八数码实验报告

马斓轩 1813076

一、编程语言及环境

语言：C++

环境：Visual Studio 2019

二、实验原理算法说明

问题说明：

八数码问题是指这样一种游戏：将分别标有数字 1，2，3，…，8 的八块正方形数码牌任意地放在一块 3*3 的数码盘上。放牌时要求不能重叠。于是，在 3*3 的数码盘上出现了一个空格。现在要求按照每次只能将与空格相邻的数码牌与空格交换的原则，将任意摆放的数码盘逐步摆成某种特殊的排列。

问题分析：

八数码问题包括一个初始状态和目标状态，所谓解八数码问题就是在两个状态间寻找一系列可过渡状态。从一个状态出发，找到空位后有最多四种变换方式，一层一层创建树形结构。

这个状态可以有很多种表示方法。第一种是使用 int 表示，这种方法的好处是可以压缩空间，但是这仅使用于八数码问题，如果扩展到 15 数码将无法表示。第二种是使用一维数组，一维数组是较好的一种兼顾空间和时间的表示方法，但是在扩展 15 数码时需要重新对位置转换进行计算。第三种是使用多维数组表示，这种表示方法的好处是简单明了，易于扩充，且可以很方便地看出运行过程，弊端就是时间和空间上的消耗。在本次实验中我选择使用构造节点进行搜索，一个节点就是一个二维数组及其他属性，因此可以清晰地观测搜索顺序及运行结果。

其次要解决的问题是如何判断初始状态到目标状态是否是可达的。两个状态之间是否可达，可以通过计算两者的逆序值，若两者奇偶性相同则可达，不然两个状态不可达。从 3*3 棋盘上可知，移动方式分为上下左右四种，左右移动不会改变逆序数，而上下移动增加 2 或减少 2。因此，如果初始排列和目标排列逆序数奇偶性相同，就认为是有解的。除了使用逆序数判断还可以通过判断 Open 表

是否为空来判断。如果状态是可达的，那么最终在 Open 表一定有一个与目标节点相同的节点，此为该题的解，若无解，运行结束后 Open 表为空。这里选择使用搜索前先进行判断即逆序数方法。

在搜索中还有一个需要解决的问题是避免重复搜索，否则有可能进入循环状态。每次得到的新节点与已经搜索过的节点做比较，如果均不相同就说明是可行的，否则就是重复的。

准备工作：

1. 用逆序数同奇偶判断是否有解。

```
//是否有解
bool is_access(int cur[], int aim[]) {
    int inver_count1 = 0;
    for (int i = 0; i < ROW*COL-1; i++) {
        //忽略0
        if (cur[i] == 0) {
            continue;
        }
        int temp = cur[i];
        for (int j = i + 1; j < ROW * COL; j++) {
            //忽略0
            if (cur[j] == 0) {
                continue;
            }
            if (temp > cur[j]) {
                inver_count1++;
            }
        }
    }
    cout << inver_count1 << endl;
    int inver_count2 = 0;
    for (int i = 0; i < ROW * COL-1; i++) {
        if (aim[i] == 0) {
            continue;
        }
        int temp = aim[i];
        for (int j = i + 1; j < ROW * COL; j++) {
            if (aim[j] == 0) {
                continue;
            }
            if (temp > aim[j]) {
                inver_count2++;
            }
        }
    }
    cout << inver_count2 << endl;
    if ((inver_count1 + inver_count2) % 2 == 0) {
        return true;
    }
    else {
        return false;
    }
}
```

2. 判断是否为重复搜索节点。使用 is_expandable()和 is_equal()两个函数达到目的。所有搜索过的节点均存储在容器 node_v 中，依次与容器中的节点比较，如果均不相同说明可扩展。

```
//两节点是否相同
bool is_equal(int node_digit[][COL], int digit[][COL]) {
    for (int i = 0; i < ROW; i++)
        for (int j = 0; j < COL; j++) {
            if (node_digit[i][j] != digit[i][j])
                return false;
        }
    return true;
}
```

```
//是否为可扩充节点
bool is_expandable(Node& node) {
    for (int i = 0; i < node_v.size(); i++) {
        if (is_equal(node_v[i].digit, node.digit))
            return false;
    }
    return true;
}
```

