CS203B Project Report

小组成员：张旭东，刘焱钰，蒋纬搏

1. 系统设计

1.1 系统整体结构

我们将主要程序分为两部分，分别是 **Node** 类与 **HuaRongDao**。其中 **Node** 类用于储存棋盘状态并搜索求解，而 **HuaRongDao** 类则负责 GUI 显示。

```
public class Node implements Comparable{
    //棋盘长宽与棋盘
    int n;
    int m;
    int[][] board = new int[n][m];

    //0的个数与0的位置，使用数组储存
    final int neoCount;
    int[] x0;
    int[] y0;

    //每个棋盘的父节点
    Node father;

    //评价参数，其中F = G + H
    int f;
    int g;
    int h;

    //用于储存移动步骤
    String movement;

    //block数量
    int numberOfblock;
    //行数代表该种block的种类
    int[][] block1;//1*2的block,以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
    int[][] block2;//2*1的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
    int[][] block3;//2*2的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
```

在这些参数中，Node **father** 用于链表的构建。以棋盘状态与评价参数为依据，按照移动步骤将不同棋盘链接在一起。我们使用了 Stack 栈，Priority Queue 优先队列与 HashMap 哈希表，来实现对不同状态的快速搜索与判断。

```
//ArrayList<String> close = new ArrayList<String>();//判断是否在close中
Map<int[][], String> close = new HashMap<>();
//ArrayList<String> openJudge = new ArrayList<String>();//判断是否在open中
Map<int[][], String> openJudge = new HashMap<>();

PriorityQueue<Node> open = new PriorityQueue<~>();//要搜索的状态


java.util.Stack<String> move = new java.util.Stack<~>();
java.util.Stack<String> boards = new java.util.Stack<~>();
java.util.Stack<int[][]> output = new java.util.Stack<>();
```

1.2 GUI 设计

```
ublic class HuaRongDao extends JFrame {

   int[][] datas = new int[][]{{1,2,3}, {4,5,6}, {7,8,0},{9,10,11}};
   JButton go;
   JButton reset;
   JPanel jPanel;
   int row;//行数
   int col;//列数
   Stack<int[][]> input;//获取每一步的棋盘
```

这个类继承了 JFrame，定义了成员变量用于定义棋盘的属性：

1. int[][] datas：存储当前棋盘上的棋子，并初始化。

2. JButton go：展示下一步的按钮

3. JButton reset: 随机生成可解的棋盘

4. JPanel jPanel: 面板

5. int row: 棋盘的行数

6. int col: 棋盘的列数

7. Stack<int[][]> input: 获取每一步的棋盘

```
public void setDatas(int[][] datas) { this.datas = datas; }
public void setRow(int row) { this.row = row; }
public void setCol(int col) { this.col = col; }
```

编写方法用于实时更新棋盘：

1. setDatas: 更新棋盘上的棋子位置

2. setRow: 更新棋盘的行数

3. setCol: 更新棋盘的列数

```
//编写重绘方法
public void repaint() {
    //移除面板上所有组件
    jPanel.removeAll();

    //重新绘制画板内容
    for (int i = 0; i < datas.length; i++) {
        for (int j = 0; j < datas[i].length; j++) {
            JLabel lb = new JLabel(new ImageIcon( filename: "Image\\" + datas[i][j] + ".png"));
            lb.setBounds( x: 90 * j,  y: 90 * i,  width: 90,  height: 90);
            jPanel.add(lb);
        }
    }
    this.add(jPanel);
    //添加到画板上
    jPanel.repaint();
}
```

编写重绘方法便于每次获取棋盘后，对棋盘进行更新：先移除面板上所有的组件，再用两个 for 循环嵌套按照数组对应索引下的数字添加对应棋子的图片，并设置每个图片的大小，然后再将图片添加到面板中。

```
//显示窗体方法
public void initFrame() {
    this.setTitle("数字华容道");
    this.setSize( width: 960,  height: 765);
    this.setDefaultCloseOperation(3);
    this.setLayout(null);
    this.setLocationRelativeTo(null);
    this.setAlwaysOnTop(true);
}
```

显示窗体方法：设置标题、大小、关闭方式等。

```java
public void paintView() {

    //标题设置
    JLabel title = new JLabel( text: "Numeric Ktloski");
    title.setBounds( x: 354,  y: 27,  width: 232,  height: 57);
    this.add(title);

    //游戏面板设置
    jPanel = new JPanel();
    jPanel.setBounds( x: 150,  y: 114,  width: 480,  height: 480);
    jPanel.setLayout(null);
    //把图片放进去
    for (int i = 0; i < datas.length; i++) {
        for (int j = 0; j < datas[i].length; j++) {
            JLabel picture = new JLabel(new ImageIcon( filename: "Image\\" + datas[i][j] + ".png"));
            picture.setBounds( x: 90 * j,  y: 90 * i,  width: 90,  height: 90);
            jPanel.add(picture);
        }
    }
    this.add(jPanel);


    //按钮
    go = new JButton( text: "go");
    go.setBounds( x: 626,  y: 344,  width: 108,  height: 45);
    this.add(go);
    reset = new JButton( text: "reset");
    reset.setBounds( x: 786,  y: 344,  width: 108,  height: 45);
    this.add(reset);

    //背景设置
    JLabel background = new JLabel();
    background.setBounds( x: 0,  y: 0,  width: 960,  height: 530);
    this.add(background);
}
```

组件设置：设置标题以及大小，设置面板的大小以及布局方式，将图片按照棋盘的顺序放进去；设置两个按钮 go 和 reset 以及两个按钮的位置及大小并添加到窗体中；设置背景图片，添加到窗口（但此处并没有真的添加背景图片）。

```java
//编写方法给按钮添加事件
public void addButtonEvent() {
   reset.addActionListener(e -> {
        upset();
    });
    go.addActionListener(e -> {
        //TODO:算法部分
        setDatas(input.pop());
        repaint();
        if(input.isEmpty()){
            View();
        }
    });
}
```

编写方法给按钮 go 和 reset 添加事件：

1. go: 读取下一步的棋盘，调用 repaint 方法重绘棋盘，如果全部读取完，则游戏胜利。

2. reset: 调用 upset 方法重置棋盘。

```java
public void upset() {
    int count = 1;
    Random rd = new Random();
    int n = 3;
    int m = 3;
    int[][] newData = new int[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            newData[i][j] = count;
            count++;
        }
    }
    newData[n - 1][m - 1] = 0;
    int ci = HuaRongDao.inverseNum(newData);

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            int x = rd.nextInt(n);
            int y = rd.nextInt(m);
            int temp = newData[i][j];
            newData[i][j] = newData[x][y];
            newData[x][y] = temp;
        }
    }
```

```java
    while (HuaRongDao.inverseNum(newData) % 2 != ci % 2){
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                int x = rd.nextInt(n);
                int y = rd.nextInt(m);
                int temp = newData[i][j];
                newData[i][j] = newData[x][y];
                newData[x][y] = temp;
            }
        }
    }
    datas = newData;
    Node data = new Node(datas, father: null, numberOfBlock: 0, allOfBlock: null);
    this.input = Node.answer(data);
    input.pop();
    repaint();
}
```

