false;

        }

    }

    return true;

}

int Node::g()

{

    return depth; // 代价函数值即结点深度

}

int Node::h()

{

    int h = 0;

    for (int i = 0; i < 9; i++)

    {

        if (state[i] != Goal\_arrary[i])

        {

            h++; // 每有一位与目标节点不一致，启发函数值加一

        }

    }

    return h;

}

int Node::fx()

{

    return (g() + h());

}

int Node::expand(Node kid[])

{

    int point, i, j;

    for (i = 0; i < 9; i++)

    {

        if (state[i] == 0)

        {

            point = i;

        }

    }

    i = 0;

    if (point - 3 >= 0)

    { // 防止向上扩展越界

        for (j = 0; j < 9; j++)

        {

            kid[i].state[j] = state[j];

        }

        swap(kid[i].state[point], kid[i].state[point - 3]);

        kid[i].depth = depth + 1;

        kid[i].f = kid[i].fx();

        i++;

    }

    if (point + 3 <= 8)

    { // 防止向下扩展越界

        for (j = 0; j < 9; j++)

        {

            kid[i].state[j] = state[j];

        }

        swap(kid[i].state[point], kid[i].state[point + 3]);

        kid[i].depth = depth + 1;

        kid[i].f = kid[i].fx();

        i++;

    }

    if (point % 3 != 0)

    { // 防止向左扩展越界

        for (j = 0; j < 9; j++)

        {

            kid[i].state[j] = state[j];

        }

        swap(kid[i].state[point], kid[i].state[point - 1]);

        kid[i].depth = depth + 1;

        kid[i].f = kid[i].fx();

        i++;

    }

    if ((point + 1) % 3 != 0)

    { // 防止向右扩展越界

        for (j = 0; j < 9; j++)

        {

            kid[i].state[j] = state[j];

        }

        swap(kid[i].state[point], kid[i].state[point + 1]);

        kid[i].depth = depth + 1;

        kid[i].f = kid[i].fx();

        i++;

    }

    if (i == 0)

    {

        return ERROR;

    }

    else

    {

        return i;

    }

}

class Table

{

public:

    Table();

    int Add(Node S);

    int Delete(int i);

    bool isempty();

    int Search(Node NS);

    int Sort();                     // 将表按fx升序排序

    void showNode();                // 打印表

    friend int A(Node S0, Node Sg); // A算法

    Node node[MAXNUM]; // 表中结点

    int length;        // 表长

};

Table::Table()

{

    for (int i = 0; i < MAXNUM; i++)

    { // 初始化表长为0

        length = 0;

    }

}

int Table::Add(Node S)

{

    if (length < MAXNUM)

    { // 判断是否已满

        node[length] = S;

        length++;

        return SUCESS;

    }

    else

    {

        return ERROR;

    }

}

int Table::Delete(int i)

{

    if (i > 0 && i < length)

    { // 判断是否表空

        for (int j = i - 1; j < length - 1; j++)

        {

            node[j] = node[j + 1];

        }

        length--;

        return SUCESS;

    }

    else if (i == length)

    {

        node[i - 1] = deleted;

        length--;

        return SUCESS;

    }

    else

    { // 错误提示

        cout << "输入错误" << endl;

        return ERROR;

    }

}

bool Table::isempty()

{

    if (length > 0)

    { // 非空

        return false;

    }

    else

    { // 为空

        return true;

    }

}

int Table::Search(Node NS)

{

    for (int i = 0; i < length; i++)

    {

        if (NS == node[i])

        {

            return i + 1; // 返回NS在表中序号

        }

    }

    return 0;

}

int Table::Sort()

{ // 按fx升序排序

    if (isempty())

    {

        return ERROR;

    }

    else

    {

        Node temp;

        for (int i = 0; i < length - 1; i++)

        {

            for (int j = 0; j < length - 1 - i; j++)

            {

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

                {

                    temp = node[j];

                    node[j] = node[j + 1];

                    node[j + 1] = temp;

                }

            }

        }

        return SUCESS;

    }

}

void Table::showNode()

{ // 打印表

    if (length == 0)

    {

        cout << "空!" << endl;

    }

    else

    {

        for (int i = 0; i < length - 1; i++)

        {

            node[i].showstate();

            cout << "     ↓     " << endl;

        }

        node[length - 1].showstate();

