A*算法求解八数码问题

学号：161220217 班级：1612001 姓名：骆克云

一、 八数码问题概述

1. 问题概述

八数码问题是在 3*3 九宫格上,放八个数码,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。问题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。状态转换的规则：空格周围的数移向空格,我们可以看作是空格移动,它最多可以有 4 个方向的移动,即上、下、左、右。九宫重排问题的求解方法,就是从给定的初始状态出发,不断地空格上下左右的数码移至空格,将一个状态转化成其它状态,直到产生目标状态。

图 初始状态和目标状态

2	8	3
1		4
7	6	5

1	2	3
8		4
7	6	5

2. 相关解法

对于八数码问题，目前有多种解法。给定初始状态,9 个数在 3*3 中的放法共有 9!＝362880 种,其状态空间是非常大。因此， 有必要考虑与问题相关的启发性信息来指导搜索,以提高搜索的效率。当然,还有个很重要的问题：每个初始状态都存在解路径吗？经过学者研究，给出了九宫重排问题是否有解的判别方法：九宫重排问题存在无解的情况,当遍历完所有可扩展的状态也没有搜索到目标状态就判断为无解。可以根据状态的逆序数来先验的判断是否有解,当初始状态的逆序数和目标状态的逆序数的奇偶性相同时,问题有解;否则问题无解。状态的逆序数是定义把三行数展开排成一行,并且丢弃数字 0 不计入其中,ηi 是第 i 个数之前比该数小的数字的个数,则 η=Σ ηi 是该状态的逆序数。

3. 问题的搜索形式描述

状态：状态描述了 8 个棋子和空位在棋盘的 9 个方格上的分布。

初始状态：任何状态都可以被指定为初始状态。

操作符：用来产生 4 个行动（上下左右移动）。

目标测试：用来检测状态是否能匹配上图的目标布局。

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。

目标：现在任意给定一个初始状态，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。

4. 程序思路

本程序采用 Java 语言编写，使用面向对象的设计模式，将 open,close,node,find,main 单独封装成类。首先根据逆序数判断手否可解，给出三个算法选项：广度优先(h(n)=0)，目标差异计数法， 曼哈顿距离，比较不同算法下的执行效率。