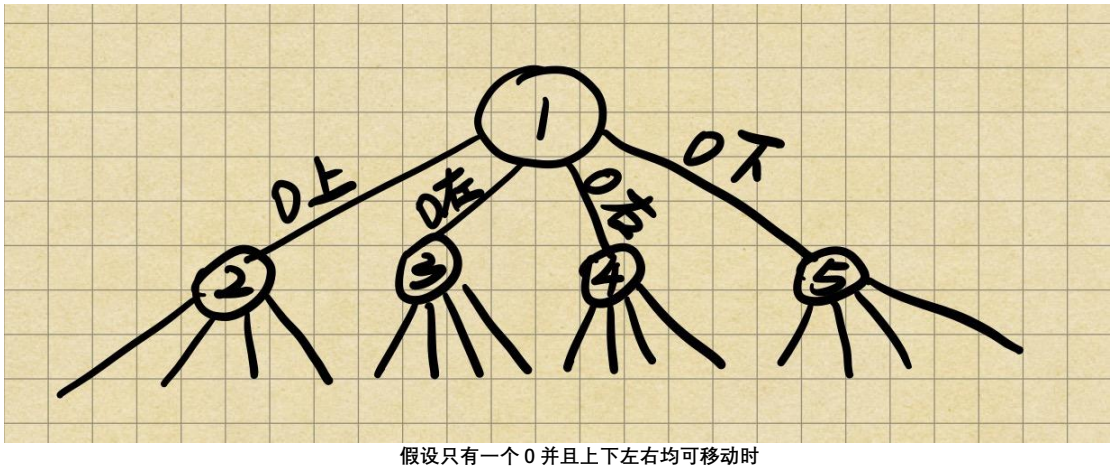
编写 upset 方法随机生成可解的棋盘：先生成一个顺序的棋盘，再将其打乱，判断其逆序数，如不可解则交换相邻的非零数，输出此棋盘。

```java
//构造方法
public HuaRongDao(Stack<int[][]> input){
    this.input = input;
    //判断input是否为空；为空则无解
    if(input.isEmpty()){
        JOptionPane.showMessageDialog( parentComponent: this,  message: "No solution");
        System.exit( status: 0);
    }
    //传入初始棋盘
    setDatas(this.input.pop());
    //窗体初始化设置
    initFrame();
    //窗体上的组件设置显示
    paintView();
    //添加按钮事件
    addButtonEvent();
    //设置窗体可见
    this.setVisible(true);
    //设置棋盘大小
    setRow(datas.length);
    setCol(datas[0].length);
}
```

这是一个构造方法。传入存储棋盘解法的 Stack<int[][]> input，如果为空，那么此棋盘无解，并弹出一个窗口告知此棋盘无解，然后关闭此程序；如果非空，传入初始棋盘，初始化窗体，显示窗体组件，添加按钮事件，并设置窗口可见。

1.3 求解算法基本过程

我们使用了 **Node** 类中的参数 **board** 二维数组储存棋盘，使用树的结构来储存步骤。每一个棋盘所指向 **father** 节点就是其上一个步骤。我们以 0 的移动作为判断标准来绘制创建新的 **board** 并将新的情况储存为全新的 Node，储存进 open 优先队列中，将棋盘的特征值储存为 String 放入名为 **openJudge** 的 Map 中用于判断该棋盘是否已经被遍历过。



假设只有一个 0 并且上下左右均可移动时

首先，程序将最开始的状态也就是标准输入，转化为 **Node** 类型，利用函数 *getH*()计算他的评价函数$f = g + h$（h 计算方式见下图）

```java
public int getH(){
    int mistake = 0;
    int[] total = new int[n * m];
    for (int i = 0; i < n * m; i++) {
        total[i] = this.board[i / board[0].length][i % board[0].length];
    }
    for (int i = 0; i < total.length - neoCount; i++) {
        if(total[i] != i + 1){
            mistake++;
        }
    }
    for (int i = total.length - neoCount; i < total.length; i++) {
        if (total[i] != 0){
            mistake++;
        }
    }
    return mistake;
}
```

```java
@Override
public int compareTo(Object o) {
    if (o instanceof Node) {
        Node node = (Node) o;
        if (this.f > node.f) {
            return 1;
        }
        else if (this.f < node.f) {
            return -1;
        }
        else if (this.h >= node.h){
            return 1;
        }
        else{
            return -1;
        }
    }
    else {
        throw new RuntimeException("传入的数据类型不一致");
    }
}
```

左图为 *getH*()函数，用于计算 Node 的评价函数$f = g + h$（f =已走过步数+预估步数）
右图为重写 *compareTo*()函数，用于 open 优先队列弹出 Node 时调用

其中优先队列 open 会优先弹出 **f** 值（也就是评价函数）最小的 **Node**，判断其是否已经完成。若并未完成，那么该节点会生成子节点（也就是所有可能的下一步骤）并储存为新的 **Node** 放入 **open** 与 **openJudge** 中。随后该节点会被标记为已被探索并加入 **close** 中。

在生成子节点时，程序使用 *findNext*()对新发现的子节点进行判断。判断该节点是否已经被探索或者是已经被找到过（也就是是否存在于 **openJudge** 或 **close** 映射中），如果该节点已经被探索过（已经在 **close** 中），那么该节点会被跳过。如果该节点未被探索，但已经被发现过（在 **openJudge** 中），那将会生成一个新的节点并更新其的 f 值再将其放入 **open** 与 **openJudge** 中。

```java
//判断是否已经完成
public boolean isComplete(){
    int mistake = 0;
    int[] total = new int[n * m];
    for (int i = 0; i < n * m; i++) {
        total[i] = this.board[i / board[0].length][i % board[0].length];
    }
    for (int i = 0; i < total.length - neoCount; i++) {
        if(total[i] != i + 1){
            mistake++;
        }
    }
    for (int i = total.length - neoCount; i < total.length; i++) {
        if (total[i] != 0){
            mistake++;
        }
    }
    return mistake == 0;
}
```

```java
//是否已被探索
if (!close.containsValue(n2)){
    //判断属于哪种类型的block，如果是block，则需要打印左上角的数值和动作
    boolean flagofblock1 = find(currentState.block1, currentState.board[x + 1][y]);
    boolean flagofblock2 = find(currentState.block2, currentState.board[x + 1][y]);
    boolean flagofblock3 = find(currentState.block3, currentState.board[x + 1][y]);

    if(flagofblock1==false&&flagofblock2==false&&flagofblock3==false){
        //属于1*1，直接打印就行
        next2.movement = currentState.board[x + 1][y] + " U";
        //是否已被发现
        if (openJudge.containsValue(n2)) {
            open.offer(next2);
        }
        else {
            open.offer(next2);
            openJudge.put(next2.board, n2);
        }
    }
```

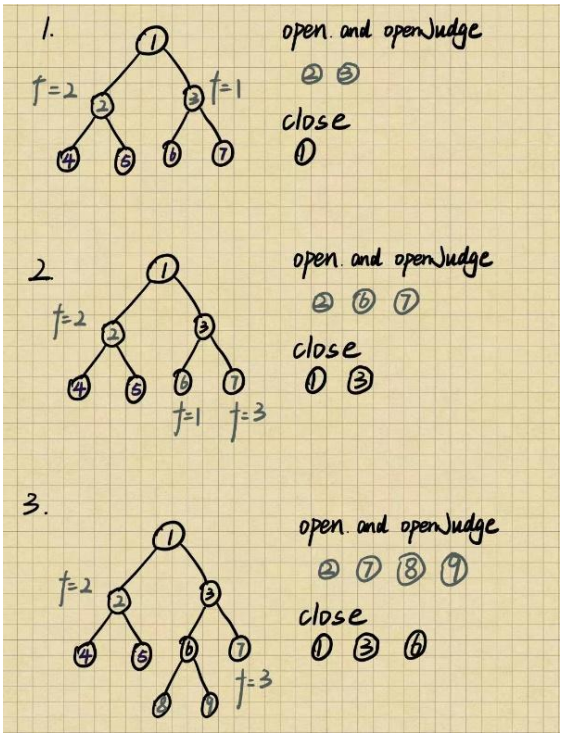
左图为判断棋盘是否已被完成
右图展示判断 0 是否可以移动并判断新生成的情况是否已被探索或是发现

该循环完成后，将再次从优先队列 **open** 中弹出下一 **Node** 并重复循环，直至 **Node** 被判断为已完成或是 **open** 队列为空，则此时中断循环并生成结果。

1.4 算法解析

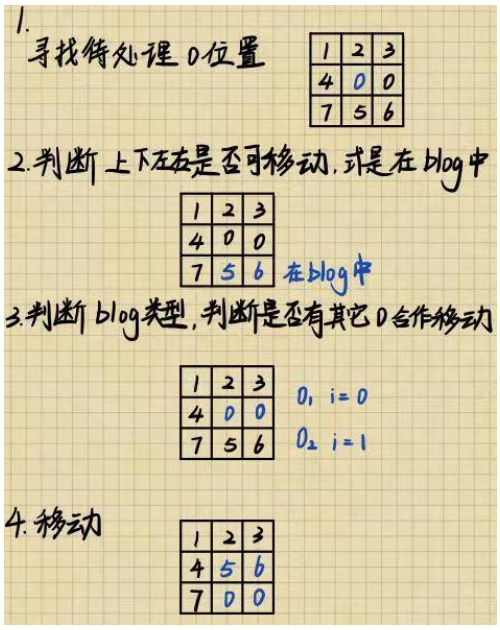
该算法主要参考了 ***A*算法***，在原有的普通广度优先搜索的基础上加入了**评价函数**，也就是$\boldsymbol{f = g + h}$。这使得程序在对 **open** 中的待处理节点进行处理时，不再简单的无序或是按先入后出顺序，而是按照 **f** 值大小优先弹出局部最优的步骤这使得程序不用对树状结构中的每一个 Node 进行处理，而是更有针对性地选择更有可能的路径。