    }

}

Table Path; // 记录求解路径

int A(Node S0, Node Sg)

{                           // A算法

    Table OPEN, CLOSED;     // OPEN和CLOSED表

    int &n = CLOSED.length; // CLOSED表长别名

    Node \*kid = new Node[4];

    cout << "初始状态：" << endl;

    S0.showstate();

    cout << "目标状态：" << endl;

    Sg.showstate();

    cout << endl

         << "============================" << endl;

    OPEN.Add(S0); //  1.将起始点加入open表

    while (!OPEN.isempty())

    {

        CLOSED.Add(OPEN.node[0]);           // 3.寻找open表中f值最小的点current

        Node &current = CLOSED.node[n - 1]; //

        if (current == Sg)// 4. 如果current是终止点，则找到结果，程序结束。

        {

            cout << "成功的找到啦" << endl;

            cout << "============================" << endl;

            return SUCESS;

        }

        OPEN.Delete(1); // 5.   否则，open表移出current，对current表中的每一个临近点

        int kid\_num = current.expand(kid);//对扩展的字结点做相应处理

        for (int i = 0; i < kid\_num; i++)

        {

            int exOPEN = OPEN.Search(kid[i]);     // 生成子结点中是否有OPEN表中已存在的点

            int exCLOSED = CLOSED.Search(kid[i]); // 生成子结点中是否有CLOSED表中已存在的点

            // 6.       若它不可走或在close表中，略过

            // 7.       若它不在open表中，加入。

            if (exOPEN > 0)

            { // 8.     若它在open表中，计算g值，若g值更小，替换其父节点为current，更新它的g值。

                if (OPEN.node[exOPEN - 1].f > kid[i].f)

                {

                    OPEN.node[exOPEN - 1].father = &current;

                    OPEN.node[exOPEN - 1].f = kid[i].f;

                }

                kid[i] = deleted;

            }

            else if (exCLOSED > 0)

            { //若在closed表中忽略

                kid[i] = deleted;

            }

            else

            {

                kid[i].father = &current; // 修改父节点指针

            }

            if (!(kid[i] == deleted)) // 其余子节点不在open中并且不必父节点小加入OPEN中

            {

                OPEN.Add(kid[i]);

            }

        }

        OPEN.Sort(); // OPEN表排序

        if ((Path.node[Path.length - 1] == \*OPEN.node[0].father) && !OPEN.isempty())

        {

            Path.Add(OPEN.node[0]);//将下一个最优解加入至PATH

        }

        cout << "OPEN表如下：" << endl;

        OPEN.showNode();

        cout << endl;

        cout << "CLOSED表如下：" << endl;

        CLOSED.showNode();

        cout << endl;

        cout << "路径如下：" << endl;

        Path.showNode();

        cout << endl;

    } // while

    delete kid;

    return ERROR;

}

int main()

{

    cout << "输入初始状态(0-9中间用空格分开): ";

    for (int i = 0; i < 9; i++)

    {

        cin >> Initial\_arrary[i];

    }

    Node S0(Initial\_arrary);

    Path.Add(S0);

    Node Sg(Goal\_arrary);

    cout << endl;

    if (A(S0, Sg) == SUCESS)

    { //搜索成功

        cout << endl

             << endl;

        cout << "最终的过程如下: " << endl;

        Path.showNode(); // 打印求解路径

        cout<<"路径长度："<<Path.node[Path.length-1].depth<<endl;

    }

    else

    { // 提示搜索失败

        cout << "失败！找不到！" << endl;

