曲阜師範大學

学生实验报告

学院 计算机学院

课程 人工智能实训

姓名 彭清元

学号 2020416240

年级 2020 级

专业 计算机科学与技术

曲阜师范大学实验报告

学院 计算机学院 年级、专业 20 级计算机科学与技术二班 学号 2020416240

姓名 彭清元 课程名称 人工智能 上课时间 第 1 节 — 第 11 节

实验日期 2023 年 5 月 14 日 星期 日 教师签名 雷玉霞 成绩

实验一 九宫重排

1. 【实训目的】

（1）学习使用盲目搜索、宽度搜索、启发式搜索解决九宫重排问题
（2）代码依次实现盲目搜索、宽度搜索、启发式搜索解决九宫重排
（3）重点使用 A*算法实现九宫重排
（4）对三种搜索过程进行可视化分析展示

2. 【实验设备】

(1) 一台装有 windows10 系统的笔记本电脑。

(2) 集成 JDK 1.8.0_151 编译环境

(3) 使用 IntelliJ IDEA 进行编写程序

3. 【实验原理】

九宫重排问题，有 0-8 九个数字，首先给出初始状态和结束状态如下图所示，通如下图所示过移动 0 这个数字和自己相邻的上下左右的数字进行交换，从初始状态一步步移动达到结束状态的过程，分别使用盲目搜索、宽度搜索、启发式搜索分别解决这个问题，对这三种搜索算法的时间性能和空间性能进行分析

2	8	3
1	6	4
7	0	5

初始状态

1	2	3
8	0	4
7	6	5

终止状态

4. 【实验数据】

九宫重排问题中，对于初始状态的数据和结束状态的数据，一开始使用固定的数据进行

分析，后续利用可视化界面进行手动数据输入，提供多个样本对算法性能进行考察

5. 【实验分析】

（1）盲目搜索解决九宫重排问题

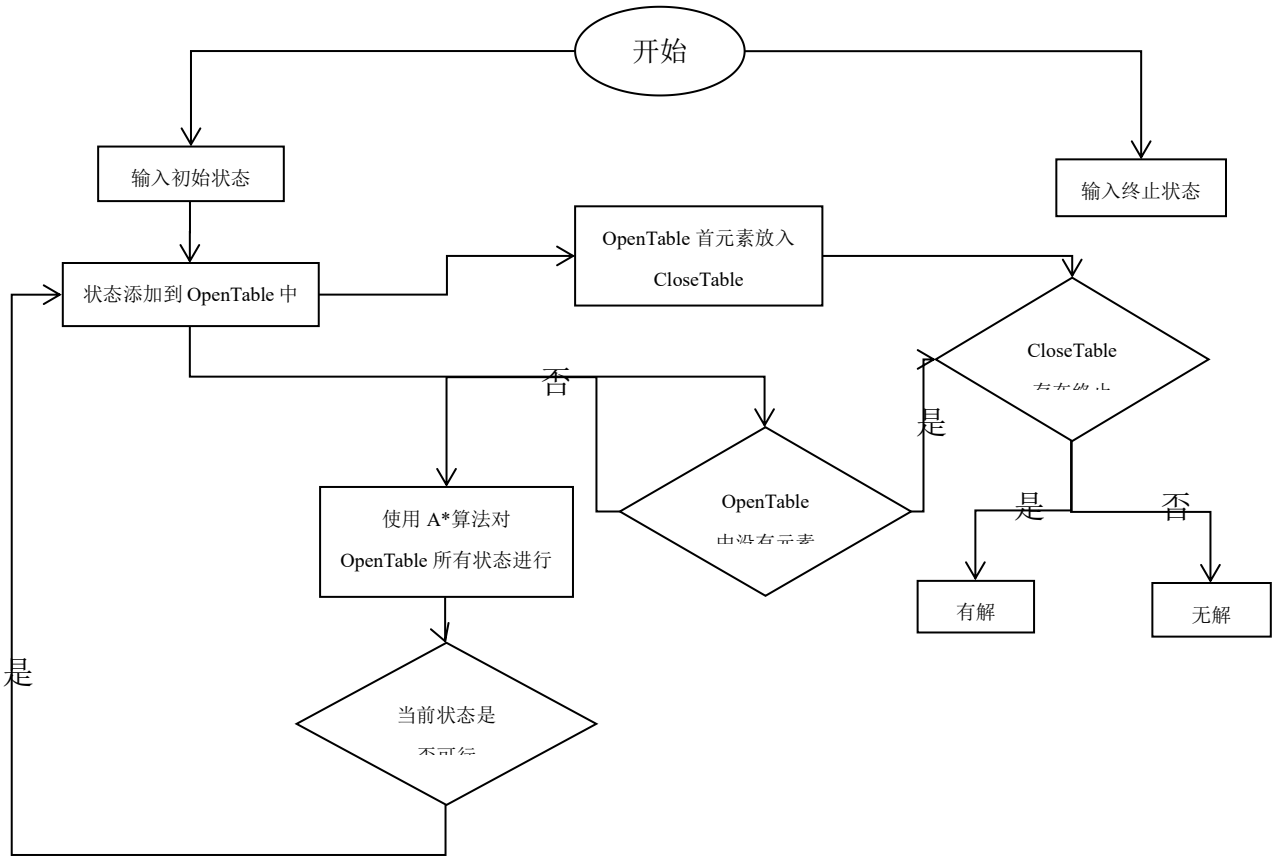
盲目搜索的算法这里使用的是深度优先搜索（DFS）,在搜索过程中一条路径走到底，如果行不通后需要返回上一步，搜索第二条路径。在实现深度优先搜索过程中使用递归调用实现。在应用于九宫重排的过程中当搜索到一种没有出现的情况，就一直沿着这条路搜索下去，直到这条路径不满足要求。在使用深度优先搜索过程中，由于反复递归调用，导致栈一直堆积，容易发生栈的溢出现象。

（2）宽度搜索解决九宫重排问题

在深度优先搜索过程中使用的是栈的数据结构对数据进行存储，在空间复杂度要求比较高，在使用宽度优先搜索（BFS）过程中，使用队列的数据结构，先访问完当前顶点的所有邻接点，然后再访问下一层的所有节点，每次访问队列顶端元素，对队列顶端元素的上下左右四种情况进行遍历，将符合要求的加入队列中，将队顶元素弹出。使用队列的数据结构很好的解决了栈溢出问题。

（3）启发式搜索解决九宫重排问题

进一步高效率解决九宫重排问题，在这里引入启发式搜索（Heuristically Search），使用 A*算法解决九宫重排问题，在这里引入评估函数，其中评估函数的选取是将实际代价+估算代价最小的点来作为一个反馈值，表示评估函数的性能，在选取估算代价最小的时候我选取的评价标准是曼哈顿距离（实际距离），评估函数具体公式为：f(n)=g(n)+h(n)，其中 g(n)表示实际代价，h(n):表示当前状态离目标状态的实际距离，如果启发函数满足 h(n) <= h*(n)，则可以证明当前问题有解，A*算法一定可以得到一个耗散值最小的结果。取 g(n)为层数，h(n)当前状态离目标状态的曼哈顿距离。