3. 为了较好的输出格式，重写<<。

```
//重写输出流，每次按格式输出
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;
}
```

4. 回溯法输出路径。

```
//打印每次移动结果
int 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;
    }
    int count = 0;
    for (int i = rstep_v.size() - 1; i >= 0; i--) {
        cout << "Step " << rstep_v.size() - i
            << endl << rstep_v[i] << endl;
        count++;
    }
    return count;
}
```

核心算法：

1. 广度优先搜索（BFS）

广度搜索是以接近起始节点的程度一次扩展节点，逐层搜索，在对下一层节点搜索前必须搜索完本层的所有节点。

广度搜索在八数码问题中的应用如下：将新节点放入 Open 表，然后寻找当前节点的空位，并进行上下左右移动，得到的新节点放入 Open 表，将当前节点放入 Close 表。

首次定义为节点定义一个结构体，其中 index 是父节点在 vector 中的下标。

```
typedef struct _Node {
    int digit[ROW][COL];
    int index; // 父节点标记
} Node;
Node src, dest;

vector<Node> node_v;    // 存储节点
```

Open 表使用 vector 动态数组实现，若扩展得到的新节点和目标节点相同，就返回当前该节点在 vector 中的下标，即最后一个位置；若和目标节点不同，返回 -1。

```
//向上移动的子节点
Node node_up;
Assign(node_up, index);
if (x > 0) {
    Swap(node_up.digit[x][y], node_up.digit[x - 1][y]);
    if (is_expandable(node_up)) {
        node_up.index = index;
        cout << node_up << "   "<<++tag<<endl;
        if (is_equal(node_up.digit, dest.digit)) {
            node_v.push_back(node_up);
            return node_v.size()-1;
        }
        else {
            node_v.push_back(node_up);
        }
    }
}
```

```
//向下移动的子节点
Node node_down;
Assign(node_down, index);
if (x < 2) {
    Swap(node_down.digit[x][y], node_down.digit[x + 1][y]);
    if (is_expandable(node_down)) {
        node_down.index = index;
        cout << node_down << "   " << ++tag << endl;
        if (is_equal(node_down.digit, dest.digit)) {
            node_v.push_back(node_down);
            return node_v.size()-1;
        }
        else {
            node_v.push_back(node_down);
        }

    }
}
```

```
//向左移动的子节点
Node node_left;
Assign(node_left, index);
if (y > 0) {
    Swap(node_left.digit[x][y], node_left.digit[x][y - 1]);
    if (is_expandable(node_left)) {
        node_left.index = index;
        cout << node_left << "   " << ++tag << endl;
        if (is_equal(node_left.digit, dest.digit)) {
            node_v.push_back(node_left);
            return node_v.size()-1;
        }
        else {
            node_v.push_back(node_left);
        }
    }
}
```

```
//向右移动的子节点
Node node_right;
Assign(node_right, index);
if (y < 2) {
    Swap(node_right.digit[x][y], node_right.digit[x][y + 1]);
    if (is_expandable(node_right)) {
        node_right.index = index;
        cout << node_right << "   " << ++tag << endl;
        if (is_equal(node_right.digit, dest.digit)) {
            node_v.push_back(node_right);
            return node_v.size()-1;
        }
        else {
            node_v.push_back(node_right);
        }
    }
}
return -1;
```

2. 深度优先搜索（DFS）

深度搜索是扩展最深的节点，这使得搜索沿着状态空间的某条单一路径从起始节点向下进行，只有当搜索到叶节点的状态时才考虑另外一条路径。为了避免考虑太长的路径，防止搜索过程沿着无益的路径扩展下去，往往给出一个扩展的最大深度界限。任何节点如果达到了深度界限都将作为没有后继节点处理。