    }

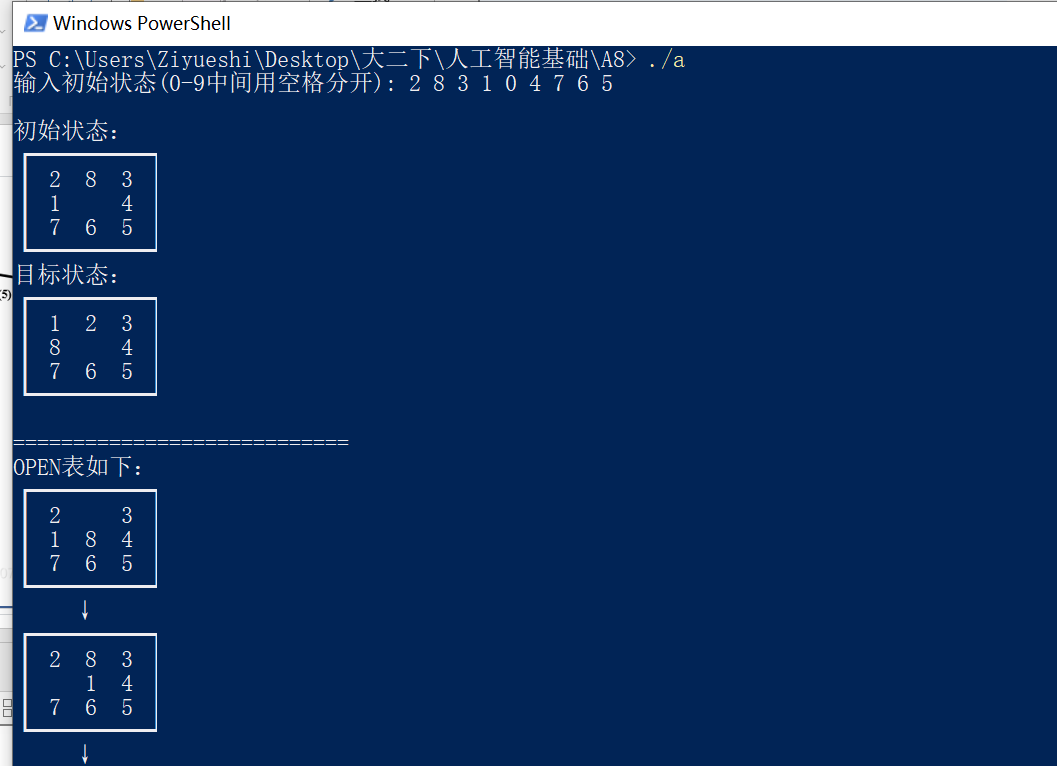
    return 0;