由右图可知在这个过程中，遍历算法不再简单的按照树状结构的层数进行遍历，而是利用对评价函数的大小评判来决定下一个处理状态。



1.5 *findNext*()函数

该函数用于寻找某一结点的所有子节点，并判断其是否应该放置于 **open** 映射中。该函数首先使用保存有 0 的坐标的数组 **int[] x0** 与 **int[] y0** 寻找该节点棋盘中 0 的位置，并以此判断该 0 的上下左右能否移动，若可以移动则进一步判断其上下左右是否包含在 blog 中（该步骤通过储存有 blog 类型与 blog 中数字的数组遍历实现）。若可以移动，则实现棋盘的变化并计算该 **Node** 的评价函数 **f**。将改步骤储存在 String **movement** 中以便最后的得到结果后回溯并标准输出步骤。



```
//block数量
int numberOfblock;
//行数代表该种block的种类
int[][] block1;//1*2的block,以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
int[][] block2;//2*1的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
int[][] block3;//2*2的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
```

储存 blog 类型与 blog 中的数字

```
public static Node up(Node state,int x,int y,int i){...}

public static Node down(Node state,int x,int y,int i){...}

public static Node left(Node state,int x,int y,int i){...}

public static Node right(Node state,int x,int y,int i){...}
```

判断上下左右是否是 blog 并判断是否可以移动，其中参数 i 为 0 的 index，也就是其在数组 x0 与 y0 中的位置

```
public static int rangeofblock1(Node state,int x,int y){
    //state为目前需要判断的状态，[x,y]为需要判断的点在state.array中的坐标
    int column=0;
    for(int i=0;i<state.block1.length;i++){
        for(int j=0;j<state.block1[0].length;j++){
            if(state.block1[i][j]==state.board[x][y]){
                column=j;
            }
        }
    }
    if(column==0){
        return 1;
    }else{
        return 2;
    }
}

public static int rangeofblock2(Node state,int x,int y){...}

public static int rangeofblock3(Node state,int x,int y){...}
```

判断该数字在 blog 中的哪个位置（由不同 blog 类型调用不同的 rangeofblog）

2.算法效率对比

我们最开始采用的方法是广度优先搜索。每一种不同的数字排序被看作是一种状态。 每一种状态包含 12 个参数:行数 **n** ，列数 **m**，和一个用来表示棋盘的二位数组 **board** ，棋盘中空格（也就是 0）的个数 **neoCount** , 一个指向上一个棋盘的 Node father，两个用来储存多个 0 位置的一维数组 **x0** 与 **y0**，棋盘中的 blog 个数 **numberOfblock** , 三个二维数组 **block1**，**block2**，**block3** 用于储存不同类型 block 中的数字还有一个 String **movement** 用于记录步骤以便标准输出。

```
int n;
    int m;
    int[][] board = new int[n][m];
    final int neoCount;
    Node father;
    int[] x0;
    int[] y0;


    //block
    int numberOfblock;
    //行数代表该种block的种类
    int[][] block1;//1*2的block,以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
    int[][] block2;//2*1的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
    int[][] block3;//2*2的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素


    String movement;
```

一个 Map **close** 用来储存已经被探索过的状态，PriorityQueue **open** 和 Map **openJudge** 共同储存已经被发现但还未被探索的状态。在棋盘移动时，**open** 队列中会依照评价函数大小弹出一个 Node，随后对这个 Node 进行操作，找出其子节点并将其存入 **open** 中，最后将该节点自身存入 **close**。单纯的***广度优先搜索***和改进后的 ***A*算法***实际上十分相似，下面是二者的运行时间对比. 输入的样本如下：

Sample Input 1

```
2 2
1 2
0 3
0
```

Sample Input 2

```
2 2
2 1
0 3
0
```

Sample Input 3

```
3 3
1 2 3
4 0 0
7 5 6
1
5 1*2
```

Sample Input 4

```
5 4
3 7 4 8
1 2 0 11
5 6 15 10
9 14 0 12
13 17 0 16
2
12 2*1
1 2*2
```

The following are the time used for the above 4 inputs using breadth-first search:

```
D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> d: && cd "d:\Study in SUSTech\First semester of junior ye
ar\dsaaB\lab\project\Numeric_Klotski" && cmd /C ""C:\Program Files\Java
\jdk1.8.0_271\bin\java.exe" -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc4
2xism.jar Breadth_First.Run "
2 2
1 2
0 3
0
Yes
1
```

```
3 L
6080 ms

D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> d: && cd "d:\Study in SUSTech\First semester of junior ye
ar\dsaaB\lab\project\Numeric_Klotski" && cmd /C ""C:\Program Files\Java
\jdk1.8.0_271\bin\java.exe" -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc4
2xism.jar Breadth_First.Run "
2 2
2 1
0 3
0
No
15479 ms

D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> d: && cd "d:\Study in SUSTech\First semester of junior ye
ar\dsaaB\lab\project\Numeric_Klotski" && cmd /C ""C:\Program Files\Java
\jdk1.8.0_271\bin\java.exe" -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc4
2xism.jar Breadth_First.Run "
3 3
1 2 3
4 0 0
7 5 6
1
5 1*2
Yes
1
5 U
31883 ms
```

第四个输入样例因运行时间过久无法得到结果。

以下是 ***A*算法***的运行结果：

```
D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> d: && cd "d:\Study in SUSTech\First semester of junior ye
ar\dsaaB\lab\project\Numeric_Klotski" && cmd /C ""C:\Program Files\Java
\jdk1.8.0_271\bin\java.exe" -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc4
2xism1.jar Node_and_Main.Run "
2 2
1 2
0 3
0
Yes
1
3 L
4709 ms
```