二、 A*算法框架

A*算法，作为启发式算法中很重要的一种，被广泛应用在最优路径求解和一些策

略设计的问题中。而 A*算法最为核心的部分，就在于它的一个估值函数的设计上：

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

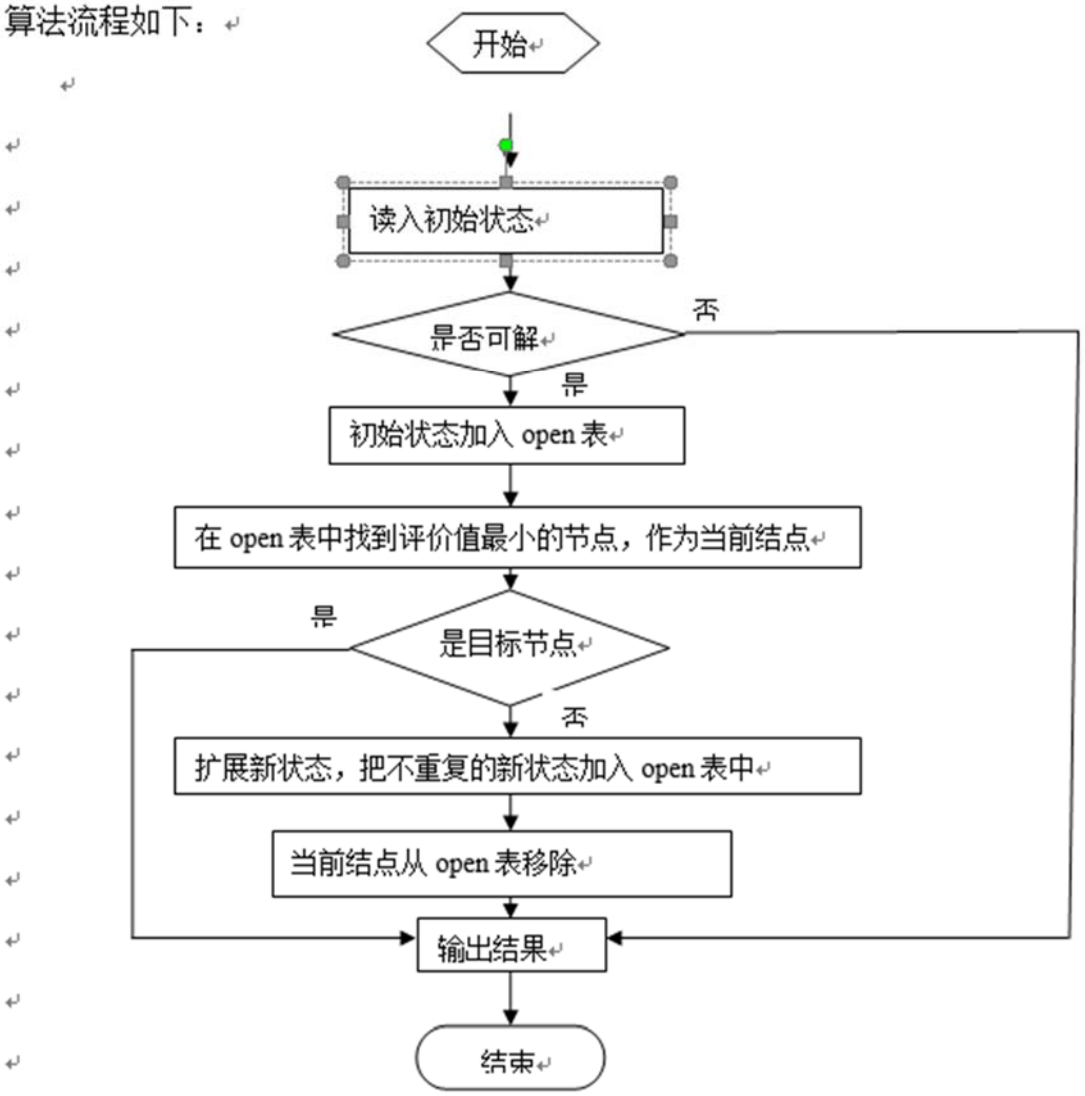
其中 f(n)是每个可能试探点的估值，它有两部分组成：一部分为 g(n)，它表示从起始搜索点到当前点的代价（通常用某结点在搜索树中的深度来表示）。另一部分，即 h(n)，它表示启发式搜索中最为重要的一部分，即当前结点到目标结点的估值，h(n)设计的好坏，直接影响着具有此种启发式函数的启发式算法的是否能称为 A*算法。

一种具有 f(n)=g(n)+h(n)策略的启发式算法能成为 A*算法的充分条件是：

1) 搜索树上存在着从起始点到终了点的最优路径。
2) 问题域是有限的。
3) 所有结点的子结点的搜索代价值>0。
4) h(n)=<h*(n) (h*(n)为实际问题的代价值)。

当此四个条件都满足时，一个具有 f(n)=g(n)+h(n)策略的启发式算法能成为 A*算法，并一定能找到最优解。对于一个搜索问题，显然，条件 1,2,3 都是很容易满足的，而 条件 4)： h(n)<=h*(n)是需要精心设计的，由于 h*(n)显然是无法知道的。所以，一个满足条件 4）的启发策略 h(n)就来的难能可贵了。不过 h(n)距离 h*(n)的程度不能过大，否则 h(n)就没有过强的区分能力，算法效率并不会很高。对一个好的 h(n)的评价是：h(n)在 h*(n)的下界之下，并且尽量接近 h*(n)。

算法流程如下：



三、数据结构

1. Node 类：

说明：九宫格：int[][] puzzle = new int[3][3];

F,g,h 值： int gvalue;int hvalue;int fvalue;

next 链表： Node pnode;