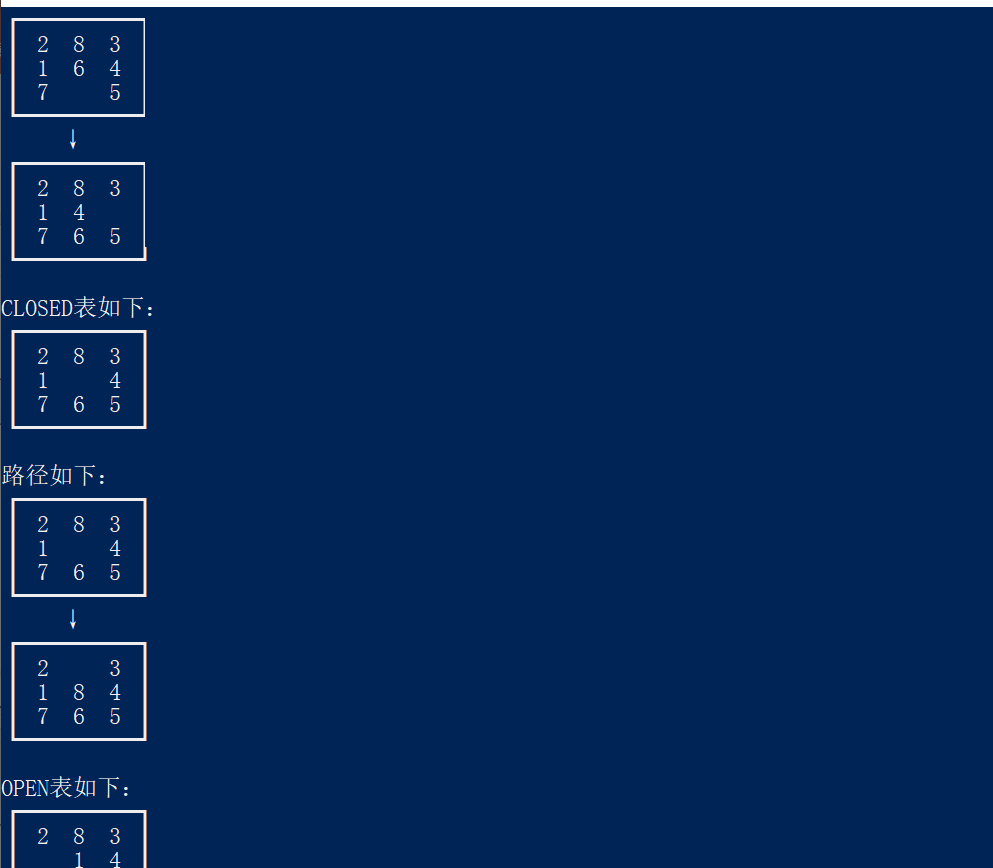
}

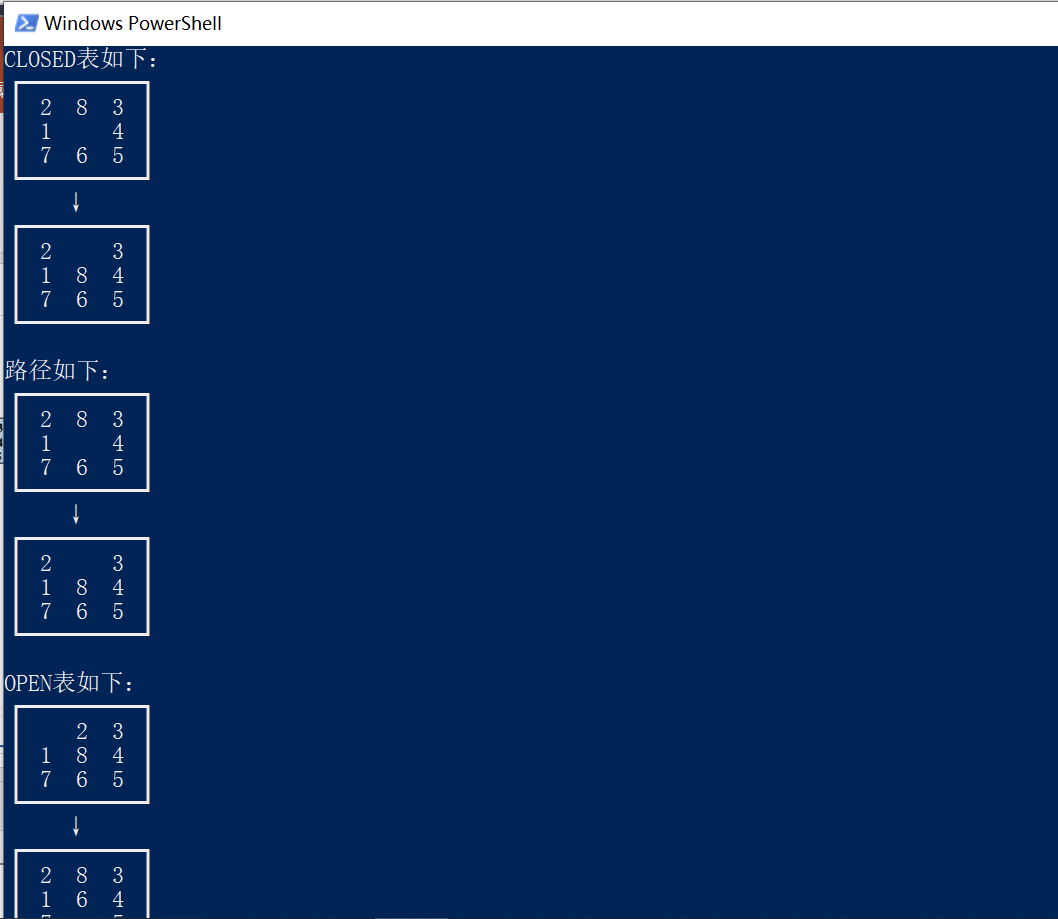