6. 【实验过程】

（1）盲目搜索

实验期间，我采用了 JavaScript 语言解决问题。首先使用盲目搜索来解决九宫重排问题，其中选取深度优先算法来基于初始状态对终止状态进行搜索。在使用深度优先算法的时候核心就是递归调用，其中递归函数原型为 DFS(x, y, star);x：表示当前 x 轴的位置，y：表示当前 y 轴的位置，star：表示当前九宫格的各个数的位置显示情况。以下为具体的实现过程。

（1）声明初始状态和终止状态，并将初始状态和终止状态转成字符串的实行进行存储，记录出现的情况

```
// 创建一个 开始状态
star = []
// 创建一个 结束状态
end = []
// 上下左右四个状态移动
dp = []
// 将初始状态 和 终止状态 加入
starString = "";
// 将初始状态 和 终止状态 加入
endString = "";
```

（2）在搜索过程中，我们需要判断当前状态是否出现过，对于已经出现过的状态我们因该是不需要在重复计算，为了更好的记录当前状态是否被访问过，我们将当前状态由数组转成字符串，并且将字符串存入 hashMap 中，在搜索过程中直接访问 map.containsKey(key)函数判断这个值是否之前出现过，由于在 JavaScript 中没有 hashMap 的数据结构，因此自己创建一个 hashMap

```
/**
 * 实现 hashMap
 */
function HashMap() {
    //定义长度
    var length = 0;
    //创建一个对象
    var obj = new Object();
    /**
     * 判断 Map 是否为空
     */
    this.isEmpty = function () {
        return length == 0;
    };
    /**
     * 判断对象中是否包含给定 Key
     */
    this.containsKey = function (key) {
        return (key in obj);
    };
    /**
```

```
     * 判断对象中是否包含给定的Value
     */
    this.containsValue = function (value) {
        for (var key in obj) {
            if (obj[key] == value) {
                return true;
            }
        }
        return false;
    };

    /**
     *向map中添加数据
     */
    this.put = function (key, value) {
        if (!this.containsKey(key)) {
            length++;
        }
        obj[key] = value;
    };
    /**
     * 根据给定的Key获得Value
     */
    this.get = function (key) {
        return this.containsKey(key) ? obj[key] : null;
    };
    /**
     * 根据给定的Key删除一个值
     */
    this.remove = function (key) {
        if (this.containsKey(key) && (delete obj[key])) {
            length--;
        }
    };
    /**
     * 获得Map中的所有Value
     */
    this.values = function () {
        var _values = new Array();
        for (var key in obj) {
            _values.push(obj[key]);
        }
        return _values;
    };
    /**
     * 获得Map中的所有Key
```

```
     */
    this.keySet = function () {
        var _keys = new Array();
        for (var key in obj) {
            _keys.push(key);
        }
        return _keys;
    };

    /**
     * 获得 Map 的长度
     */
    this.size = function () {
        return length;
    };
    /**
     * 清空 Map
     */
    this.clear = function () {
        length = 0;
        obj = new Object();
    };
}
```

（3）在具体实现搜索过程中，对当前状态进行上下左右搜索，判断数字 0 的位置，将数字 0 进行上下左右移动，对新的位置进行位置判断，对满足条件的位置进行下一步的搜索过程，具体代码实现如下所示

```
if (newx <= 2 && newx >= 0 && newy >= 0 && newy <= 2) {

    let value = temp[x][y];

    temp[x][y] = temp[newx][newy];

    temp[newx][newy] = value;

    t = "";

    for (let a = 0; a < temp.length; a++) {
        for (let b = 0; b < temp[a].length; b++) {
            t += temp[a][b] + "";
        }
    }

    if (map.containsKey(t) == false) {
```

```
        DFS(newx, newy, temp);
        // 将数据恢复

        let v = temp[newx][newy];

        temp[newx][newy] = temp[x][y];

        temp[x][y] = v;

        // console.log("diyici");
    } else {

        let v = temp[newx][newy];

        temp[newx][newy] = temp[x][y];

        temp[x][y] = v;

        continue;
    }
}
```

（4）如果搜索到的状态是终止状态，就需要结束递归调用，设置递归出口或者当前状态是访问过的我们也是不再进行重复访问的

```
// 到达了 终点
if (s == endString) {

    console.log("over");

    IsFind = true;

    return;
}

// 如果走过 就 行不通
if (map.containsKey(s) == true) {

    console.log("访问过");
    return;
}
```

（5）盲目搜索解决九宫重排问题实验结果
从键盘上输入数据进行测试的时候：

```
输入初始状态
2 8 3
1 6 4
7 0 5
输入终止状态
6 0 3
4 5 8
2 1 7
```

输出搜索结果展示

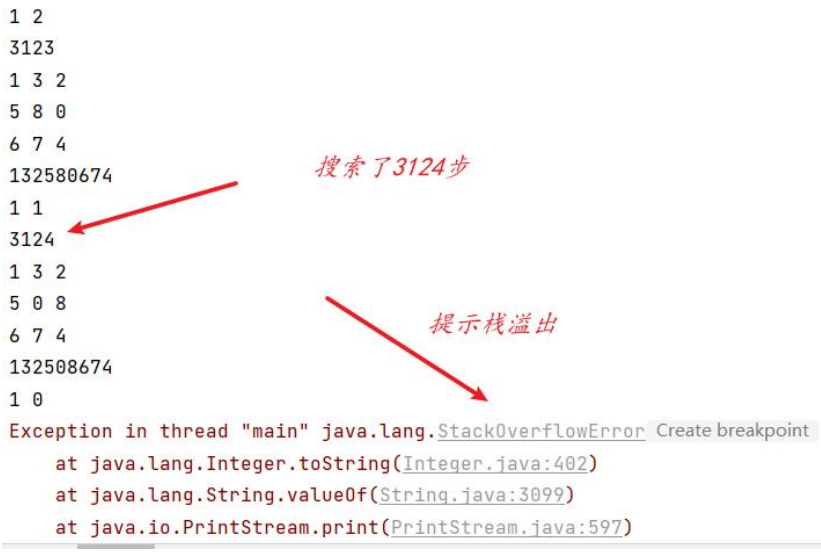
```
541
6 3 0
4 5 8
2 1 7
630458217
0 1
542
6 0 3
4 5 8
2 1 7
```

从搜索结果来看，进行搜索了 542 步才最后搜索到了结果，也就是说调用了 500 多次递归才找到最终结果，如果初始状态和终止状态差别比较大的时候，可能需要调用更多次递归才能找到最终状态。

从键盘上输入新的数据进行测试的时候：

```
输入初始状态
2 8 3
1 6 4
7 0 5
输入终止状态
1 2 3
8 0 4
7 6 5
```

输出结果展示



数据表明，无法搜索到最终结果，这个无法搜索到最终结果并不是初始状态无法到达终止状态，而是由于在当前状态下调用了 3124 次递归，导致栈的溢出无法进行下一次搜索，因此无法搜索到最终状态，为了验证猜想是递归过深导致于栈的溢出，于是进行测试

```
public void rec2num2(int i1,int i2)
{
    int i;
    for(i=0;i<1000;i++)
        i1++;
    for(i=0;i<1000;i++)
        i1--;

    System.out.println("This is "+i1+"recursion");
    rec2num2(i1+1,i2+1);
}
```