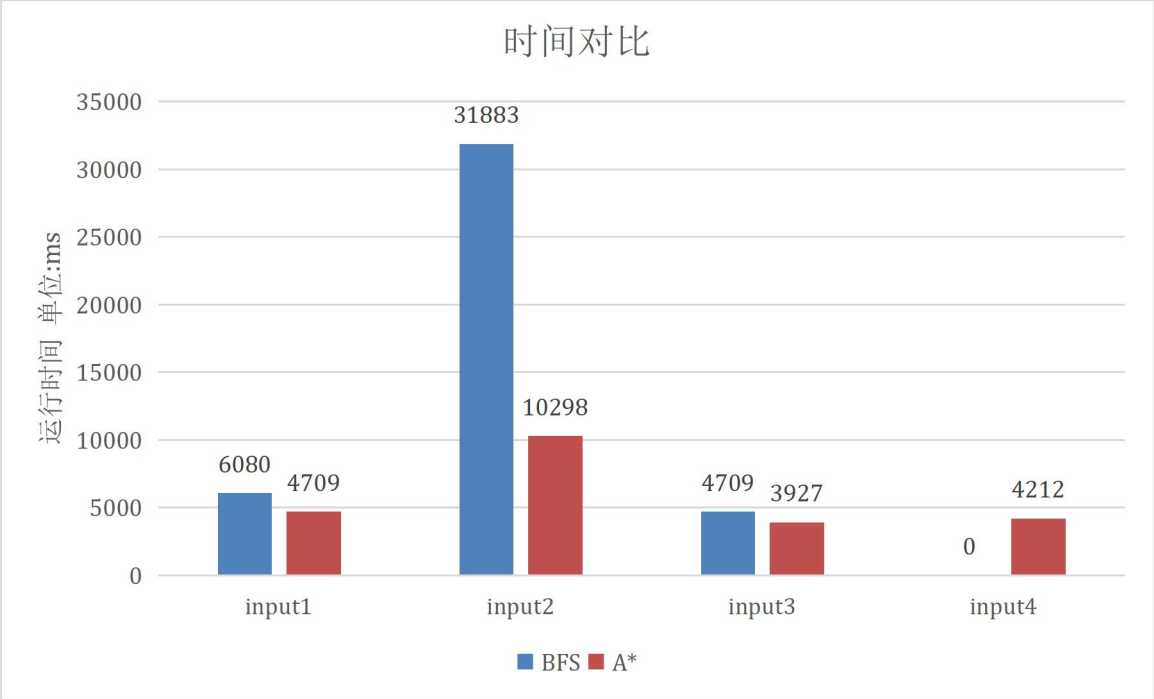
```
D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> cmd /C ""C:\Program Files\Java\jdk1.8.0_271\bin\java.exe"
 -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc42xism.jar Node_and_Main.Run
 "
2 2
2 1
0 3
0
No
10298 ms

D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> cmd /C ""C:\Program Files\Java\jdk1.8.0_271\bin\java.exe"
 -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc42xism.jar Node_and_Main.Run
 "
3 3
1 2 3
4 0 0
7 5 6
1
5 1*2
Yes
1
5 U
3972 ms

D:\Study in SUSTech\First semester of junior year\dsaaB\lab\project\Num
eric_Klotski> cmd /C ""C:\Program Files\Java\jdk1.8.0_271\bin\java.exe"
 -cp C:\WINDOWS\TEMP\cp_9jl9qgyvfb3upo2hqbc42xism.jar Node_and_Main.Run
 "
5 4
3 7 4 8
1 2 0 11
5 6 15 10
9 14 0 12
13 17 0 16
2
12 2*1
1 2*2
Yes
16
15 D
10 L
11 D
8 D
4 R
7 R
7 D
3 R
3 R
```

```
1 U
10 L
11 L
12 U
9 U
13 U
17 L
4212 ms
```

按照上面的实时间对比, A*算法的效率明显高于未经优化的广度优先算法. 这是因为广度优先算法只能按照基本的先入先出或者先入后出顺序遍历子节点，而 A*算法可以根据评价函数在每种 state 下优先弹出局部最优解，在有向图寻找中提供了方向性，提前排除了部分局部非优的分支。