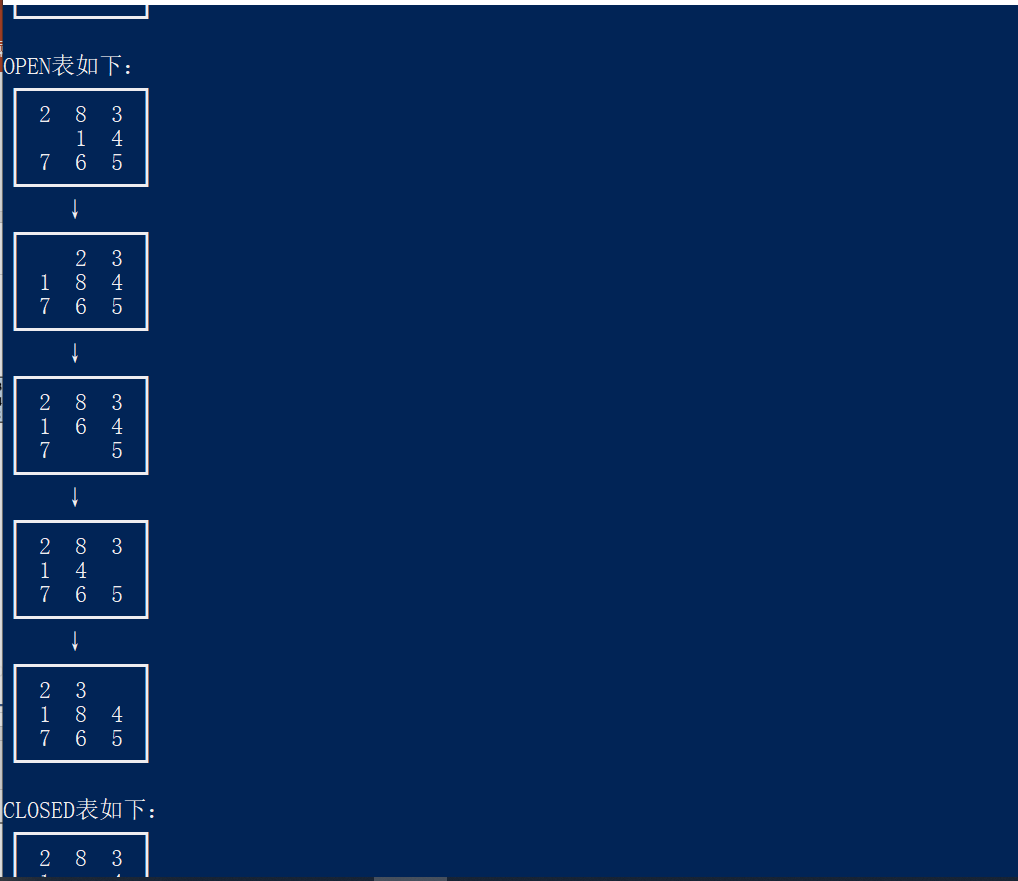
1. **实验结果**