以上程序调用了 3141 次，导致栈的溢出，更加验证了栈的溢出是因为递归调用过深。于是考虑一下在九宫重排在最坏的情况下是总共需要进行调用次数

1	2	3
4	5	6
7	8	0

九宫重排各个数字排布

对于一个九宫格，0-8 的各个数字进行随机排列，总共的状态因该是存在 9!种，也是就 362880，当使用深度搜索进行解决九宫格问题的时候，由极大可能的可能性是会存在由于递归调用过深而导致栈的溢出，引起无法到达搜索状态的情况，这也为下面的宽度优先搜索和启发式搜索的提出说明了必要性。

（2）宽度优先搜索

在使用宽度优先搜索过程中，依然是使用 JavaScript 语言解决问题，在使用宽度优先搜索解决九宫重排过程中，总体的思想是将符合条件的元素加入队列中，当搜索该层完毕，将队列中的首个元素进行弹出，对这个状态的九宫格的 0 的位置进行上下左右四个方位进行搜索，并对当前状态进行判断，如果符合位置条件并且从未出现过，将这个状态加入队列中，一直循环下去，直到队列为空或者找到和最终状态相等的时候结束程序。以下为具体的实现过程。

(1) 声明初始状态和终止状态，并将初始状态和终止状态转成字符串的实行进行存储，记录出现的情况

```
let begin = new Array(3);
let end = new Array(3);
for (let i = 0; i < begin.length; i++) {
    begin[i] = new Array(3)
    end[i] = new Array(3)
}

// console.log("将 初始状态 和 终止状态 进行 赋值")

for (let i = 0; i < begin.length; i++) {
    for (let j = 0; j < begin[i].length; j++) {
        begin[i][j] = starNumber[3 * i + j]
        s += starNumber[3 * i + j]
        end[i][j] = endNumber[3 * i + j]
        answer += endNumber[3 * i + j]
    }
}
```

(2) 在 JavaScript 语言中由于没有自带的队列数据结构，于是创建 Queue 数据结构，将节点存入队列当中

```
// 基于数组封装队列类
function Queue() {
    // 属性
    this.items = []

    // 方法
    // 1.add():将元素加入到队列中
    Queue.prototype.add = element => {
        this.items.push(element)
    }

    // 2.pop():从队列中删除前端元素
    Queue.prototype.pop = () => {
        return this.items.shift()
    }

    // 3.poll():查看前端的元素
```

```
    Queue.prototype.poll = () => {
        return this.items[0]
    }

    // 4.isEmpty:查看队列是否为空
    Queue.prototype.isEmpty = () => {
        return this.items.length == 0;
    }

    // 5.size():查看队列中元素的个数
    Queue.prototype.size = () => {
        return this.items.length
    }

    // 6.toString():将队列中元素以字符串形式输出
    Queue.prototype.toString = () => {
        let resultString = ''
        for (let i of this.items) {
            resultString += i + ' '
        }
        return resultString
    }
}
```

(3) 使用 BFS 算法对九宫格进行调用，当拿到了初始状态之后，将初始状态加入到队列中，对队列中的第一个元素中 0 的位置进行左右上下移动，得到新的状态，并且将新的状态加入到队列中，代码实现如下所示：

```
while (!que.isEmpty()) {
    // 查看元素
    t = que.poll();
    // 删除元素
    que.pop()

    ch = t.s.split("");
    count++;

    // console.log("第" + count + "步")

    // console.log(ch)
    // 判断 位置
    // t.pos - 3 表示 空格上面
    if ((t.pos - 3) > 0) {
        swap(t.pos, t.pos - 3);
        // 将字符数组转成 字符串
        iString = ch.join("")
```

```
        if (answer === iString) {
            IsFind = true
            set.add(answer)
            count++;
            // console.log("第" + count + "步")
            // console.log(ch)
            return t.swap_num + 1;
        }

        if (!set.has(iString)) {
            set.add(iString);
            que.add(new BFSNode(iString, t.pos - 3, t.swap_num + 1));
        }

        swap(t.pos - 3, t.pos);
    }

    // 检查空格右侧的元素 如果时最右侧的 则 % 3 == 0
    if ((t.pos % 3) != 0) {
        swap(t.pos, t.pos + 1);
        iString = ch.join("")
        if (answer == iString) {
            IsFind = true
            set.add(answer)
            count++;
            // console.log("第" + count + "步")
            // console.log(ch)
            return t.swap_num + 1;
        }
        if (!set.has(iString)) {
            set.add(iString);
            que.add(new BFSNode(iString, t.pos + 1, t.swap_num + 1));
        }
        swap(t.pos + 1, t.pos);
    }

    // 如果是最左侧 现在 的位置 减去 1 就是 上一层 最后的那个元素的位置
    if ((t.pos - 1) % 3 != 0)   //检查空格左侧的元素
    {
        swap(t.pos, t.pos - 1);
        iString = ch.join("")
        if (answer == iString) {
            IsFind = true
            set.add(answer)
            count++;
            // console.log("第" + count + "步")
```

```
            // console.log(ch)
            return t.swap_num + 1;
        }
        if (!set.has(iString)) {
            set.add(iString);
            que.add(new BFSNode(iString, t.pos - 1, t.swap_num + 1));
        }
        swap(t.pos - 1, t.pos);
    }

    // t.pos+3<10 表示 不在 最下

    if (t.pos + 3 < 10)        //检查空格下面的元素
    {
        swap(t.pos, t.pos + 3);
        iString = ch.join("")
        if (answer == iString) {
            IsFind = true
            set.add(answer)
            count++;
            // console.log("第" + count + "步")
            // console.log(ch)
            return t.swap_num + 1;
        }
        if (!set.has(iString)) {
            set.add(iString);
            que.add(new BFSNode(iString, t.pos + 3, t.swap_num + 1));
        }
        swap(t.pos + 3, t.pos);
    }
}
```

(4) 使用手动输入数据展示结果：

```
请输入 九宫重排初始值
2 8 3
1 6 4
7 0 5
请输入 九宫重排结束值
1 2 3
8 0 4
7 6 5
```

运行结果展示

第 1 步:
2 8 3
1 6 4
7 0 5

第 2 步:
2 8 3
1 0 4
7 6 5

第 3 步:
2 8 3
1 6 4
7 5 0

第 4 步:
2 8 3
1 6 4
0 7 5

第 5 步:
2 0 3
1 8 4
7 6 5

第 6 步:
2 8 3
1 4 0
7 6 5

第 7 步:
2 8 3
0 1 4
7 6 5

第 8 步:
2 8 3
1 6 0
7 5 4

第 9 步:
2 8 3
0 6 4
1 7 5

第 10 步:

2 3 0
1 8 4
7 6 5

第 11 步:
0 2 3
1 8 4
7 6 5

第 12 步:
2 8 0
1 4 3
7 6 5

第 13 步:
2 8 3
1 4 5
7 6 0

第 14 步:
0 8 3
2 1 4
7 6 5

第 15 步:
2 8 3
7 1 4
0 6 5

第 16 步:
2 8 0
1 6 3
7 5 4

第 17 步:
2 8 3
1 0 6
7 5 4

第 18 步:
0 8 3
2 6 4
1 7 5

第 19 步:
2 8 3

```
6 0 4
1 7 5

第 20 步:
2 3 4
1 8 0
7 6 5

第 21 步:
1 2 3
0 8 4
7 6 5

第 22 步:
1 2 3
8 0 4
7 6 5
Process finished with exit code 0
```