深度搜索在八数码问题上的应用如下：考察当前节点是否为目标节点，若不是目标节点则将新节点放入 Open 表前端，这里可以使用栈来辅助运行；若是得到的节点与目标节点相同，使用回溯法打印查询路径。

为一个节点定义一个结构体，self 为当前节点在 vector 中的位置，设置 bool 变量维护搜索状态，初始均未被搜索。

```
typedef struct _Node {
    int digit[ROW][COL];
    int self;//当前标记
    int depth;//深度
    bool visit = false;//是否已经搜索过
    int index; // 父节点标记

} Node;
Node src, dest;
```

设置深度限度为 7（根节点深度为 0）。未到达深度限度且栈顶元素未被搜索过，将栈顶元素作为当前节点进行搜索，该节点的空位向上下左右四个方向移动后，将可扩展节点依次放入栈和容器中，如果与目标节点相同，返回该节点在容器中的位置，回溯打印路径；否则返回-1。已到达最大深度或栈顶元素已被搜索过，直接弹出，考虑新的栈顶元素。

```
//向上移动的子节点
Node node_up;
Assign(node_up, node_parent);
if (x > 0) {
    Swap(node_up.digit[x][y], node_up.digit[x - 1][y]);
    if (is_expandable(node_up)) {
        node_up.index = index;
        node_up.self = node_v.size();
        node_up.depth = node_v[index].depth + 1;
        node_v.push_back(node_up);
        node_s.push(node_up);
        cout << node_up << "   " << ++tag << endl;
        if (is_equal(node_up.digit, dest.digit)) {
            return node_v.size()-1;
        }
    }
}
//向左移动的子节点
Node node_left;
Assign(node_left, node_parent);
if (y > 0) {
    Swap(node_left.digit[x][y], node_left.digit[x][y - 1]);
    if (is_expandable(node_left)) {
        node_left.index = index;
        node_left.self = node_v.size();
        node_left.depth = node_v[index].depth + 1;
        node_v.push_back(node_left);
        node_s.push(node_left);
        cout << node_left << "   " << ++tag << endl;
        if (is_equal(node_left.digit, dest.digit)) {
            node_v.push_back(node_s.top());
            return node_v.size() - 1;;
        }
    }
}
```

```
//向下移动的子节点
Node node_down;
Assign(node_down, node_parent);
if (x < 2) {
    Swap(node_down.digit[x][y], node_down.digit[x + 1][y]);
    if (is_expandable(node_down)) {
        node_down.index = index;
        node_down.self = node_v.size();
        node_down.depth = node_v[index].depth + 1;
        node_v.push_back(node_down);
        node_s.push(node_down);
        cout << node_down << "   " << ++tag << endl;
        if (is_equal(node_down.digit, dest.digit)) {
            return node_v.size() - 1;
        }
    }
}
//向右移动的子节点
Node node_right;
Assign(node_right, node_parent);
if (y < 2) {
    Swap(node_right.digit[x][y], node_right.digit[x][y + 1]);
    if (is_expandable(node_right)) {
        node_right.index = index;
        node_right.self = node_v.size();
        node_right.depth = node_v[index].depth + 1;
        node_v.push_back(node_right);
        node_s.push(node_right);
        cout << node_right << "   " << ++tag << endl;
        if (is_equal(node_right.digit, dest.digit)) {
            return node_v.size() - 1;
        }
    }
}
return -1;
}
```

3. 启发式搜索（A*）

启发式搜索是在路径搜索问题中很使用的搜索方式，通过设计一个好的启发式函数来计算状态的优先级，优先考虑优先级高的状态，可以提早搜索到达目标态的时间。

A*算法的核心是启发式函数 f=g+h，g 是初始节点到节点 n 的一条最佳路径的实际代价，可以用节点所在层数来表示；h 是节点 n 与目标节点的预测距离，可以使用曼哈顿距离计算，每移动一步相当于移动一个数字，如果每次移动都是完美的，那么从初始态到目标态的距离就是曼哈顿距离，也是最少要走的步数。Open 表使用 vector 动态数组，Close 表通过将 h 标记为 MAXNUM 达到已搜索过的效果。