向左移动：getLeft

向右移动：getRight

向上移动：getUp

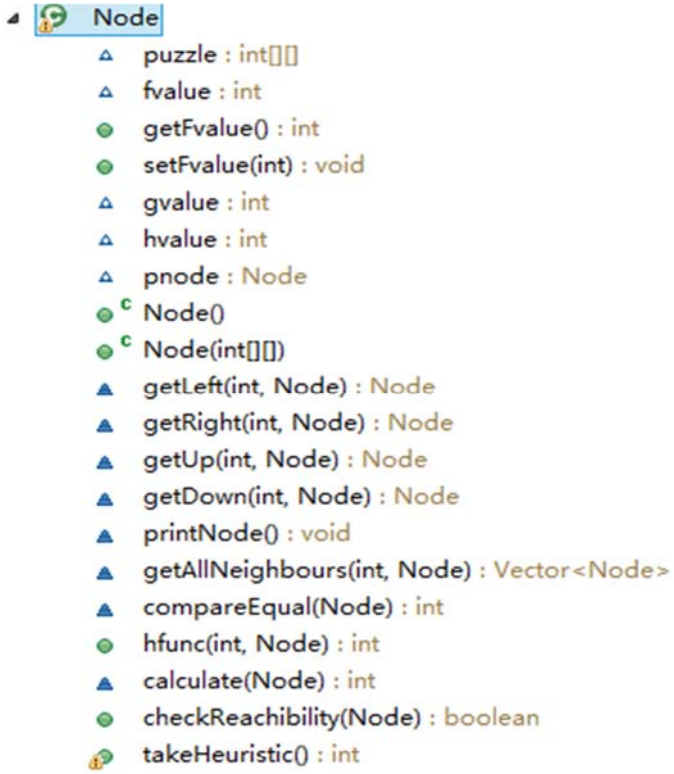
向下移动：getDown

输出结点：printNode

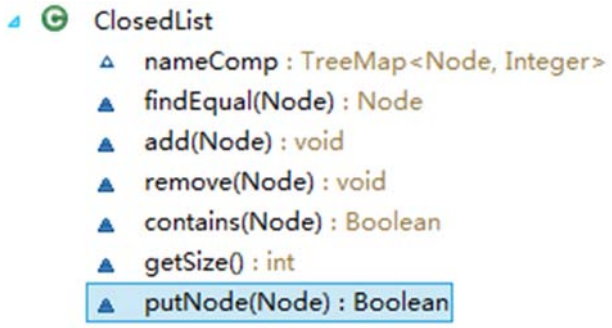
计算逆序数：calculate

计算是否有解:checkReachibility

选取启发式算法：takeHeuristic

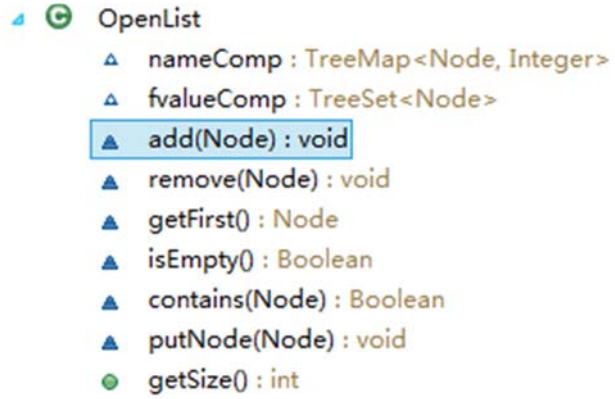


2. Close 表类：



说明：nameComp:结点与数字之间的映射
FindEqual:比较两个结点的值是否相等
Add:添加结点
Remove：移除结点
Contains:是否包括某个结点
PutNode:判断 g 值，若小于原来的值，则返回不能搜索的标志：false，否则返回 true，用于生成结点。

3. Open 表类：



说明：putNode：比较 f 值确定是否设置新的 fvalue 值，对 f 进行扩展。

四、 启发式函数设计

1. 广度优先搜索:在九宫格中只要有一个位置的值不同，就返回 count=1，该启发函数相当于为 1，所需要的代价也最大。

```
if(heuristic == 1){
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                if(puzzle[i][j]!=goal.puzzle[i][j])
                    return 1;
            }
        }
 }
```

2. 目标差异计数法搜索:根据在九宫格中与目标结点的各个位置中不同的数目来启发搜索，朝着值越来越小的方向。

```
else if(heuristic == 2){
        for(int i=0;i<3;i++){
```

```
                for(int j=0;j<3;j++){
                    if(puzzle[i][j]==0)
                        continue;
                    if(puzzle[i][j]!=goal.puzzle[i][j] )
                        count++;
                }
            }
        }
```