(5) 对结果性能分析

当使用宽度优先搜索的时候，使用的是队列的数据结构，因此无需考虑栈的溢出情况，在输入同样的初始状态和终止状态的时候，深度优先搜索需要几百步或者是由于递归调用次数过多导致栈的溢出无法搜索到最终结果，但是使用宽度优先搜索，只需几十步就搜索到了最终状态，比较而言宽度优先搜索的性能优于深度优先搜索。

（3）启发式搜索

为了更高效率解决九宫重排问题，使用启发式搜索（Heuristically Search）求解九宫重排问题，使用 A*算法，在这里引入评估函数，其中评估函数的选取是将实际代价+估算代价最小的点来作为一个反馈值，表示评估函数的性能，估算代价最小我选取的评价标准是曼哈顿距离也就是实际距离，评估函数具体公式为：f(n)=g(n)+h(n)，其中 g(n)表示实际代价，h(n):表示当前状态离目标状态的实际距离，如果启发函数满足 h(n) <= h*(n)，则可以证明当问题有解时，A*算法一定可以得到一个耗散值最小的结果。取 g(n)为层数，h(n)当前状态离目标状态的曼哈顿距离。

曼哈顿距离公式：

$$\text{dist} = |X_1 - X_2| - |Y_1 - Y_2|$$

2	8	3
1	6	4
7	0	5

1	2	3
8	0	4
7	6	5

初始状态　　终止状态

对于上述初始状态转变终止状态中，在评估函数 f(n)=g(n)+h(n)中，g(0) = 0, h(0) = 1 + 2 + 0 + 1 + 1 + 0 + 0 + 1 + 0 = 6

(1)　　创建 openTable，closeTable 分别存储

openTable 表示用于记录可被访问的结点，closeTable 是记录已访问过的结点，如果在 closeTable 中存在一条路径是从初始状态到终止状态则表示存在路径并返回结束；如果在 openTable 中为空并且没有找到一条路径从初始状态到终止状态则返回失败并结束；其中对于 openTable 的创建是使用的是优先队列的数据结构

```
var openTable = new PriorityQueue((B, A) => ((A.value + A.depth) - (B.value + B.depth)))

var closeTable = new Stack()
```

每次访问 openTable 都是从其中取出最小值，这个最小值是每一个状态的评估函数的最小值

(2)　　创建优先队列和栈的数据结构

由于 JavaScript 中没有 Stack 和 PriorityQueue 的数据结构于是自己创建

```
class Stack {
    constructor() {
        this.items = []
    }

    // 新增元素
    add(el) {
        this.items.push(el)
    }

    // 删除栈顶的元素并返回其值
    pop() {
        return this.items.pop()
    }

    // 返回栈顶的元素
    peek() {
        return this.items[this.items.length - 1]
    }

    // 清空栈
    clear() {
        this.items = []
    }

    // 栈的大小
    size() {
```

```
        return this.items.length
    }

    // 栈是否为空
    isEmpty() {
        return this.items.length === 0
    }
}

class PriorityQueue {
    constructor(compare) {

        if (typeof compare !== 'function') {
            throw new Error('compare function required!')
        }

        this.data = []
        this.compare = compare

    }

    //二分查找 寻找插入位置
    search(target) {
        let low = 0, high = this.data.length
        while (low < high) {
            let mid = low + ((high - low) >> 1)
            if (this.compare(this.data[mid], target) > 0) {
                high = mid
            } else {
                low = mid + 1
            }
        }
        return low;
    }

    //添加
    add(elem) {
        let index = this.search(elem)
        this.data.splice(index, 0, elem)
        return this.data.length
    }

    //取出最优元素
    poll() {
        return this.data.pop()
    }
```

```
    //查看最优元素
    peek() {
        return this.data[this.data.length - 1];
    }
}
```

在使用优先队列的时候需要对比较函数进行重写，这个重写规则就是根据 f(n)=g(n)+h(n)，根据 f(n)的大小从小到大进行排列，因此每一次队列顶端元素都是评估函数的最小值

(3)　进行 A*搜索

① 访问 openTable，如果 openTable 为空，就结束搜索
② openTable 不为空，取出 openTable 的首个元素，加入到 closeTable 中，并且对首个元素进行上下左右搜索
③ 进行上下左右搜索的结果符合条件并且从未被访问过的加入到 openTable 中
④ 循环操作，直到 openTable 为空或者 closeTable 中存在一条路径是从初始状态到终止状态

具体代码实现如下：

```
while (true) {

    // 获取 open 表中最小的 那个值 也就是 第一个元素
    // 加入到 close 表中 之后 弹出去
    // console.log(openTable.peek())
    closeTable.add(openTable.peek());

    openTable.poll();

    if (equal(closeTable.peek(), Sg) == false) {
        createNode(closeTable.peek(), Sg);
    } else {
        IsFind = true
        break;
    }
}

function createNode(S, G) {
    /* 计算原状态下,空格所在的行列数，从而判断空格可以往哪个方向移动 */
    let blank; //定义 0 的下标
    for (blank = 0; blank < 9 && S.state[blank] != 0; blank++) ;//找到 0 的位置
    let x = Math.floor(blank / 3), y = blank % 3; //获取 0 所在行列编号
    for (let d = 0; d < 4; d++) //找到 S 扩展的子节点，加入 open 表中
    {
        let newX = x, newY = y;//

        tempNode = new Node();
```

```
        /*移动数字 0*/
        if (d == 0) newX = x - 1;  //向左
        if (d == 1) newY = y - 1;  //向下
        if (d == 2) newX = x + 1;   //向右
        if (d == 3) newY = y + 1;   //向上

        let newBlank = newX * 3 + newY; //0 新的位置

        if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) //如果移动合法
        {
            /* 交换格子的内容*/
            for (let i = 0; i < S.state.length; i++) {
                tempNode.state[i] = S.state[i];
            }
            //                tempNode = (Node)S.clone();

            // 将 0 和 要与 0 交换的那个值 进行交换
            tempNode.state[blank] = S.state[newBlank];
            tempNode.state[newBlank] = 0;

            if (S.parent != null && (S.parent).state[newBlank] == 0) //如果新节点
和爷爷节点一样，舍弃该节点
                continue;

            /* 把子节点都加入 open 表中 */
            tempNode.parent = S;
            tempNode.value = value(tempNode, G);
            tempNode.depth = S.depth + 1;
            openTable.add(tempNode);
        }
    }
}
```

(4)　　使用输入数据进行执行代码

```
请输入初始状态
2 8 3
1 6 4
7 0 5
请输入终止状态
1 2 3
8 0 4
7 6 5
```

对结果进行展示

```
至少要移动 5 步
2 8 3
1 6 4
7 0 5
-----------

2 8 3
1 0 4
7 6 5
-----------

2 0 3
1 8 4
7 6 5
-----------

0 2 3
1 8 4
7 6 5
-----------

1 2 3
0 8 4
7 6 5
-----------

1 2 3
8 0 4
7 6 5
-----------
Process finished with exit code 0
```