首先为一个节点定义一个结构体，src 为初始状态节点，dest 为最终状态节点。

```
typedef struct _Node {
    int digit[ROW][COL];
    int dist;    // h(n)
    int dep;     // g(n)
    int index; // 父节点标记
} Node;
Node src, dest;
```

获取 Open 表中估价值最小的节点并将其作为可扩充节点继续向下搜索子节点，每次搜索需要计算估价值。Distance 用于计算 h(n)，g(n)深度递增 1。直到得到目标节点，按照父节点标记向上回溯打印变换过程。

将节点放入 Open 表的过程分别是先查找空位，向上、下、左、右移动判断是否为可扩充节点。

```
//获取open表中估价最小的节点
int GetMinNode() {
    int dist = MAXNUM;
    int pos; //最小节点在数组中的位置
    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) {
            pos = i;
            dist = node_v[i].dist + node_v[i].dep;
        }
    }
    return pos;
}
```

```
void ProcessNode(int index) {
    int x, y;
    bool flag;
    //查找空位
    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;
    }
```

```
//向上移动的子节点进入open表
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 (is_expandable(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);
    }
}
```

```
//向下移动的子节点进入open表
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 (is_expandable(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);
    }
}
```

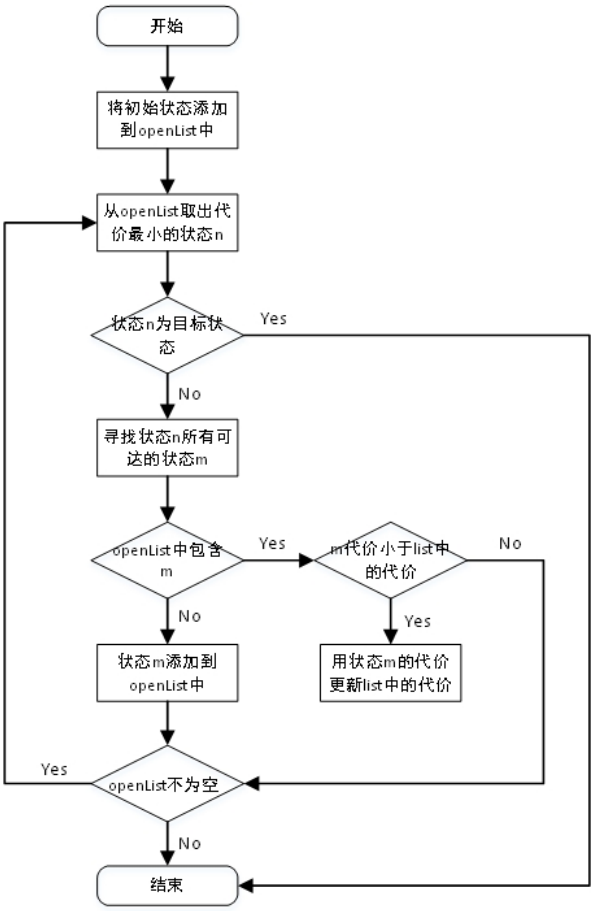
```
//向左移动的子节点进入open表
Node node_left;
Assign(node_left, index);
int dist_left = MAXDISTANCE;
if (y > 0) {
    Swap(node_left.digit[x][y], node_left.digit[x][y - 1]);
    if (is_expandable(node_left)) {
        dist_left = Distance(node_left, dest.digit);
        node_left.index = index;
        node_left.dist = dist_left;
        node_left.dep = node_v[index].dep + 1;
        node_v.push_back(node_left);
    }
}
```

```
//向右移动的子节点进入open表
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 (is_expandable(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);
    }
}
```

Distance 使用曼哈顿距离计算启发函数

```
//曼哈顿距离计算h(n)
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;
}
```