3. 曼哈顿距离搜索：每一个数字位与目标中该数字位的距离，满足单调限制，朝着距离越来越小的方向搜索。A*算法是启发式搜索算法，搜索时充分利用当前状态距目标距离远近的启发信息，选取当前未扩展结点中估价函数最小的进行扩展，生成结点数少，搜索空间较小。

```
    else if(heuristic ==3){
            int[] x = new int[8];
            int[] y = new int[8];
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                    if(puzzle[i][j]==0)
                        continue;
                    x[puzzle[i][j]-1] = i;
                    y[puzzle[i][j]-1] = j;
                }
            }
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                    if(goal.puzzle[i][j]==0)
                        continue;
                    int row = x[goal.puzzle[i][j]-1];
                    int column = y[goal.puzzle[i][j]-1];
                    count = count + Math.abs(i-row)+Math.abs(j-column);
                }
            }
    }
```

五、 运行结果与分析

1. 测试数据：
   a. 无解的情况

      初始：src_matrix = {{1,3,5},{0,7,4},{6,8,2}};

      目标：goal_matrix = {{1,2,3},{4,5,6},{7,8,0}}

```
选择算法
1 广度优先          (h=1)
2 目标棋异计数法    (h=Σ(与目标结点不同的位置数))
3 曼哈顿距离        (h=Σ(|src.x-goal.x|+|src.y-goal.y|))

1
源结点逆序数：9
目标结点逆序数：0
无解！
```

截图：

b. 有解的情况

初始：src_matrix = {{1,3,5},{0,7,4},{6,2,8}};

目标：goal_matrix = {{1,2,3},{4,5,6},{7,8,0}}

2. 广度优先搜索(结果过长，部分展示)

运行结果：

```
选择算法
1 广度优先          (h=1)
2 目标棋异计数法    (h=Σ(与目标结点不同的位置数))
3 曼哈顿距离        (h=Σ(|src.x-goal.x|+|src.y-goal.y|))

1
源结点逆序数：8
目标结点逆序数：0
找到最优路径，该路径为：

第1步：
打印结点：
1 3 5
7 0 4
6 2 8
f(n)=2
g(n)= 1
h(n)=1

第2步：
打印结点：
1 3 5
7 4 0
6 2 8
f(n)=3
g(n)= 2
h(n)=1

第3步：
打印结点：
1 3 5
7 4 8
6 2 0
f(n)=4
g(n)= 3
```

……

```
f(n)=15
g(n)= 14
h(n)=1

第15步：
打印结点：
1 2 3
4 0 5
7 8 6
f(n)=16
g(n)= 15
h(n)=1

第16步：
打印结点：
1 2 3
4 5 0
7 8 6
f(n)=17
g(n)= 16
h(n)=1

第17步：
打印结点：
1 2 3
4 5 6
7 8 0
f(n)=17
g(n)= 17
h(n)=0

路径长度：17
Open 表大小：4393
Closed 表大小：8689
```

3. 目标差异计数法搜索(部分)

```
选择算法
1 广度优先          (h=1)
2 目标错异计数法    (h=Σ(与目标结点不同的位置数))
3 曼哈顿距离        (h=Σ(|src.x-goal.x|+|src.y-goal.y|))

2
初始结点逆序数：8
目标结点逆序数：0
找到最优路径，该路径为：

第1步：
打印结点：
1 3 5
7 0 4
6 2 8
f(n)=8
g(n)= 1
h(n)=7

第2步：
打印结点：
1 3 5
7 4 0
6 2 8
f(n)=9
g(n)= 2
h(n)=7

第3步：
打印结点：
1 3 5
7 4 8
6 2 0
f(n)=10
g(n)= 3
```

...

```
f(n)=17
g(n)= 14
h(n)=3

第15步：
打印结点：
1 2 3
4 0 5
7 8 6
f(n)=17
g(n)= 15
h(n)=2

第16步：
打印结点：
1 2 3
4 5 0
7 8 6
f(n)=17
g(n)= 16
h(n)=1

第17步：
打印结点：
1 2 3
4 5 6
7 8 0
f(n)=17
g(n)= 17
h(n)=0

路径长度：17
Open 表大小：425
Closed 表大小：666
```