完整的广度优先算法版本程序如下:

```
package Breadth_First;
```

```java
import java.util.ArrayList;
import java.util.PriorityQueue;

import edu.princeton.cs.algs4.Queue;

public class Node {
    int n;
    int m;
    int[][] board = new int[n][m];
    final int neoCount;
    Node father;
    int[] x0;
    int[] y0;


    //block
    int numberOfblock;
    //行数代表该种block的种类
    int[][] block1;//1*2的block,以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
    int[][] block2;//2*1的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素
    int[][] block3;//2*2的block，以二维数组的形式记录，行数代表该种block的个数，每行的元素代表该block中的元素


    String movement;


    //初始化Node
    public Node(int n, int m, int neoCount, Node father){
        this.n = n;
        this.m = m;
        this.board = new int[n][m];
        this.neoCount = neoCount;
        this.father = father;
        this.x0 = new int[neoCount];
        this.y0 = new int[neoCount];

        this.numberOfblock=father.numberOfblock;
        this.block1=new int[n*m/2][2];
        this.block2=new int[n*m/2][2];
        this.block3=new int[n*m/4][4];



    }

    public Node(int[][] board, Node father,int numberOfblock,int[][] allOfBlock){
```

```
        int count = 0;
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if(board[i][j] == 0){
                    count++;
                }
            }
        }
        this.neoCount = count;
        int neo = 0;
        this.n = board.length;
        this.m = board[0].length;
        int[][] array = new int[board.length][board[0].length];
        for(int i = 0; i < board.length; i++){
            System.arraycopy(board[i], 0, array[i], 0, board[0].length);

        }
        this.board = array.clone();
        this.father = father;
        this.x0 = new int[count];
        this.y0 = new int[count];
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                if(board[i][j] == 0){
                    x0[neo] = i;
                    y0[neo] = j;
                    neo++;
                }
            }
        }

        //block 的一些初始化
        this.numberOfblock=numberOfblock;
        this.block1=new int[n*m/2][2];
        this.block2=new int[n*m/2][2];
        this.block3=new int[n*m/4][4];

        //根据allOfBlock 中每行第二列元素来判断属于哪种block，并将其存入相
应的block 中
        int numberOfblock1=0;
        int numberOfblock2=0;
        int numberOfblock3=0;

        //allofblock 为numofblock*2 的二维数组，第二列代表block 的种类，第
一列代表block 中左上角的数字，
        if(numberOfblock>0){
            for(int i=0;i<numberOfblock;i++){
                //判断是否属于第一种block 1*2
                if(allOfBlock[i][1]==1){
```

```
                    //将属于该block 的值存入其中
                    this.block1[numberOfblock1][0]=allOfBlock[i][0];
                    this.block1[numberOfblock1][1]=board[coordinateOfTo
pInBlock(board, allOfBlock[i][0])[0]][coordinateOfTopInBlock(board, all
OfBlock[i][0])[1]+1];
                    numberOfblock1=numberOfblock1+1;

                }
                //判断是否属于第二种block 2*1
                if(allOfBlock[i][1]==2){
                    //将属于该block 的值存入其中
                    this.block2[numberOfblock2][0]=allOfBlock[i][0];
                    this.block2[numberOfblock2][1]=board[coordinateOfTo
pInBlock(board, allOfBlock[i][0])[0]+1][coordinateOfTopInBlock(board, a
llOfBlock[i][0])[1]];
                    numberOfblock2=numberOfblock2+1;

                }
                //判断是否属于第二种block 2*2
                if(allOfBlock[i][1]==3){
                    //将属于该block 的值存入其中
                    this.block3[numberOfblock3][0]=allOfBlock[i][0];
                    this.block3[numberOfblock3][1]=board[coordinateOfTo
pInBlock(board, allOfBlock[i][0])[0]][coordinateOfTopInBlock(board, all
OfBlock[i][0])[1]+1];
                    this.block3[numberOfblock3][2]=board[coordinateOfTo
pInBlock(board, allOfBlock[i][0])[0]+1][coordinateOfTopInBlock(board, a
llOfBlock[i][0])[1]];
                    this.block3[numberOfblock3][3]=board[coordinateOfTo
pInBlock(board, allOfBlock[i][0])[0]+1][coordinateOfTopInBlock(board, a
llOfBlock[i][0])[1]+1];
                    numberOfblock3=numberOfblock3+1;
                }
            }
        }
    }

    //复制Node
    public Node(Node node){
        this.n = node.n;
        this.m = node.m;
        int[][] array=new int[node.board.length][node.board[0].length];

        this.x0 = new int[node.x0.length];
        this.y0 = new int[node.y0.length];
        for(int i = 0; i < node.board.length; i++){
            System.arraycopy(node.board[i], 0, array[i], 0, node.board
[0].length);
        }
```

```
        this.board = array.clone();
        this.neoCount = node.neoCount;
        this.father = node;
        System.arraycopy(node.x0, 0, this.x0, 0, node.x0.length);
        System.arraycopy(node.y0, 0, this.y0, 0, node.y0.length);


        this.numberOfblock=node.numberOfblock;

        //先判断各种block 数组是否为空，不为空就拷贝
        //判断第一种block 1*2 是否为空
        if(node.block1!=null&&node.block1.length!=0&&node.block1[0].len
gth!=0){
            int[][] array1=new int[node.block1.length][node.block1[0].l
ength];
            for(int i = 0; i < node.block1.length; i++){
                System.arraycopy(node.block1[i], 0, array1[i], 0, node.
block1[0].length);
            }
            this.block1 = array1.clone();
        }
        //判断第二种block 2*1 是否为空
        if(node.block2!=null&&node.block2.length!=0&&node.block2[0].len
gth!=0){
            int[][] array2=new int[node.block2.length][node.block2[0].l
ength];
            for(int i = 0; i < node.block2.length; i++){
                System.arraycopy(node.block2[i], 0, array2[i], 0, node.
block2[0].length);
            }
            this.block2 = array2.clone();

        }
        //判断第三种block 2*2 是否为空
        if(node.block3!=null&&node.block3.length!=0&&node.block3[0].len
gth!=0){
            int[][] array3=new int[node.block3.length][node.block3[0].l
ength];
            for(int i = 0; i < node.block3.length; i++){
                System.arraycopy(node.block3[i], 0, array3[i], 0, node.
block3[0].length);
            }
            this.block3 = array3.clone();

        }



    }
```

```
    public static String printState(Node state){
        StringBuilder output = new StringBuilder();
        for(int i=0;i<state.n;i++){
            for(int j=0;j<state.m;j++){
                output.append(state.board[i][j]).append(" ");
            }
            output.append("\n");
        }
        output.append("\n");
        return output.toString();
    }

    public static String arrayToString(Node state){
        String s="";
        for(int i =0;i<state.n;i++){
            for(int j =0;j<state.m;j++){
                s=s.concat(String.valueOf(state.board[i][j]));//此处可能
有问题
            }
        }
        return s;
    }

    public boolean isComplete(){
        int mistake = 0;
        int[] total = new int[n * m];
        for (int i = 0; i < n * m; i++) {
            total[i] = this.board[i / board[0].length][i % board[0].len
gth];
        }
        for (int i = 0; i < total.length - neoCount; i++) {
            if(total[i] != i + 1){
                mistake++;
            }
        }
        for (int i = total.length - neoCount; i < total.length; i++) {
            if (total[i] != 0){
                mistake++;
            }
        }
        return mistake == 0;
    }

    public void findNext(Queue<Node> queue, ArrayList<String> close){//
queue 代表为搜索的状态，close 代表为已经搜索过的状态
        Node currentState = this;
        close.add(arrayToString(currentState));

        for (int i = 0; i < currentState.x0.length; i++) {
```

```
            int x = currentState.x0[i];
            int y = currentState.y0[i];


            //上面能不能换
            Node next1= up(currentState, x, y, i);
            //以下是判断移动的那个数字属于哪种block，属于block则需要打印该
block中最小值及动作
            if (!close.contains(arrayToString(next1))){
                //判断属于哪种类型的block，如果是block，则需要打印左上角的
数值和动作
                boolean flagofblock1 = find(currentState.block1, curren
tState.board[x - 1][y]);
                boolean flagofblock2 = find(currentState.block2, curren
tState.board[x - 1][y]);
                boolean flagofblock3 = find(currentState.block3, curren
tState.board[x - 1][y]);

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==false){
                    //属于1*1，直接打印就行
                    next1.movement = currentState.board[x - 1][y] + " D
";
                    queue.enqueue(next1);

                }

                if(flagofblock1==true&&flagofblock2==false&&flagofblock
3==false){
                    //属于1*2，找左上角的值，并打印以及相应动作
                    int j=rangeofblock1(currentState, x-1, y);
                    if(j==1){//代表就是左上角的值
                        next1.movement = currentState.board[x - 1][y] +
 " D";
                        queue.enqueue(next1);
                    }
                    else{//在它左边的数字就是左上角的值
                        next1.movement = (currentState.board[x - 1][y-
1]) + " D";
                        queue.enqueue(next1);
                    }

                }

                if(flagofblock1==false&&flagofblock2==true&&flagofblock
3==false){
                    //属于2*1，找左上角的值，并打印以及相应动作
                    next1.movement = currentState.board[x - 2][y] + " D
";
```

```
                    queue.enqueue(next1);
                }

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==true){
                    //属于2*2，找左上角的值，并打印以及相应动作
                    int j=rangeofblock3(currentState, x-1, y);
                    if(j==3){//代表就是左下角的值
                        next1.movement = currentState.board[x - 2][y] +
 " D";
                        queue.enqueue(next1);
                    }
                    else{//j==4,代表是右下角的值
                        next1.movement = (currentState.board[x - 2][y-
1]) + " D";
                        queue.enqueue(next1);
                    }

                }




            }

            //下边能不能换