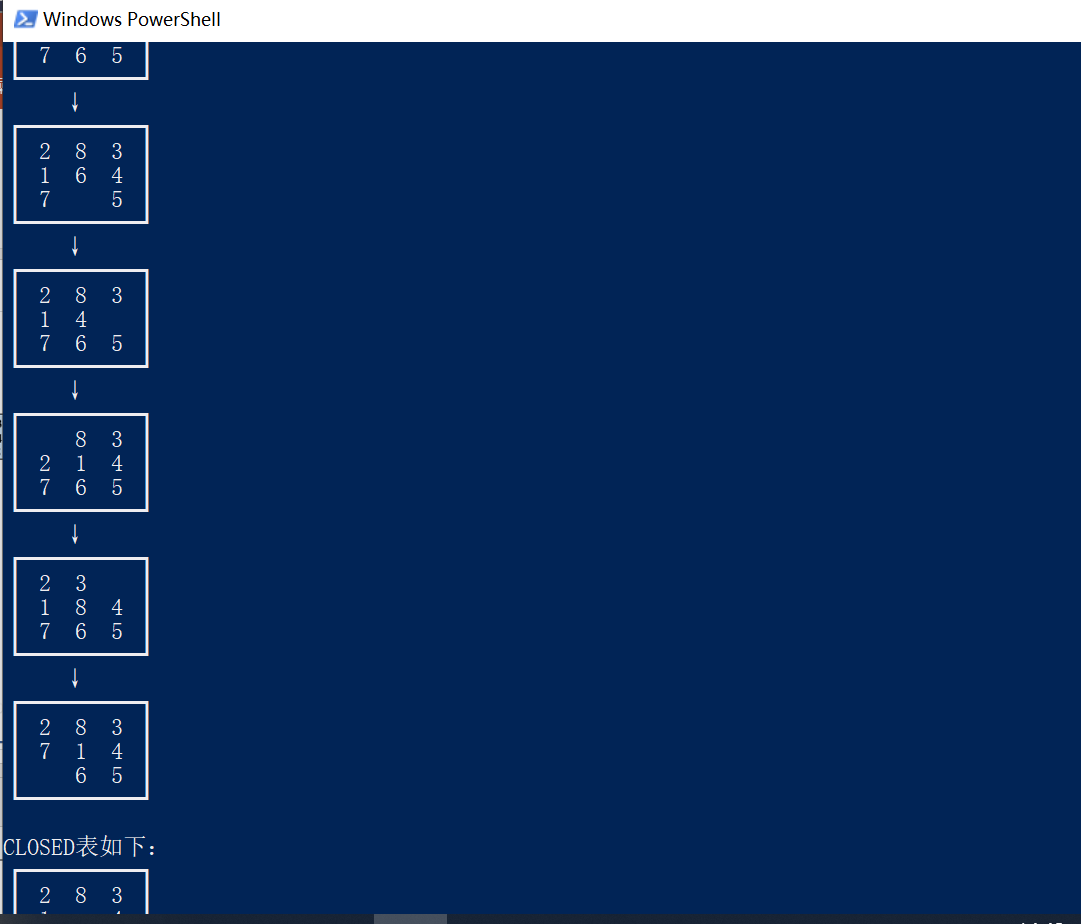
在终端中输入如下2 8 3 1 0 4 7 6 5

将会打印出每一次移动后的OPEN表，CLOSED表和当前走过的路径，在最后输出路径长度。