4. 曼哈顿距离搜索(完整)

选择算法

1 广度优先 (h=1)

2 目标差异计数法 (h=Σ(与目标结点不同的位置数))

3 曼哈顿距离 (h=Σ(|src.x-goal.x|+|src.y-goal.y|))

3

源结点逆序数： 8

目标结点逆序数： 0

找到最优路径，该路径为：

第 1 步：

打印结点：

1 3 5

7 0 4

6 2 8

f(n)=13

g(n)= 1

h(n)=12

第 2 步：
打印结点：
1 3 5
7 4 0
6 2 8
f(n)=13
g(n)= 2
h(n)=11

第 3 步：
打印结点：
1 3 5
7 4 8
6 2 0
f(n)=15
g(n)= 3
h(n)=12

第 4 步：
打印结点：
1 3 5
7 4 8
6 0 2
f(n)=17
g(n)= 4
h(n)=13

第 5 步：
打印结点：
1 3 5
7 4 8
0 6 2
f(n)=17
g(n)= 5
h(n)=12

第 6 步：
打印结点：
1 3 5
0 4 8
7 6 2
f(n)=17
g(n)= 6
h(n)=11

第 7 步：
打印结点：
1 3 5
4 0 8
7 6 2
f(n)=17
g(n)= 7
h(n)=10

第 8 步：
打印结点：
1 3 5
4 8 0
7 6 2
f(n)=17
g(n)= 8
h(n)=9

第 9 步：
打印结点：
1 3 5
4 8 2
7 6 0
f(n)=17
g(n)= 9
h(n)=8

第 10 步：
打印结点：
1 3 5
4 8 2
7 0 6
f(n)=17
g(n)= 10
h(n)=7

第 11 步：
打印结点：
1 3 5
4 0 2
7 8 6
f(n)=17
g(n)= 11

h(n)=6

第 12 步：
打印结点：
1 3 5
4 2 0
7 8 6
f(n)=17
g(n)= 12
h(n)=5

第 13 步：
打印结点：
1 3 0
4 2 5
7 8 6
f(n)=17
g(n)= 13
h(n)=4

第 14 步：
打印结点：
1 0 3
4 2 5
7 8 6
f(n)=17
g(n)= 14
h(n)=3

第 15 步：
打印结点：
1 2 3
4 0 5
7 8 6
f(n)=17
g(n)= 15
h(n)=2

第 16 步：
打印结点：
1 2 3
4 5 0
7 8 6
f(n)=17

g(n)= 16
h(n)=1

第 17 步：
打印结点：
1 2 3
4 5 6
7 8 0
f(n)=17
g(n)= 17
h(n)=0

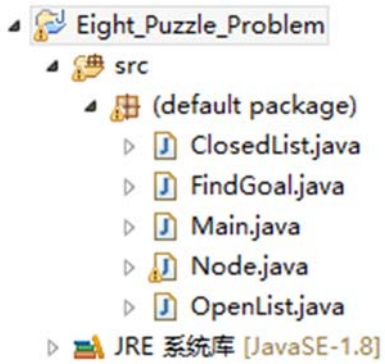
路径长度： 17
Open 表大小： 62
Closed 表大小： 96

5. 结果分析

对比三个不同的启发式算法，可以发现单纯的宽度优先搜索耗费的代价最大，在搜索路径长度为 17 的情况下，Open 表达到了 4393 个，Close 表大小达到了 8689，即生成了 8689 个结点，拓展了 4393 个结点。而目标差异计数法则生成了 666 个结点，拓展了 425 个结点。最快的是曼哈顿距离启发式算法，只扩展了 62 个结点，生成了 96 个结点。因此，曼哈顿距离启发式算法搜索效果比较好。