A*算法流程图：



4. 双向广度优先搜索（DBFS）

双向广度搜索是同时从初始节点和目标节点开始搜索，如果它们的路径发生了交汇点就说明找到了路径。这种方式比广度搜索所用时间减少一半。

三、测试案例

Case1 ： src：2 7 3 6 4 5 8 0 1　dest：2 7 3 0 4 5 6 8 1

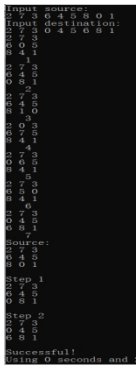
Case2 ： src：0 5 3 2 7 6 1 8 4　dest：1 2 3 8 5 6 7 0 4

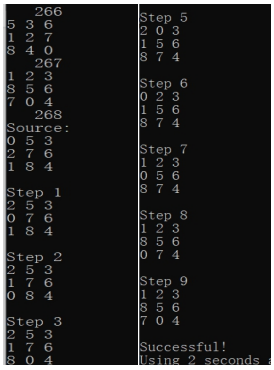
Case3 ： src：8 3 6 7 5 2 1 0 4　dest：5 1 3 8 7 6 2 0 4

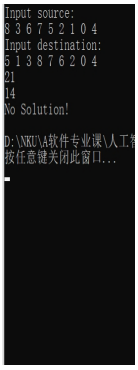
四、结果检验

1. 步数比较

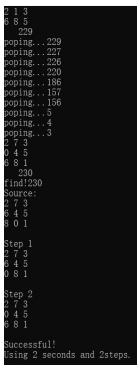
BFS：Case1 Case2 Case3

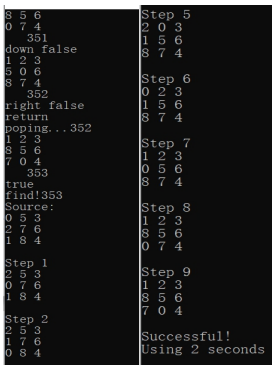


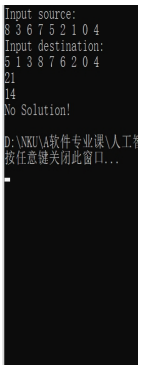




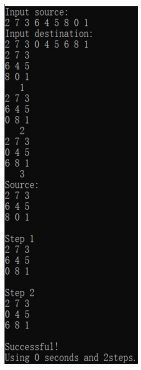
DFS: Case1 Case2 Case3

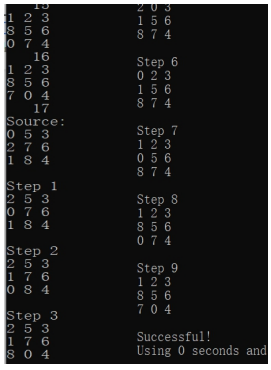


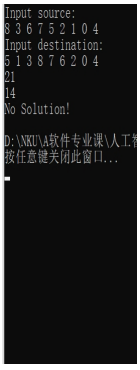




A*： Case1 Case2 Case3







Case\algorithm	BFS	DFS	A*
1 min/all	2/7	2/230	2/3
2 min/all	9/268	9/353	9/17
3 min/all	Null	Null	Null

2. 使用时间

Case\algorithm	BFS	DFS	A*
1	0s	2s	0s
2	2s	2s	0s
3	Null	Null	Null

从上面两个表中可以看出，A*算法在搜索步数和所用时间上明显优于其他算法，这也说明启发式搜索比深度搜索和广度搜索两种方法效率更高。

五、十五数码扩展

问题说明：

十五数码问题同八数码问题，是人工智能中一个很典型的智力问题。十五数码问题是在 4*4 方格盘上，放有 15 个数码，剩下一个位置为空，每一空格其上下左右的数码可移至空格。给定初始位置和目标位置，要求通过一系列的数码移动，将初始状态转化为目标状态。

问题分析：

将八数码中的 A*算法扩展到十五数码问题。八数码和十五数码两个问题的区别主要在于判断是否有解，其次则在于估价函数的选取。

对于 N*N 数码问题，在使用逆序数判断两个状态是否可达时，有如下结论：

当 N 为奇数时，两个 N 数码的逆序数奇偶性相同时，可以互达，否则不行；

当 N 为偶数时，两个 N 数码的逆序数奇偶性相同且它们的 0 所在行的差值也是偶数，或两个 N 数码的逆序数奇偶性不同且它们的 0 所在行差值为奇数，才能互达。