使用 A*算法进行求解同深度优先搜索和宽度优先搜索相同的初始状态和终止状态，需要的步骤只有 5 步，进一步缩小的时间复杂性，将性能再次提升。

（4）九宫重排问题的可视化操作

为了更加直观的展示和对比这三种搜索方法的步骤和性能，采用可视化来展示搜索过程。其中前端使用 html 语言进行编写，后端使用 JavaScript 进行编写。

手动输入的数据，接收到之后使用数组进行缓存，当执行当前的搜索算法时，将数组中的数据传送当前函数，进行数据反馈，当执行完成搜索函数之后返回结果之后，将结果进行解构，将其中的值分别获取出来，使用 v-for 指令在页面进行渲染

```
<div class="DisplayMain">
    <div class="Table" v-for="item in TableList">
        <h4>{{item.title}}</h4>
        <div v-for="(it,index) in item.state">
            <span>{{item.state[index * 3 + 0]}}</span>
            <span>{{item.state[index * 3 + 1]}}</span>
            <span>{{item.state[index * 3 + 2]}}</span>
        </div>
    </div>
</div>
```

输入固定的数据对比以下三种搜索算法的性能
当输入数据为

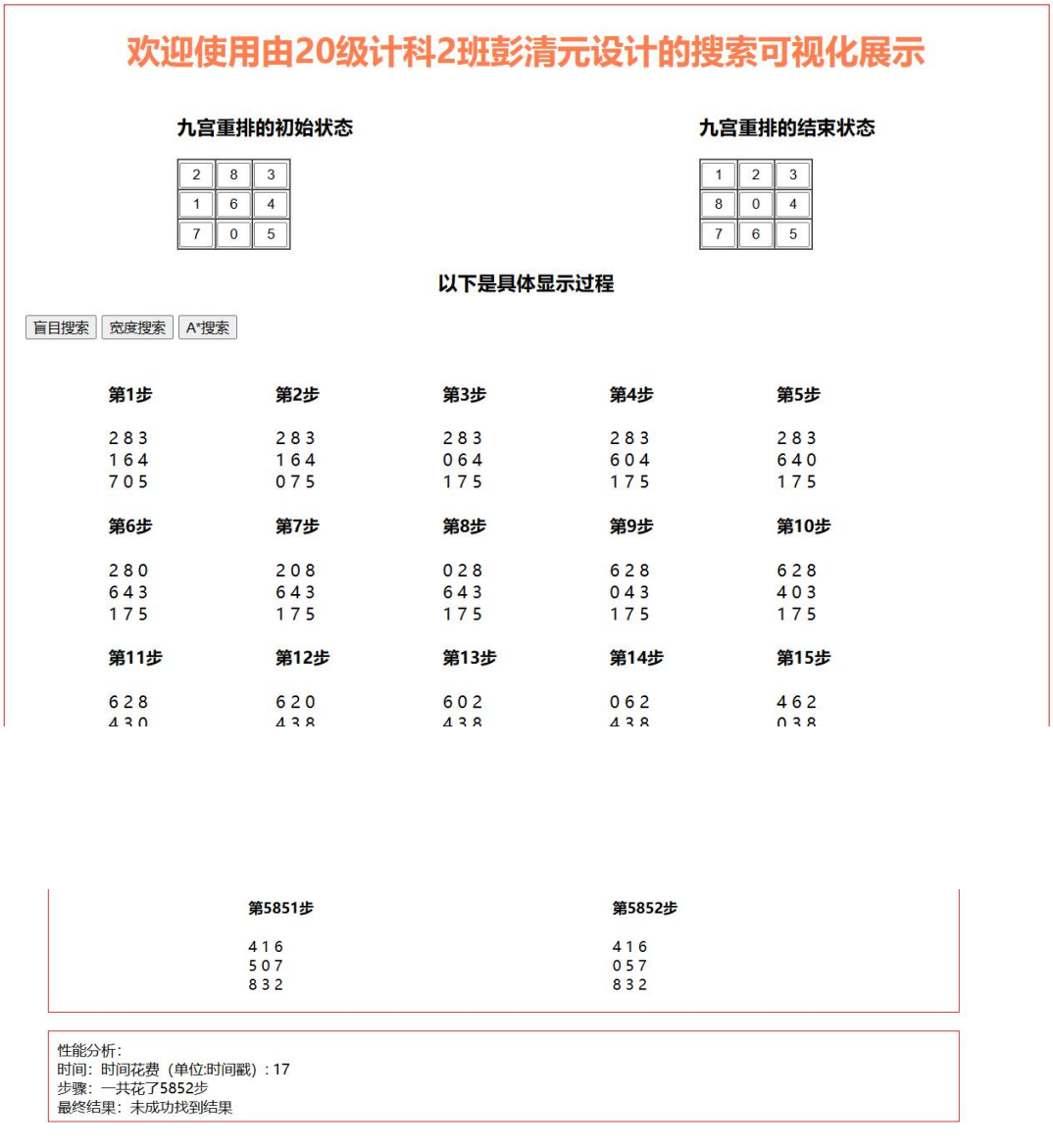
2	8	3
1	6	4
7	0	5

初始状态

1	2	3
8	0	4
7	6	5

终止状态

(1) 使用盲目搜索(深度搜索)