            Node next2=down(currentState, x, y, i);

            //以下是判断移动的那个数字属于哪种block，属于block则需要打印该
block中最小值及动作
            if (!close.contains(arrayToString(next2))){
                //判断属于哪种类型的block，如果是block，则需要打印左上角的
数值和动作
                boolean flagofblock1 = find(currentState.block1, curren
tState.board[x + 1][y]);
                boolean flagofblock2 = find(currentState.block2, curren
tState.board[x + 1][y]);
                boolean flagofblock3 = find(currentState.block3, curren
tState.board[x + 1][y]);

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==false){
                    //属于1*1，直接打印就行
                    next2.movement = currentState.board[x + 1][y] + " U
";
                    queue.enqueue(next2);
                }
```

```
                if(flagofblock1==true&&flagofblock2==false&&flagofblock
3==false){
                    //属于1*2，找左上角的值，并打印以及相应动作
                    int j=rangeofblock1(currentState, x+1, y);
                    if(j==1){//代表就是左上角的值
                        next2.movement = currentState.board[x + 1][y] +
 " U";
                        queue.enqueue(next2);
                    }
                    else{//在它左边的数字就是左上角的值
                        next2.movement = currentState.board[x + 1][y-1]
 + " U";
                        queue.enqueue(next2);
                    }

                }

                if(flagofblock1==false&&flagofblock2==true&&flagofblock
3==false){
                    //属于2*1，找左上角的值，并打印以及相应动作
                    next2.movement = currentState.board[x + 1][y] + " U
";
                    queue.enqueue(next2);
                }

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==true){
                    //属于2*2，找左上角的值，并打印以及相应动作
                    int j=rangeofblock3(currentState, x+1, y);
                    if(j==1){//代表就是左上角的值
                        next2.movement = currentState.board[x + 1][y] +
 " U";
                        queue.enqueue(next2);
                    }
                    else{//j==2,代表是右上角的值
                        next2.movement = currentState.board[x + 1][y-1]
 + " U";
                        queue.enqueue(next2);
                    }

                }



            }
```

```
            //左边能不能换
            Node next3=left(currentState, x, y, i);

            //以下是判断移动的那个数字属于哪种block，属于block则需要打印该block中最小值及动作
            if (!close.contains(arrayToString(next3))){
                //判断属于哪种类型的block，如果是block，则需要打印左上角的数值和动作
                boolean flagofblock1 = find(currentState.block1, curren
tState.board[x][y-1]);
                boolean flagofblock2 = find(currentState.block2, curren
tState.board[x][y-1]);
                boolean flagofblock3 = find(currentState.block3, curren
tState.board[x][y-1]);

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==false){
                    //属于1*1，直接打印就行
                    next3.movement = currentState.board[x][y - 1] + " R
";
                    queue.enqueue(next3);
                }

                if(flagofblock1==true&&flagofblock2==false&&flagofblock
3==false){
                    //属于1*2，有解的情况下是j==2
                    next3.movement = currentState.board[x][y - 2] + " R
";
                    queue.enqueue(next3);
                }

                if(flagofblock1==false&&flagofblock2==true&&flagofblock
3==false){
                    //属于2*1
                    int j=rangeofblock2(currentState, x, y-1);
                    if(j==1){//代表就是上面的值
                        next3.movement = currentState.board[x][y - 1] +
 " R";
                        queue.enqueue(next3);
                    }
                    else{//代表就是下面的值
                        next3.movement = currentState.board[x-1][y - 1]
 + " R";
                        queue.enqueue(next3);
                    }


                }
```

```
                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==true){
                    //属于2*2，直接打印就行
                    int j=rangeofblock3(currentState, x, y-1);
                    //j 在有解的情况下只有两个值，2 和4
                    if(j==2){//代表就是右上角的值
                        next3.movement = currentState.board[x][y - 2] +
 " R";
                        queue.enqueue(next3);
                    }
                    else{//代表就是右下角的值

                        next3.movement = currentState.board[x-1][y - 2]
 + " R";
                        queue.enqueue(next3);

                    }
                }

            }




            //右边能不能换
            Node next4=right(currentState, x, y, i);


            //以下是判断移动的那个数字属于哪种block，属于block 则需要打印该
block 中最小值及动作
            if (!close.contains(arrayToString(next4))){
                //判断属于哪种类型的block，如果是block，则需要打印左上角的
数值和动作
                boolean flagofblock1 = find(currentState.block1, curren
tState.board[x][y+1]);
                boolean flagofblock2 = find(currentState.block2, curren
tState.board[x][y+1]);
                boolean flagofblock3 = find(currentState.block3, curren
tState.board[x][y+1]);

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==false){
                    //属于1*1，直接打印就行
                    next4.movement = currentState.board[x][y + 1] + " L
";
                    queue.enqueue(next4);

                }
```

```
                if(flagofblock1==true&&flagofblock2==false&&flagofblock
3==false){
                    //属于1*2，有解的情况下就是左上角的值，直接打印就行
                    next4.movement = currentState.board[x][y + 1] + " L
";
                    queue.enqueue(next4);

                }

                if(flagofblock1==false&&flagofblock2==true&&flagofblock
3==false){
                    //属于2*1
                    int j=rangeofblock2(currentState, x, y+1);
                    if(j==1){//代表就是左上角的值
                        next4.movement = currentState.board[x][y + 1] +
 " L";
                        queue.enqueue(next4);

                    }
                    else{//代表就是下面的值
                        next4.movement = currentState.board[x-1][y + 1]
 + " L";
                        queue.enqueue(next4);
                    }

                }

                if(flagofblock1==false&&flagofblock2==false&&flagofbloc
k3==true){
                    //属于2*2
                    int j=rangeofblock3(currentState, x, y+1);
                    //有解的情况下j==1or3
                    if(j==1){//代表就是左上角的值
                        next4.movement = currentState.board[x][y + 1] +
 " L";
                        queue.enqueue(next4);
                    }
                    else{//代表就是左下角的值
                        next4.movement = currentState.board[x-1][y + 1]
 + " L";
                        queue.enqueue(next4);
                    }


                }


            }
        }
    }
```

```
    //将三种类型的字符串转为对应的标识
    public static int transform(String s){
        if(s.equals("1*2")){
            return 1;
        }
        else if(s.equals("2*1")){
            return 2;
        }
        else if(s.equals("2*2")){
            return 3;
        }
        else{
            return 0;//0 代表不支持以外的block
        }

    }

    //获取各种block 中左上角元素在数组中位置的方法
    public static int[] coordinateOfTopInBlock(int[][] array,int number)
{
        //coordinate 第一个元素代表所在行数，第二元素代表所在列数
        int[] coordinate= new int[2];
        for(int i=0;i<array.length;i++){
            for(int j=0;j<array[0].length;j++){
                if(array[i][j]==number){
                    coordinate[0]=i;
                    coordinate[1]=j;
                    break;
                }
            }
        }
        return coordinate;
    }

    //判断上面能不能换及结果
    public static Node up(Node state,int x,int y,int i){
        //state 表示当前状态，[x,y]表示当前选定0 的坐标,i 代表选定的0 在x0,y
0 中的index
        if(x-1>=0&&state.board[x-1][y]!=0){
            Node next = new Node(state);

            //判断属于哪种类型的block
            //需要判断的点的坐标在array 中的坐标是[x-1,y]
            boolean flag1=find(state.block1, state.board[x-1][y]);
            boolean flag2=find(state.block2, state.board[x-1][y]);
```

```
boolean flag3=find(state.block3, state.board[x-1][y]);
if(flag1==false&&flag2==false&&flag3==false){//属于1*1

        //更新数值
        next.board[x][y]=next.board[x-1][y];
        next.board[x-1][y]=0;

        next.x0[i]=x-1;
        next.y0[i]=y;


        return next;



}else if(flag1==true&&flag2==false&&flag3==false){//属于1*2


        int j=rangeofblock1(state, x-1, y);
        if(j==1){
            //下处需不需要判定y+1 是否超出边界
            if(state.board[x][y+1]==0){
                //更新数值
                next.board[x][y]=next.board[x-1][y];
                next.board[x][y+1]=next.board[x-1][y+1];
                next.board[x-1][y]=0;
                next.board[x-1][y+1]=0;

                next.x0[i]=x-1;
                next.y0[i]=y;
                next.x0[indexOfZero(state, x, y+1)]=x-1;
                next.y0[indexOfZero(state, x, y+1)]=y+1;



                return next;
            }
            else{
                return state;
            }
        }
        else{
            //下处需不需要判定y-1 是否超出边界
            if(state.board[x][y-1]==0){
                //更新数值
                next.board[x][y]=next.board[x-1][y];
                next.board[x][y-1]=next.board[x-1][y-1];
                next.board[x-1][y]=0;
                next.board[x-1][y-1]=0;
```

```
                    next.x0[i]=x-1;
                    next.y0[i]=y;
                    next.x0[indexOfZero(state, x, y-1)]=x-1;
                    next.y0[indexOfZero(state, x, y-1)]=y-1;


                    return next;
                }
                else{
                    return state;
                }
            }


        }
        else if(flag1==false&&flag2==true&&flag3==false){//属于2*1

            //更新数值
            next.board[x][y]=next.board[x-1][y];
            next.board[x-1][y]=next.board[x-2][y];
            next.board[x-2][y]=0;

            next.x0[i]=x-2;
            next.y0[i]=y;



            return next;