六、 程序代码

1. 程序架构



2. CloseList.java

```
import java.util.Map;
import java.util.TreeMap;
public class ClosedList {

    TreeMap<Node,Integer> nameComp = new TreeMap<Node,Integer>(new
MyNameComp());

    Node findEqual(Node node){
        for(Map.Entry<Node,Integer> entry : nameComp.entrySet()) {
            Node key = entry.getKey();
            if(key.compareEqual(node)==0){
```

```
                    return key;
                }

            }
        return null;
    }
    void add(Node node){
        nameComp.put(node,node.gvalue);
    }

    void remove(Node node){
        nameComp.remove(node);
    }

    Boolean contains(Node node){
        return nameComp.containsKey(node);
    }

    int getSize(){
        return nameComp.size();
    }


    Boolean putNode(Node node){
        if(this.contains(node)){
            if(nameComp.get(node) > node.gvalue){
                nameComp.remove(node);
                return false;
            }
            return true;
        }
        return false;
    }
}
```

3. OpenList.java

```
import java.util.TreeMap;
import java.util.TreeSet;

public class OpenList {
 TreeMap<Node,Integer> nameComp = new TreeMap<Node,Integer>(new MyNameComp());
 TreeSet<Node> fvalueComp = new TreeSet<Node>(new MyFvalueComp());

 void add(Node node){
```

```
        nameComp.put(node,node.fvalue);
        fvalueComp.add(node);
}

void remove(Node node){
        nameComp.remove(node);
        fvalueComp.remove(node);
}

Node getFirst(){
        Node node = fvalueComp.first();
        return node;
}

Boolean isEmpty(){
        return fvalueComp.isEmpty();
}

Boolean contains(Node node){
        return nameComp.containsKey(node);
}

void putNode(Node node){
        if(this.contains(node)){
                Integer oldVal = nameComp.get(node);
                if(oldVal > node.getFvalue()){
                        int newVal = node.getFvalue();
                        node.setFvalue(oldVal);
                        fvalueComp.remove(node);
                        nameComp.put(node,node.fvalue);
                        node.setFvalue(newVal);
                        fvalueComp.add(node);
                }

        }
        else{
                this.add(node);
        }

}

public int getSize() {
        return fvalueComp.size();
}
```

```
}
```