使用深度优先搜索执行了 5852 次之后由于递归调用深度过深，导致栈进行溢出，无法进行搜索；

(2) 使用宽度优先搜索对上述数据进行搜索

欢迎使用由20级计科2班彭清元设计的搜索可视化展示

九宫重排的初始状态

2	8	3
1	6	4
7	0	5

九宫重排的结束状态

1	2	3
8	0	4
7	6	5

以下是具体显示过程

盲目搜索 宽度搜索 A*搜索

第1步

2 8 3
1 6 4
7 0 5

第2步

2 8 3
1 0 4
7 6 5

第3步

2 8 3
1 6 4
7 5 0

第4步

2 8 3
1 6 4
0 7 5

第5步

2 0 3
1 8 4
7 6 5

第6步

2 8 3
1 4 0
7 6 5

第7步

2 8 3
0 1 4
7 6 5

第8步

2 8 3
1 6 0
7 5 4

第9步

2 8 3
0 6 4
1 7 5

第10步

2 3 0
1 8 4
7 6 5

第11步

0 2 3
1 8 4
7 6 5

第12步

2 8 0
1 4 3
7 6 5

第13步

2 8 3
1 4 5
7 6 0

第14步

0 8 3
2 1 4
7 6 5

第15步

2 8 3
7 1 4
0 6 5

第16步

2 8 0
1 6 3
7 5 4

第17步

2 8 3
1 0 6
7 5 4

第18步

0 8 3
2 6 4
1 7 5

第19步

2 8 3
6 0 4
1 7 5

第20步

2 3 4
1 8 0
7 6 5

第21步

1 2 3
0 8 4
7 6 5

第22步

2 0 8
1 4 3
7 6 5

第23步

2 8 3
1 4 5
7 0 6

第24步

8 0 3
2 1 4
7 6 5

第25步

2 8 3
7 1 4
6 0 5

第26步

2 0 8
1 6 3
7 5 4

第27步

2 0 3
1 8 6
7 5 4

第28步

2 8 3
0 1 6
7 5 4

第29步

2 8 3
1 5 6
7 0 4

第30步

8 0 3
2 6 4
1 7 5

第31步

2 0 3
6 8 4
1 7 5

第32步

2 8 3
6 4 0
1 7 5

第33步

2 8 3
6 7 4
1 0 5

第34步

2 3 4
1 0 8
7 6 5

第35步

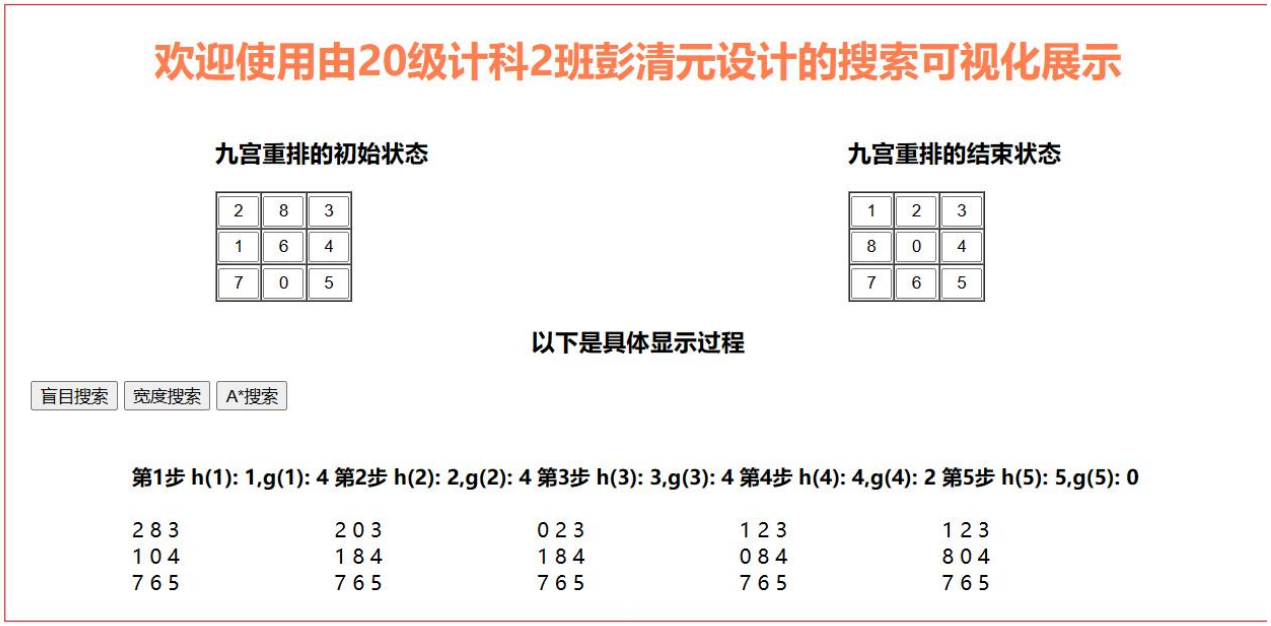
2 3 4
1 8 5
7 6 0

第36步

1 2 3
8 0 4
7 6 5

性能分析:
时间: 时间花费（单位:时间戳）: 0
步骤: 一共花了36步
最终结果: 成功找到结果

使用宽度优先搜索执行了 36 次之后寻找到了和结束状态相同的值。
(3) 使用启发式搜索搜索对上述数据进行搜索



性能分析:
时间: 时间花费（单位:时间戳）: 0
步骤: 一共花了5步
最终结果: 成功找到结果

使用启发式搜索，对上述输入只需要 5 步就可以找到结束状态，使用启发式搜索进一步简化了搜索过程

(4) 将上述三种搜索进行对比

搜索方案	是否可以到达终态	步骤	时间戳
深度优先搜索	否	5852	17
宽度优先搜索	是	36	0
启发式搜索	是	5	0

7. 【实验心得】

通过这次实验使用三种搜索算法分别解决九宫重排问题，在使用这三种搜索算法的时候各自遇到不同的问题，最终经过不断地调试得以解决。

(1) 在盲目搜索过程中使用的是深度优先搜索，使用深度优先搜索过程中，如果初始状态和结束状态差别比较大时候，进行递归调用深度较大的时候，可能出现栈溢出现象，在第一次出现栈溢出导致无法得出最终结果，并未想到是由于递归深度较深的原因，于是使用 try { DFS(x, y, star); } catch (e) { console.log(e) },对异常行捕获，发现是栈溢出导致，

通过分析这个九宫重排总体可能性的情况与递归调用深度最大值进行比较得出导致出现异常的原因是栈溢出所导致的；
(2) 在使用宽度优先搜索解决九宫重排问题时，是使用比较常规的队列的数据结构进行存储，需要自己创建一个队列的数据结构，使用创建好的队列存储 Node，每次取出队头元素，对队头元素进行上下左右遍历，将符合条件的 Node 加入到队列中，直达队列为空或者找到终止状态停止搜索；
(3) 在使用启发式搜索解决九宫重排问题时，选用的是 A*算法，在 A*算法中选取的评估函数是 f(n)=g(n)+h(n)，对于 h(n)的选择是选择曼哈顿距离作为评价标准，对于存储 f(n)的数据结构使用的优先队列，将 f(n)作为比较函数的依据，对优先队列中的比较函数根据 f(n)进行重写，在通过访问 openTable 和 closeTable 过程中，在 java 中，对象是引用类型的，因此把对象 a 的值赋给另一个变量 b 时，实际上是把 a 的内存地址赋给 b 了，因此当时在使用 java 编程过程中直接对 tempNode 进行赋值时，在后续修改 tempNode 导致原来的值也跟着发生变化了，导致结果一直出现错误，于是采用 Debug 工具单步调试最后发现在 Java 中对象的直接赋值是赋值引用，会将地址也复制给对方，导致修改中间值会导致原始值发生变化，因此使用循环变量赋值解决问题；
(4) 为了更加形象展示各种搜索效果，因此制作可视化界面，通过输入初始状态和终止状态，点击对应的搜索按钮，将会出现当前搜索的流程图以及相应的性能分析，制作可视化界面使用的是 html 和 JavaScript 语言，通过将输入的数据由 v-model 绑定的参数获取，传送给 JavaScript 中对应搜索函数,进行执行,将函数的搜索结果返回给 html 中的 list,使用 v-for 将 list 中的数据进行渲染
(5) 通过这次实验，在编写深度优先搜索时，对递归深度调用的限度有了一个更加的清晰认识，并对 Debug 工具有了进一步掌握，通过使用编写宽度优先搜索时，对队列的数据结构有了进一步加强，明白了宽度优先遍历在九宫重排的应用，编写 A*算法搜索解决九宫重排的过程中，对选取评估函数有了更加深刻理解，对为什么选取曼哈顿的实际意义进一步掌握，对 OpenTable 和 CloseTable 的使用过程使用编程思想加以实现，最后再配上 html 的数据可视化效果使运行结果更加容易理解