证明如下：

当 0 左右移动时

这个 N 数码的逆序数是不变的；

当 0 上下移动时，

当 N 为奇数时，上下移动时，中间有 N-1 个数，N-1 为偶数，那么整个

N 数码的逆序数只会有两种可能：加减一个偶数。举例说明：当 N 为 3 时，N-1 为 2，当 0 上下移动时，中间的两个数的逆序数 有三种可能，同时加 1 或同时减 1 ，或一个加 1，一个减 1，这三种情况都使得总体的逆序数增加或减少偶数个，所以不管上下移几次，总体的逆序数的奇偶性是不变得；

当 N 为偶数时，上下移动时，中间有 N-1 个数，N-1 为奇数，上下移动一次整个 N 数码的逆序数只会有两种情况：加上或减去一个奇数。举例说明：当 N 为 4 时，N-1 为 3，当 0 上下移动时，中间的 3 个数的逆序数有四种情况，0 个增加 1、3 个减少 1，1 个增加 1、2 个减少 1，2 个增加 1，1 个减少 1，3 个增加 1、0 个减少 1，这四种情况全部都是 使全部的逆序数增加或减去一个奇数。

而估价函数 f=g+h 中，h 可以有多种选取方式。

h(n) = 状态 n 与目标状态不同的元素个数

效果极差，十五数码问题的状态空间树要远复杂于八数码问题，且十五数码问题中空白块的移动更为复杂，此评估函数不适用。

h(n) = 状态 n 与目标状态各个位置数字偏差的绝对值

因为下部数字较大，移动后差值较大造成评估值较大，因此搜索集中在了数值较小的部分，效果很差。

h(n) = 状态 n 与目标状态各个元素的曼哈顿距离

效果较理想。

算法设计：

为了便于扩展，八数码并未使用一元数组存储，因此不必考虑康托展开的区别，仅需对是否有解函数 is_access()按上面的结论进行修改，估价函数本身使用的就是曼哈顿，因此不必改动其他部分。由于其他方法计算量过大，这里只做 A*算法的拓展

判断是否有解条件在原来的基础上增加了计算行号的函数 is_minus(),若两行号差为偶数，返回 true，否则返回 false。

```
//空位所在行号的差
bool is_minus(int digit_src[][COL], int digit_dest[][COL]) {
    int x1 = 0, x2 = 0;
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            if (digit_src[i][j] == 0) {
                x1 = i;
                break;
            }
        }
    }
    for (int i = 0; i < ROW; i++) {
        for (int j = 0; j < COL; j++) {
            if (digit_dest[i][j] == 0) {
                x2 = i;
                break;
            }
        }
    }
    if ((abs(x1 - x2) % 2) == 0) {
        return true;
    }
    else {
        return false;
    }
}
```

Main 函数中无解的判断条件

```
if ((!is_access(_src, _dest)&&is_minus(src.digit,dest.digit))
    ||(is_access(_src, _dest) && !is_minus(src.digit, dest.digit))) {
    cout << "No Solution!" << endl;
    return -1;
}
```

测试案例：

Case1: src: 2 5 7 13 3 0 8 12 15 1 4 10 9 11 14 6

dest: 2 7 0 13 3 5 8 12 15 1 4 10 9 11 14 6

Case2: src: 8 3 5 0 4 12 9 10 1 7 15 14 11 6 13 2

dest: 8 3 0 5 9 12 7 10 4 15 1 14 11 6 13 2

Case3: src: 8 3 5 0 4 12 9 10 1 7 15 14 11 6 13 2

dest: 8 3 5 4 9 12 7 1 10 0 11 6 14 15 13 2

结果检验：

Case\index	min	all	time
1	2	3	0s
2	15	128	1s
3	Null	Null	Null