        }else{//属于2*2
            int j=rangeofblock3(state, x-1, y);
            if(j==1||j==2){
                return state;
            }
            else if(j==3){
                //下处需不需要判定y+1 是否超出边界
                if(state.board[x][y+1]==0){
                    //更新数值
                    next.board[x][y]=next.board[x-1][y];
                    next.board[x][y+1]=next.board[x-1][y+1];
                    //是否需要判断x-2 超出边界
                    next.board[x-1][y]=next.board[x-2][y];
                    next.board[x-1][y+1]=next.board[x-2][y+1];
                    next.board[x-2][y]=0;
                    next.board[x-2][y+1]=0;

                    next.x0[i]=x-2;
                    next.y0[i]=y;
                    next.x0[indexOfZero(state, x, y+1)]=x-2;
                    next.x0[indexOfZero(state, x, y+1)]=y;
```

```
                        return next;
                    }
                    else{
                        return state;
                    }
                }
                else{
                    //下处需不需要判定y-1 是否超出边界
                    if(state.board[x][y-1]==0){
                        //更新数值
                        next.board[x][y]=next.board[x-1][y];
                        next.board[x][y-1]=next.board[x-1][y-1];
                        //是否需要判断x-2 超出边界
                        next.board[x-1][y]=next.board[x-2][y];
                        next.board[x-1][y-1]=next.board[x-2][y-1];
                        next.board[x-2][y]=0;
                        next.board[x-2][y-1]=0;

                        next.x0[i]=x-2;
                        next.y0[i]=y;
                        next.x0[indexOfZero(state, x, y-1)]=x-2;
                        next.y0[indexOfZero(state, x, y-1)]=y;


                        return next;
                    }
                    else{
                        return state;
                    }

                }

            }

        }else{
            return state;
        }

    }

    //判断下面能不能换及结果
    public static Node down(Node state,int x,int y,int i){
        //state 表示当前状态，[x,y]表示当前选定0 的坐标,i 代表选定的0 在x0,y
0 中的index
        if(x+1<=state.n-1&&state.board[x+1][y]!=0){
            Node next = new Node(state);
```

```
//判断属于哪种类型的block
//需要判断的点的坐标在array 中的坐标是[x+1,y]
boolean flag1=find(state.block1, state.board[x+1][y]);
boolean flag2=find(state.block2, state.board[x+1][y]);
boolean flag3=find(state.block3, state.board[x+1][y]);

if(flag1==false&&flag2==false&&flag3==false){//属于1*1

        //更新数值
        next.board[x][y]=next.board[x+1][y];
        next.board[x+1][y]=0;

        next.x0[i]=x+1;
        next.y0[i]=y;


        return next;


}else if(flag1==true&&flag2==false&&flag3==false){//属于1*2



        int j=rangeofblock1(state, x+1, y);
        if(j==1){
            //下处需不需要判定y+1 是否超出边界
            if(state.board[x][y+1]==0){
                //更新数值
                next.board[x][y]=next.board[x+1][y];
                next.board[x][y+1]=next.board[x+1][y+1];
                next.board[x+1][y]=0;
                next.board[x+1][y+1]=0;

                next.x0[i]=x+1;
                next.y0[i]=y;
                next.x0[indexOfZero(state, x, y+1)]=x+1;
                next.y0[indexOfZero(state, x, y+1)]=y+1;



                return next;
            }
            else{
                return state;
            }
        }
        else{
            //下处需不需要判定y-1 是否超出边界
```

```
                if(state.board[x][y-1]==0){
                 //更新数值
                    next.board[x][y]=next.board[x+1][y];
                    next.board[x][y-1]=next.board[x+1][y-1];
                    next.board[x+1][y]=0;
                    next.board[x+1][y-1]=0;

                    next.x0[i]=x+1;
                    next.y0[i]=y;
                    next.x0[indexOfZero(state, x, y-1)]=x+1;
                    next.y0[indexOfZero(state, x, y-1)]=y-1;




                    return next;
                }
                else{
                    return state;
                }
            }


        }else if(flag1==false&&flag2==true&&flag3==false){//属于2*1



            //更新数值
            //是否需要判断x+2 超出边界
            next.board[x][y]=next.board[x+1][y];
            next.board[x+1][y]=next.board[x+2][y];
            next.board[x+2][y]=0;

            next.x0[i]=x+1;
            next.y0[i]=y;




            return next;


        }else{//属于2*2


            int j=rangeofblock3(state, x+1, y);
            if(j==3||j==4){
                return state;
            }else if(j==1){
```

```
            //下处需不需要判定y+1 是否超出边界
            if(state.board[x][y+1]==0){
            //更新数值
            next.board[x][y]=next.board[x+1][y];
            next.board[x][y+1]=next.board[x+1][y+1];
            //是否需要判断x+2 超出边界
            next.board[x+1][y]=next.board[x+2][y];
            next.board[x+1][y+1]=next.board[x+2][y+1];
            next.board[x+2][y]=0;
            next.board[x+2][y+1]=0;


            next.x0[i]=x+2;
            next.y0[i]=y;
            next.x0[indexOfZero(state, x, y+1)]=x+2;
            next.y0[indexOfZero(state, x, y+1)]=y+1;



            return next;
        }
            else{
            return state;
        }

        }else{
        //下处需不需要判定y-1 是否超出边界
        if(state.board[x][y-1]==0){
            //更新数值
            next.board[x][y]=next.board[x+1][y];
            next.board[x][y-1]=next.board[x+1][y-1];
            //是否需要判断x+2 超出边界
            next.board[x+1][y]=next.board[x+2][y];
            next.board[x+1][y-1]=next.board[x+2][y-1];
            next.board[x+2][y]=0;
            next.board[x+2][y-1]=0;

            next.x0[i]=x+2;
            next.y0[i]=y;
            next.x0[indexOfZero(state,x,y-1)]=x+2;
            next.y0[indexOfZero(state,x,y-1)]=y-1;


            return next;
        }
        else{
            return state;
        }
    }
```

```
        }

    }
    else{
        return state;
    }

}

//判断左面能不能换及结果
public static Node left(Node state,int x,int y,int i){
    //state 表示当前状态，[x,y]表示当前选定0 的坐标
    if(y-1>=0&&state.board[x][y-1]!=0){
        Node next = new Node(state);

        //判断属于哪种类型的block
        //需要判断的点的坐标在array 中的坐标是[x,y-1]
        boolean flag1=find(state.block1, state.board[x][y-1]);
        boolean flag2=find(state.block2, state.board[x][y-1]);
        boolean flag3=find(state.block3, state.board[x][y-1]);
        if(flag1==false&&flag2==false&&flag3==false){//属于1*1

                //更新数值
                next.board[x][y]=next.board[x][y-1];
                next.board[x][y-1]=0;

                next.x0[i]=x;
                next.y0[i]=y-1;


                return next;


        }else if(flag1==true&&flag2==false&&flag3==false){//属于1*2


                //更新数值
                //是否需要判断y-2 超出边界
                next.board[x][y]=next.board[x][y-1];
                next.board[x][y-1]=next.board[x][y-2];
                next.board[x][y-2]=0;

                next.x0[i]=x;
                next.y0[i]=y-2;


                return next;
```

```
}else if(flag1==false&&flag2==true&&flag3==false){//属于2*1


        int j=rangeofblock2(state, x, y-1);
        if(j==1){
            //下处需不需要判定x+1是否超出边界
            if(state.board[x+1][y]==0){
                //更新数值
                next.board[x][y]=next.board[x][y-1];
                next.board[x+1][y]=next.board[x+1][y-1];
                next.board[x][y-1]=0;
                next.board[x+1][y-1]=0;

                next.x0[i]=x;
                next.y0[i]=y-1;
                next.x0[indexOfZero(state, x+1, y)]=x+1;
                next.y0[indexOfZero(state, x+1, y)]=y-1;



                return next;
            }else{
                return state;
            }

        }else{
            //下处需不需要判定x-1是否超出边界
            if(state.board[x-1][y]==0){
                //更新数值
                next.board[x][y]=next.board[x][y-1];
                next.board[x-1][y]=next.board[x-1][y-1];
                next.board[x][y-1]=0;
                next.board[x-1][y-1]=0;

                next.x0[i]=x;
                next.y0[i]=y-1;
                next.x0[indexOfZero(state, x-1, y)]=x-1;
                next.y0[indexOfZero(state, x-1, y)]=y-1;



                return next;
            }
            else{
                return state;
            }
        }

}
```

```
else{//属于2*2

        int j=rangeofblock3(state, x, y-1);
        if(j==1||j==3){
            return state;
        }else if(j==2){
            //下处需不需要判定x+1 是否超出边界
            if(state.board[x+1][y]==0){
                //更新数值
                next.board[x][y]=next.board[x][y-1];
                next.board[x+1][y]=next.board[x+1][y-1];
                //是否需要判断y-2 超出边界
                next.board[x][y-1]=next.board[x][y-2];
                next.board[x+1][y-1]=next.board[x+1][y-2];
                next.board[x][y-2]=0;
                next.board[x+1][y-2]=0;

                next.x0[i]=x;
                next.y0[i]=y-2;
                next.x0[indexOfZero(state, x+1, y)]=x+1;
                next.y0[indexOfZero(state, x+1, y)]=y-2;



                return next;
            }
            else{
                return state;
            }
        }else{
            //下处需不需要判定x-1 是否超出边界
            if(state.board[x-1][y]==0){
                //更新数值
                next.board[x][y]=next.board[x][y-1];
                next.board[x-1][y]=next.board[x-1][y-1];
                //是否需要判断x-2 超出边界
                next.board[x][y-1]=next.board[x][y-2];
                next.board[x-1][y-1]=next.board[x-1][y-2];
                next.board[x-1][y-2]=0;
                next.board[x][y-2]=0;

                next.x0[i]=x;
                next.y0[i]=y-2;
                next.x0[indexOfZero(state, x-1, y)]=x-1;
                next.y0[indexOfZero(state, x-1, y)]=y-2;

                return next;
            }
            else{
```

```
                        return state;
                    }
                }



            }