8. 【附录】

(1) 盲目搜索

```
/**
 * 深度搜索
 */
// 创建一个 3 * 3 的 迷宫
count = 0;
// 创建一个 开始状态
star = []
// 创建一个 结束状态
end = []
// 上下左右四个状态移动
dp = []
// 将初始状态 和 终止状态 加入
starString = "";
// 将初始状态 和 终止状态 加入
endString = "";
```

```
IsFind = false;

ansString = []

/**
 *  实现 hashMap
 */
function HashMap() {
    //定义长度
    var length = 0;
    //创建一个对象
    var obj = new Object();

    /**
     *  判断 Map 是否为空
     */
    this.isEmpty = function () {
        return length == 0;
    };

    /**
     *  判断对象中是否包含给定 Key
     */
    this.containsKey = function (key) {
        return (key in obj);
    };

    /**
     *  判断对象中是否包含给定的 Value
     */
    this.containsValue = function (value) {
        for (var key in obj) {
            if (obj[key] == value) {
                return true;
            }
        }
        return false;
    };

    /**
     *向 map 中添加数据
     */
    this.put = function (key, value) {
        if (!this.containsKey(key)) {
            length++;
```

```
        }
        obj[key] = value;
};

/**
 * 根据给定的 Key 获得 Value
 */
this.get = function (key) {
        return this.containsKey(key) ? obj[key] : null;
};

/**
 * 根据给定的 Key 删除一个值
 */
this.remove = function (key) {
        if (this.containsKey(key) && (delete obj[key])) {
                length--;
        }
};

/**
 * 获得 Map 中的所有 Value
 */
this.values = function () {
        var _values = new Array();
        for (var key in obj) {
                _values.push(obj[key]);
        }
        return _values;
};

/**
 * 获得 Map 中的所有 Key
 */
this.keySet = function () {
        var _keys = new Array();
        for (var key in obj) {
                _keys.push(key);
        }
        return _keys;
};

/**
 * 获得 Map 的长度
 */
this.size = function () {
```

```
        return length;
    };

    /**
     * 清空 Map
     */
    this.clear = function () {
        length = 0;
        obj = new Object();
    };
}

map = new HashMap();

function DFS(x, y, arr) {

    if (IsFind == true) return;

    count++;

    s = ""

    for (let i = 0; i < arr.length; i++) {
        for (let j = 0; j < arr[i].length; j++) {
            s += arr[i][j];
        }
    }
    // 到达了 终点
    if (s == endString) {

        map.put(s, 1);
        ansString.push(s)
        console.log("over");

        IsFind = true;

        return;
    }

    // 如果走过 就 行不通
    if (map.containsKey(s) == true) {

        console.log("访问过");
        return;
    }
```

```
map.put(s, 1);
ansString.push(s)

let Mapsize = map.size()

temp = arr;

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

    state = dp[i];

    let newx = x + state[0];
    let newy = y + state[1];

    // console.log(newx + " " + newy);

    if (newx <= 2 && newx >= 0 && newy >= 0 && newy <= 2) {

        let value = temp[x][y];

        temp[x][y] = temp[newx][newy];

        temp[newx][newy] = value;

        t = "";

        for (let a = 0; a < temp.length; a++) {
            for (let b = 0; b < temp[a].length; b++) {
                t += temp[a][b] + "";
            }
        }

        if (map.containsKey(t) == false) {

            DFS(newx, newy, temp);
            //  将数据恢复

            let v = temp[newx][newy];

            temp[newx][newy] = temp[x][y];

            temp[x][y] = v;

            // console.log("diyici");
        } else {
```

```
                    let v = temp[newx][newy];

                    temp[newx][newy] = temp[x][y];

                    temp[x][y] = v;

                    continue;
                }
            }
        }
    }

function UnibformedSearch() {
    let x = 0;
    let y = 0;

    // 将初始状态加入到 map 中
    for (let i = 0; i < star.length; i++) {
        for (let j = 0; j < star[i].length; j++) {
            if (star[i][j] == 0) {
                x = i;
                y = j;
                break;
            }
        }
    }
    try {
        DFS(x, y, star);
    } catch (e) {
        console.log(e)
    }

}

function main2(starNumber, endNumber) {

    // 创建一个 3 * 3 的 迷宫
    count = 0;
// 创建一个 开始状态
    star = []
// 创建一个 结束状态
    end = []
// 上下左右四个状态移动
    dp = []
// 将初始状态 和 终止状态 加入
    starString = "";
```

```
// 将初始状态 和 终止状态 加入
    endString = "";

    map.clear()

    IsFind = false;

    ansString = []

    let time1 = new Date().getTime()
    // 初始状态
    star = new Array(3);
    // 结束状态
    end = new Array(3);
    // 位置偏移
    dp = new Array(4);

    /**
     * 创建对应的数据组
     */
    for (let i = 0; i < star.length; i++) {
        star[i] = new Array(3)
        end[i] = new Array(3)
    }

    for (let i = 0; i < 4; i++) {
        dp[i] = new Array(2)
    }

    /**
     * 数值初始化
     */
    for (let i = 0; i < star.length; i++) {
        for (let j = 0; j < star[i].length; j++) {
            star[i][j] = starNumber[3 * i + j]
            end[i][j] = endNumber[3 * i + j]
        }
    }

    dp[0][0] = 0
    dp[0][1] = -1

    dp[1][0] = 0
    dp[1][1] = 1

    dp[2][0] = -1
```

```
    dp[2][1] = 0


    dp[3][0] = 1
    dp[3][1] = 0


    // 将终止状态加入到 map 中
    for (let i = 0; i < end.length; i++) {
        for (let j = 0; j < end[i].length; j++) {
            endString += end[i][j] + "";
        }
    }

    // 使用 盲目搜索 进行 搜索展示
    UnibformedSearch();

    let time2 = new Date().getTime()
    let time = time2 - time1

    let obg = {
        time,
        ansString,
        IsFind
    }
    return obg

    return ansString

}
```