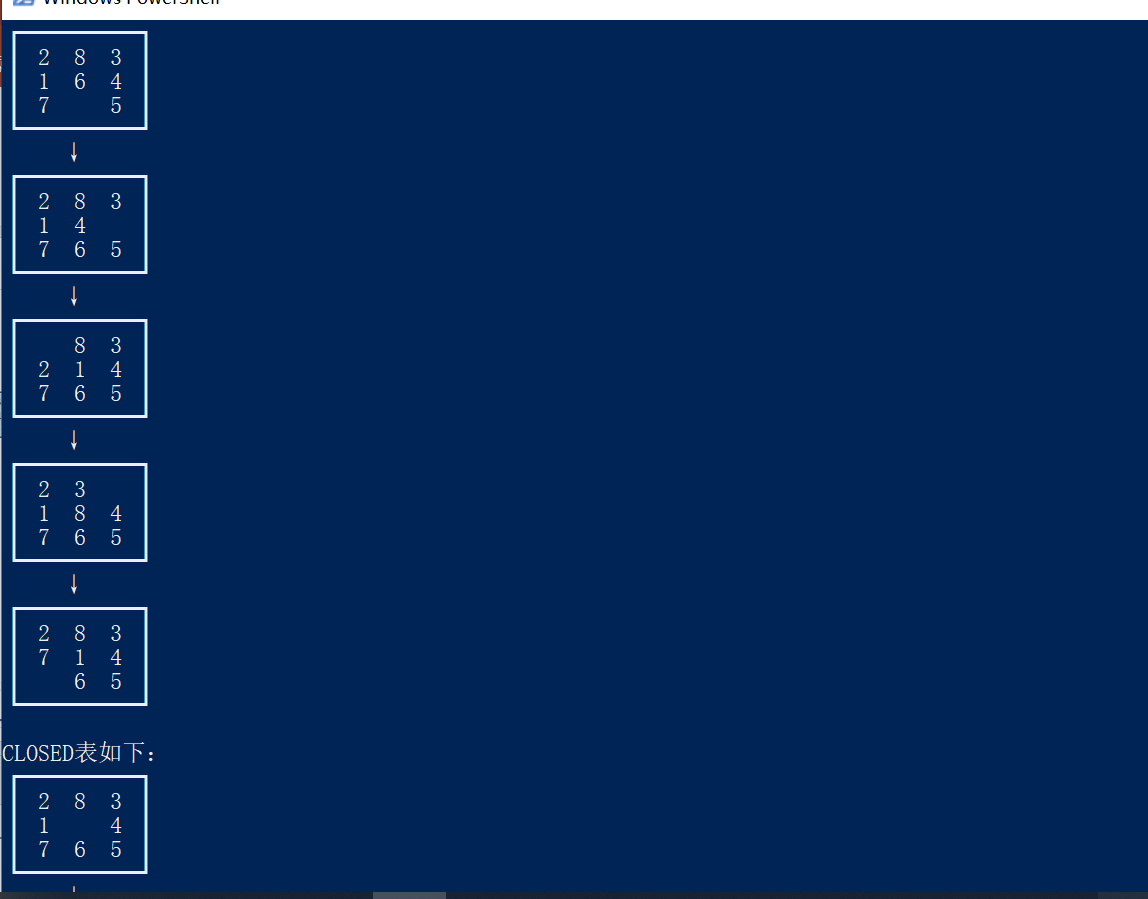
****

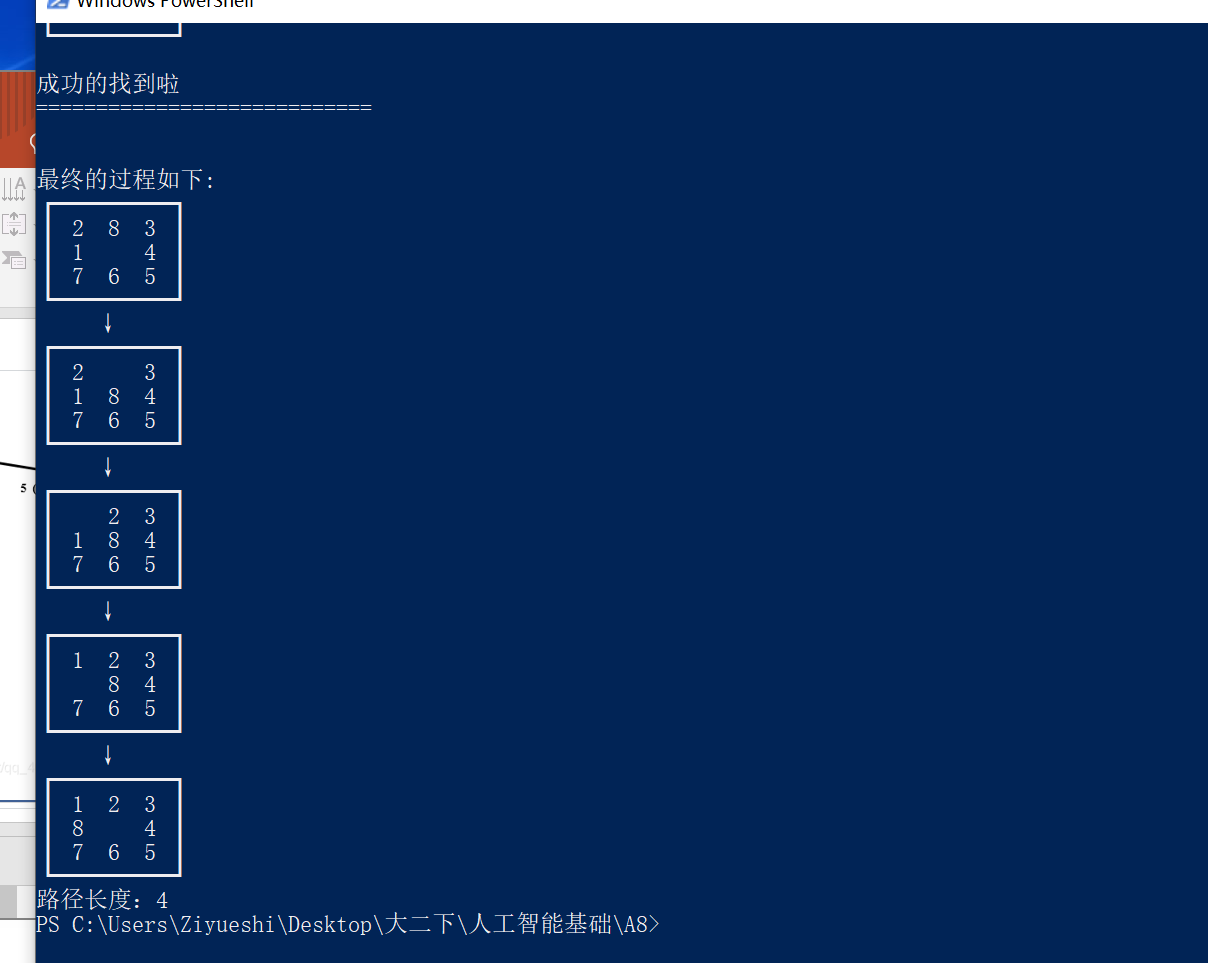
****

****

****

****

****

****