        }else{
            return state;
        }
    }

    //判断右面能不能换及结果
    public static Node right(Node state,int x,int y,int i){
        //state 表示当前状态，[x,y]表示当前选定0 的坐标
        if(y+1<=state.m-1){
            Node next = new Node(state);

        //判断属于哪种类型的block
        //需要判断的点的坐标在array 中的坐标是[x,y+1]
        boolean flag1=find(state.block1, state.board[x][y+1]);
        boolean flag2=find(state.block2, state.board[x][y+1]);
        boolean flag3=find(state.block3, state.board[x][y+1]);

        if(flag1==false&&flag2==false&&flag3==false){//属于1*1

                //更新数值
             next.board[x][y]=next.board[x][y+1];
             next.board[x][y+1]=0;

             next.x0[i]=x;
             next.y0[i]=y+1;


             return next;


        }else if(flag1==true&&flag2==false&&flag3==false){//属于1*2

                //更新数值
                //是否需要判断y+2 超出边界
                next.board[x][y]=next.board[x][y+1];
                next.board[x][y+1]=next.board[x][y+2];
                next.board[x][y+2]=0;

                next.x0[i]=x;
                next.y0[i]=y+2;
```

```
            return next;


        }else if(flag1==false&&flag2==true&&flag3==false){//属于2*1

            int j=rangeofblock2(state, x, y+1);
            if(j==1){
                //下处需不需要判定x+1是否超出边界
                if(state.board[x+1][y]==0){
                    //更新数值
                    next.board[x][y]=next.board[x][y+1];
                    next.board[x+1][y]=next.board[x+1][y+1];
                    next.board[x][y+1]=0;
                    next.board[x+1][y+1]=0;

                    next.x0[i]=x;
                    next.y0[i]=y+1;
                    next.x0[indexOfZero(state, x+1, y)]=x+1;
                    next.y0[indexOfZero(state, x+1, y)]=y+1;



                    return next;
                }else{
                    return state;
                }

            }else{
                //下处需不需要判定x-1是否超出边界
                if(state.board[x-1][y]==0){
                    //更新数值
                    next.board[x][y]=next.board[x][y+1];
                    next.board[x-1][y]=next.board[x-1][y+1];
                    next.board[x][y+1]=0;
                    next.board[x-1][y+1]=0;

                    next.x0[i]=x;
                    next.y0[i]=y+1;
                    next.x0[indexOfZero(state, x-1, y)]=x-1;
                    next.y0[indexOfZero(state, x-1, y)]=y+1;



                    return next;
                }
                else{
                    return state;
```

```
                }


        }
    }else{//属于2*2

            int j=rangeofblock3(state, x, y+1);
            if(j==2||j==4){
                return state;
            }else if(j==1){
                //下处需不需要判定x+1是否超出边界
                if(state.board[x+1][y]==0){
                    //更新数值
                    next.board[x][y]=next.board[x][y+1];
                    next.board[x+1][y]=next.board[x+1][y+1];
                    //是否需要判断y-2超出边界
                    next.board[x][y+1]=next.board[x][y+2];
                    next.board[x+1][y+1]=next.board[x+1][y+2];
                    next.board[x][y+2]=0;
                    next.board[x+1][y+2]=0;

                    next.x0[i]=x;
                    next.y0[i]=y+2;
                    next.x0[indexOfZero(state, x+1, y)]=x+1;
                    next.y0[indexOfZero(state, x+1, y)]=y+2;



                    return next;
                }
                else{
                    return state;
                }

            }else{
                //下处需不需要判定x-1是否超出边界
                if(state.board[x-1][y]==0){
                    //更新数值
                    next.board[x][y]=next.board[x][y+1];
                    next.board[x-1][y]=next.board[x-1][y+1];
                    //是否需要判断y+2超出边界
                    next.board[x][y+1]=next.board[x][y+2];
                    next.board[x-1][y+1]=next.board[x-1][y+2];
                    next.board[x][y+2]=0;
                    next.board[x-1][y+2]=0;

                    next.x0[i]=x;
                    next.y0[i]=y+2;
```

```
                    next.x0[indexOfZero(state, x-1, y)]=x-1;
                    next.y0[indexOfZero(state, x-1, y)]=y+2;



                    return next;
                }
                else{
                    return state;
                }


            }

        }

        }
        else{
            return state;
        }

    }

    //判断一个数是否属于二维数组中
    public static boolean find(int[][]array,int target){
        boolean flag = false;
        if(array!=null&&array.length!=0&&array[0].length!=0){
            for (int i = 0; i < array.length; i++) {
                for (int j = 0; j <array[i].length ; j++) {
                    if (target == array[i][j]) { //判断是否存在该整数
                        flag = true;  //存在
                        return flag;
                    }
                }
            }

        }
        return flag;
    }

    //判断属于1*2block 中的第一个元素还是第二个元素
    //如果是第一个，就返回1，如果是第二个，就返回2
    public static int rangeofblock1(Node state,int x,int y){
        //state 为目前需要判断的状态，[x,y]为需要判断的点在state.array 中的
坐标
        int column=0;
        for(int i=0;i<state.block1.length;i++){
            for(int j=0;j<state.block1[0].length;j++){
                if(state.block1[i][j]==state.board[x][y]){
                    column=j;
```

```
                }
            }
        }
        if(column==0){
            return 1;
        }else{
            return 2;
        }
    }

    //判断属于2*1 block 中的第一个元素还是第二个元素
    //如果是第一个，就返回1，如果是第二个，就返回2
    public static int rangeofblock2(Node state,int x,int y){
        //state 为目前需要判断的状态，[x,y]为需要判断的点在state.array 中的
坐标
        int column=0;
        for(int i=0;i<state.block2.length;i++){
            for(int j=0;j<state.block2[0].length;j++){
                if(state.block2[i][j]==state.board[x][y]){
                    column=j;
                }
            }
        }
        if(column==0){
            return 1;
        }else{
            return 2;
        }
    }

    //判断属于2*2 block 中的第几个元素
    //如果是第一个，就返回1，如果是第二个，就返回2，以此类推
    public static int rangeofblock3(Node state,int x,int y){
        int column=0;
        for(int i=0;i<state.block3.length;i++){
            for(int j=0;j<state.block3[0].length;j++){
                if(state.block3[i][j]==state.board[x][y]){
                    column=j;
                }
            }
        }
        if(column==0){
            return 1;
        }else if(column==1){
            return 2;
        }else if(column==2){
            return 3;
        }else{
            return 4;
```

```
        }
    }

    //根据选定的0 的坐标判断这个0 属于neoCount 中第几个0
    public static int indexOfZero(Node state,int x,int y){
        //[x,y]为该0 在state.board 中的坐标
        int index=0;

        for(int i=0;i<state.x0.length;i++){
            if(state.x0[i]==x&&state.y0[i]==y){
                index=i;
            }
        }

        return index;

    }


}

package Breadth_First;

import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.PriorityQueue;
import java.util.Scanner;
import java.util.Stack;

import edu.princeton.cs.algs4.Queue;

import static Breadth_First.Node.arrayToString;
import static Breadth_First.Node.printState;
import static Breadth_First.Node.transform;



public class Run {
    public static void main(String[] args) throws FileNotFoundException
 {
        long start = System.currentTimeMillis();
        //standard input
        try (Scanner input = new Scanner(System.in)) {
            int N = input.nextInt();
            int M = input.nextInt();
            int[][] initArray = new int[N][M];
            for(int i=0;i<N;i++){
                for(int j=0;j<M;j++){
                    initArray[i][j]=input.nextInt();
                }
            }
```

```
            int numberOfblock = input.nextInt();
            int[][] allOfBlock=new int[numberOfblock][2];
            if(numberOfblock!=0){
                for(int i=0;i<numberOfblock;i++){
                    allOfBlock[i][0]=input.nextInt();
                    //因为表示block种类的输入是字符类型，将各种种类转换为
整数来分类，
                    allOfBlock[i][1]=transform(input.next());
                }
            }else{
                allOfBlock=null;
            }

            ArrayList<String> close = new ArrayList<String>();//判断是否
在close中，表示已经搜索过的状态
            Queue<Node> queue = new Queue<Node>();//要搜索的状态



            //初始状态
            Node initState = new Node(initArray, null,numberOfblock,all
OfBlock);
            //将初始状态放入
            queue.enqueue(initState);

            boolean flag = false;
            Node ans = null;

            while (!queue.isEmpty()){
                Node currentState = queue.dequeue();
                if(currentState.isComplete()){
                    flag = true;
                    ans = currentState;
                    break;
                }
                currentState.findNext(queue,close);
            }

            Stack<String> move = new Stack<String>();
            Stack<String> boards = new Stack<String>();
            int moveCount = 0;

            if(flag){
                Node t = ans;



                boards.push(printState(t));
                move.push(t.movement);
```

```
                t = t.father;

                while(t != null){
                    boards.push(printState(t));
                    if(t.movement != null){
                        move.push(t.movement);
                    }
                    moveCount++;
                    t = t.father;
                }

                System.out.println("Yes");
                System.out.println(moveCount);

                while (!move.isEmpty()){
                    System.out.println(move.pop());

                }
            }
            else{
                System.out.print("No\n");
            }
            System.out.print((System.currentTimeMillis() - start) + " m
s");
        }


    }
}
```