4. Node.java

```
import java.util.Comparator;
import java.util.Scanner;
import java.util.Vector;
import java.lang.Math;

class MyNameComp implements Comparator<Node>{
	@Override
	public int compare(Node n1, Node n2) {
	 int val = n1.compareEqual(n2);
		return val;
	}
}

class MyFvalueComp implements Comparator<Node>{
	@Override
	public int compare(Node n1, Node n2) {
		if(n1.getFvalue() > n2.getFvalue()){
			return 1;
		}
		else if(n1.getFvalue() < n2.getFvalue()){
			return -1;
		}
		return n1.compareEqual(n2);
	}
}


public class Node {
 int[][] puzzle = new int[3][3];
 int fvalue;
 public int getFvalue() {
	 return fvalue;
 }

 public void setFvalue(int fvalue) {
	 this.fvalue = fvalue;
 }

 int gvalue;
 int hvalue;
 Node pnode;
```

```
//默认构造函数
public Node(){
	for(int i=0;i<3;i++){
		for(int j=0;j<3;j++){
			puzzle[i][j]=-1;
		}
	}
	fvalue = 0;
	gvalue = 0;
	hvalue=0;
	pnode = null;
}

//结点带参数的构造函数
public Node(int[][] check){
	for(int i=0;i<3;i++){
		for(int j=0;j<3;j++){
			puzzle[i][j]=check[i][j];
		}
		fvalue = 0;
		gvalue = 0;
		hvalue=0;
		pnode = null;
	}
}

//获取移动到左面的结点
Node getLeft(int heuristic, Node goal){
	Node temp = new Node();
	for(int i=0;i<3;i++){
		for(int j=0;j<3;j++){
			if(puzzle[i][j]==0){
				if(j==0) return null;
				temp.puzzle[i][j-1] = puzzle[i][j];
				temp.puzzle[i][j] = puzzle[i][j-1];
			}
			else{
				temp.puzzle[i][j]=puzzle[i][j];
			}
		}
	}
	temp.gvalue = gvalue + 1;
	temp.hvalue=temp.hfunc(heuristic,goal);
	temp.fvalue = temp.gvalue+temp.hvalue;
```

```
        temp.pnode = this;
        return temp;
}

//获取移动到右面的结点
Node getRight(int heuristic, Node goal){
        Node temp = new Node();
        for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                        if(puzzle[i][j]==0){
                                if(j==2) return null;
                                temp.puzzle[i][j+1] = puzzle[i][j];
                                temp.puzzle[i][j] = puzzle[i][j+1];
                                j++;
                        }
                        else{
                                temp.puzzle[i][j]=puzzle[i][j];
                        }
                }
        }
        temp.gvalue = gvalue + 1;
        temp.hvalue=temp.hfunc(heuristic,goal);
        temp.fvalue = temp.gvalue+temp.hvalue;
        temp.pnode = this;
        return temp;
}

//获取移动到上面的结点
Node getUp(int heuristic, Node goal){
        Node temp = new Node();
        for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                        if(puzzle[i][j]==0){
                                if(i==0) return null;
                                temp.puzzle[i-1][j] = puzzle[i][j];
                                temp.puzzle[i][j] = puzzle[i-1][j];
                        }
                        else{
                                temp.puzzle[i][j]=puzzle[i][j];
                        }
                }
        }
        temp.gvalue = gvalue + 1;
        temp.hvalue=temp.hfunc(heuristic,goal);
```

```
        temp.fvalue = temp.gvalue+temp.hvalue;
        temp.pnode = this;
        return temp;
}

//获取移动到下面的结点
Node getDown(int heuristic, Node goal){
        Node temp = new Node();
        for(int j=0;j<3;j++){
                for(int i=0;i<3;i++){
                        if(puzzle[i][j]==0){
                                if(i==2) return null;
                                temp.puzzle[i+1][j] = puzzle[i][j];
                                temp.puzzle[i][j] = puzzle[i+1][j];
                                i++;
                        }
                        else{
                                temp.puzzle[i][j]=puzzle[i][j];
                        }
                }
        }
        temp.gvalue = gvalue + 1;
        temp.hvalue=temp.hfunc(heuristic,goal);
        temp.fvalue = temp.gvalue+temp.hvalue;
        temp.pnode = this;
        return temp;
}

//输出八数码结点
void printNode(){
        System.out.println("\n 打印结点： ");
        System.out.println(puzzle[0][0]+" "+puzzle[0][1]+" "+puzzle[0][2]);
        System.out.println(puzzle[1][0]+" "+puzzle[1][1]+" "+puzzle[1][2]);
        System.out.println(puzzle[2][0]+" "+puzzle[2][1]+" "+puzzle[2][2]);
        System.out.println("f(n)="+fvalue);
        System.out.println("g(n)= "+gvalue);
        System.out.println("h(n)="+hvalue);
}

//获取邻居结点
Vector<Node> getAllNeighbours(int heuristic,Node goal){
        Vector<Node> allNeighbours = new Vector<Node>();

        Node leftNode = this.getLeft(heuristic,goal);
```

```
        if(leftNode!=null)
            allNeighbours.add(leftNode);
        Node rightNode = this.getRight(heuristic,goal);
        if(rightNode!=null)
            allNeighbours.add(rightNode);
        Node upNode = this.getUp(heuristic,goal);
        if(upNode!=null)
            allNeighbours.add(upNode);
        Node downNode = this.getDown(heuristic,goal);
        if(downNode!=null)
            allNeighbours.add(downNode);
        return allNeighbours;
    }

    //结点比较
    int compareEqual(Node node){
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                if(puzzle[i][j]<node.puzzle[i][j]){
                    return -1;
                }
                if(puzzle[i][j]>node.puzzle[i][j]){
                    return 1;
                }
            }
        }
        return 0;
    }

    //启发函数设计
    public int hfunc(int heuristic, Node goal) {
        int count=0;
        //与目标结点的一个位置不同就返回 1
        if(heuristic == 1){
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                    if(puzzle[i][j]!=goal.puzzle[i][j])
                        return 1;
                }
            }
        }

        //返回与目标结点的位置不同数目
        else if(heuristic == 2){
```

```
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                    if(puzzle[i][j]==0)
                        continue;
                    if(puzzle[i][j]!=goal.puzzle[i][j] )
                        count++;
                }
            }
        }

        //曼哈顿距离
        else if(heuristic ==3){
            int[] x = new int[8];
            int[] y = new int[8];
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                    if(puzzle[i][j]==0)
                        continue;
                    x[puzzle[i][j]-1] = i;
                    y[puzzle[i][j]-1] = j;
                }
            }
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                    if(goal.puzzle[i][j]==0)
                        continue;
                    int row = x[goal.puzzle[i][j]-1];
                    int column = y[goal.puzzle[i][j]-1];
                    count = count + Math.abs(i-row)+Math.abs(j-column);
                }
            }
        }

        return count;
    }

    //计算逆序数
    int calculate(Node node){
        int[] sample=new int[8];
        int num=0;
        int index=0;
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                if(node.puzzle[i][j]==0){
```

```
                    index=1;
                    continue;
                }
                sample[3*i+j-index]=node.puzzle[i][j];
            }
        }

        for(int i=0;i<8;i++){
            for(int j=i+1;j<8;j++){
                if(sample[j]<sample[i])
                    num++;
            }
        }
        return num;
    }

    //判断可解性
    public boolean checkReachibility(Node goal) {
        int srcnum = calculate(this);
        int goalnum = calculate(goal);
        System.out.println("源结点逆序数：  "+srcnum);
        System.out.println("目标结点逆序数：  "+goalnum);
        if((srcnum-goalnum)%2==0)
            return true;
        return false;
    }

    //扫描启发式算法选择
    public int takeHeuristic() {
        System.out.println("  选  择  算  法  \n1  广  度  优  先
(h=1)\n"+
                           "2 目标差异计数法          (h= Σ(与目标结点不
同的位置数))\n"+
                           "3 曼哈顿距离                    (h= Σ(|src.x-
goal.x|+|src.y-goal.y|))\n");
        @SuppressWarnings("resource")
        int h = new Scanner(System.in).nextInt();
        return h;
    }

}
```

5. FindGoal.java

```
    import java.util.ArrayList;
    import java.util.Enumeration;
```

```
import java.util.List;
import java.util.Vector;

public class FindGoal {

 void findGoal(Node src, Node goal,int heuristic){
        OpenList ol = new OpenList();
        ClosedList cl = new ClosedList();
        ol.add(src);
        Boolean reachability = false;
        while(!ol.isEmpty()){
            Node node = ol.getFirst();
            ol.remove(node);
            cl.add(node);
            if(node.compareEqual(goal)==0){
                System.out.println("找到最优路径，该路径为：");
                reachability = true;
                int pathLength = node.gvalue;

                List<Node> tmpNodes=new ArrayList<Node>();
                while(node.pnode!=null){
                    tmpNodes.add(0, node);
                    node = node.pnode;
                }

                for(int i=0;i<tmpNodes.size();i++){
                    System.out.printf("\n 第%d 步： ", i+1);
                    tmpNodes.get(i).printNode();
                }

                System.out.println("\n 路径长度：  "+pathLength);
                break;
            }
            else{
                Vector<Node> neighbours = node.getAllNeighbours(heuristic,goal);
                Enumeration<Node> vEnum = neighbours.elements();
                while(vEnum.hasMoreElements()){
                    Node element = vEnum.nextElement();
                    if(cl.putNode(element))
                        continue;
                    ol.putNode(element);
                }
            }
        }
```

```
            if(!reachability)
                System.out.println("解路径不存在！");
            System.out.println("Open 表大小： "+ol.getSize());
            System.out.println("Closed 表大小： "+cl.getSize());

            return;
        }

       }
```

6. Main.java

```
    public class Main {
     public static void main(String[] args){
            int[][] src_matrix = {{1,3,5},{0,7,4},{6,2,8}};
            int[][] goal_matrix = {{1,2,3},{4,5,6},{7,8,0}};
            Node src = new Node(src_matrix);
            Node goal = new Node(goal_matrix );
            FindGoal findGoal=new FindGoal();
            int heuristic = src.takeHeuristic();
            if(src.checkReachibility(goal))
                findGoal.findGoal(src,goal,heuristic);
            else
                System.out.println("无解！");
        }
    }
```