(2) 宽度搜索

```
/**
 * 创建一些全局变量
 */
ch = []
s = ''
answer = ''
count = 0
set = new Set()

/**
 *      创建一个节点 node
 */
class BFSNode {
    s = '';
```

```
        pos = 0;
        swap_num = 0;

        constructor(ss, ppos, num) {
            this.s = ss;
            this.pos = ppos;
            this.swap_num = num;
        }
}

/**
 * 创建一个 交换函数
 * @param x
 * @param y
 */
function swap(x, y) {
        t = ch[x - 1];
        ch[x - 1] = ch[y - 1];
        ch[y - 1] = t;
}

/**
 * 创建一个 Queue
 * @constructor
 */

// 基于数组封装队列类
function Queue() {
        // 属性
        this.items = []

        // 方法
        // 1.add():将元素加入到队列中
        Queue.prototype.add = element => {
            this.items.push(element)
        }

        // 2.pop():从队列中删除前端元素
        Queue.prototype.pop = () => {
            return this.items.shift()
        }

        // 3.poll():查看前端的元素
        Queue.prototype.poll = () => {
            return this.items[0]
        }
```

```
    // 4.isEmpty:查看队列是否为空
    Queue.prototype.isEmpty = () => {
        return this.items.length == 0;
    }

    // 5.size():查看队列中元素的个数
    Queue.prototype.size = () => {
        return this.items.length
    }

    // 6.toString():将队列中元素以字符串形式输出
    Queue.prototype.toString = () => {
        let resultString = ''
        for (let i of this.items) {
            resultString += i + ' '
        }
        return resultString
    }
}

/**
 * 使用 bfs 进行搜索
 * @param e
 */
function bfs(e) {

    // console.log(e)
    let que = new Queue();

    let iString = ''

    que.add(e)

    let t = null;

    while (!que.isEmpty()) {
        // 查看元素
        t = que.poll();
        // 删除元素
        que.pop()

        ch = t.s.split("");
        count++;

        // console.log("第" + count + "步")
```

```
// console.log(ch)
// 判断 位置
// t.pos - 3 表示 空格上面
if ((t.pos - 3) > 0) {
    swap(t.pos, t.pos - 3);
    // 将字符数组转成 字符串
    iString = ch.join("")
    if (answer === iString) {
        IsFind = true
        set.add(answer)
        count++;
        // console.log("第" + count + "步")
        // console.log(ch)
        return t.swap_num + 1;
    }

    if (!set.has(iString)) {
        set.add(iString);
        que.add(new BFSNode(iString, t.pos - 3, t.swap_num + 1));
    }
    swap(t.pos - 3, t.pos);
}

// 检查空格右侧的元素 如果时最右侧的 则 % 3 == 0
if ((t.pos % 3) != 0) {
    swap(t.pos, t.pos + 1);
    iString = ch.join("")
    if (answer == iString) {
        IsFind = true
        set.add(answer)
        count++;
        // console.log("第" + count + "步")
        // console.log(ch)
        return t.swap_num + 1;
    }
    if (!set.has(iString)) {
        set.add(iString);
        que.add(new BFSNode(iString, t.pos + 1, t.swap_num + 1));
    }
    swap(t.pos + 1, t.pos);
}

// 如果是最左侧 现在 的位置 减去 1 就是 上一层 最后的那个元素的位置
if ((t.pos - 1) % 3 != 0)    //检查空格左侧的元素
{
```

```
                swap(t.pos, t.pos - 1);
                iString = ch.join("")
                if (answer == iString) {
                    IsFind = true
                    set.add(answer)
                    count++;
                    // console.log("第" + count + "步")
                    // console.log(ch)
                    return t.swap_num + 1;
                }
                if (!set.has(iString)) {
                    set.add(iString);
                    que.add(new BFSNode(iString, t.pos - 1, t.swap_num + 1));
                }
                swap(t.pos - 1, t.pos);
            }

            // t.pos+3<10  表示  不在  最下

            if (t.pos + 3 < 10)            //检查空格下面的元素
            {
                swap(t.pos, t.pos + 3);
                iString = ch.join("")
                if (answer == iString) {
                    IsFind = true
                    set.add(answer)
                    count++;
                    // console.log("第" + count + "步")
                    // console.log(ch)
                    return t.swap_num + 1;
                }
                if (!set.has(iString)) {
                    set.add(iString);
                    que.add(new BFSNode(iString, t.pos + 3, t.swap_num + 1));
                }
                swap(t.pos + 3, t.pos);
            }
        }
        return -1;
    }

    /**
     * 主函数
     * @param starNumber
     * @param endNumber
     */
```

```
function main1(starNumber, endNumber) {

    let time1 = new Date().getTime()
    ch = []
    s = ''
    answer = ''
    count = 0

    IsFind = false

    let begin = new Array(3);
    let end = new Array(3);
    for (let i = 0; i < begin.length; i++) {
        begin[i] = new Array(3)
        end[i] = new Array(3)
    }

    // console.log("将 初始状态 和 终止状态 进行 赋值")

    for (let i = 0; i < begin.length; i++) {
        for (let j = 0; j < begin[i].length; j++) {
            begin[i][j] = starNumber[3 * i + j]
            s += starNumber[3 * i + j]
            end[i][j] = endNumber[3 * i + j]
            answer += endNumber[3 * i + j]
        }
    }

    // console.log("初始状态是 == > " + s)
    // console.log("结束状态是 == > " + answer)

    n = new BFSNode(s, s.indexOf("0") + 1, 0)
    // console.log(n)
    set.add(s);
    bfs(n);

    let time2 = new Date().getTime()


    let time = time2 - time1

    let obg = {
        time: time,
        set,
        IsFind
    }
```

```
    return obg
}

// main1()
```

(3) A*搜索

```
/*
 * @Descripttion: my project
 * @version: 1.0
 * @Author: Typecoh
 * @Date: 2023-05-16 14:46:26
 * @LastEditors: Typecoh
 * @LastEditTime: 2023-05-16 15:54:58
 */
class Node {
    state = [];
    parent = null;
    value = 0;
    depth = 0;
}

class Stack {
    constructor() {
        this.items = []
    }

    // 新增元素
    add(el) {
        this.items.push(el)
    }

    // 删除栈顶的元素并返回其值
    pop() {
        return this.items.pop()
    }

    // 返回栈顶的元素
    peek() {
        return this.items[this.items.length - 1]
    }

    // 清空栈
    clear() {
        this.items = []
    }
```

```
    // 栈的大小
    size() {
        return this.items.length
    }

    // 栈是否为空
    isEmpty() {
        return this.items.length === 0
    }
}

class PriorityQueue {
    constructor(compare) {

        if (typeof compare !== 'function') {
            throw new Error('compare function required!')
        }

        this.data = []
        this.compare = compare

    }

    //二分查找 寻找插入位置
    search(target) {
        let low = 0, high = this.data.length
        while (low < high) {
            let mid = low + ((high - low) >> 1)
            if (this.compare(this.data[mid], target) > 0) {
                high = mid
            } else {
                low = mid + 1
            }
        }
        return low;
    }

    //添加
    add(elem) {
        let index = this.search(elem)
        this.data.splice(index, 0, elem)
        return this.data.length
    }

    //取出最优元素
    poll() {
```

```
            return this.data.pop()
        }

        //查看最优元素
        peek() {
            return this.data[this.data.length - 1];
        }
}

// function

var openTable = new PriorityQueue((B, A) => ((A.value + A.depth) - (B.value + B.depth)))
var closeTable = new Stack()

Path = new Stack();
var num = 9

var count1 = 0;
var count2 = 0;

function Judge(S, G) {

        S.parent = null;
        S.value = 0;
        S.depth = 0;

        G.parent = null;
        G.value = 0;
        G.depth = 0;

        // System.out.println("请输入初始状态");
        // 0 1 2 3 4 5 6 7 8

        // 1 2 0 3 4 5 6 7 8
        // S.state[0] = 0
        // S.state[1] = 1
        // S.state[2] = 2
        // S.state[3] = 3
        // S.state[4] = 4
        // S.state[5] = 5
        // S.state[6] = 6
        // S.state[7] = 7
        // S.state[8] = 8
        //
        // G.state[0] = 1
        // G.state[1] = 2
```

```
    // G.state[2] = 0
    // G.state[3] = 3
    // G.state[4] = 4
    // G.state[5] = 5
    // G.state[6] = 6
    // G.state[7] = 7
    // G.state[8] = 8

    return 0;
}

function equal(S, G)   //判断两个方阵是否相等。
{
    for (let i = 0; i <= 8; i++) {
        if (S.state[i] != G.state[i]) {
            return false;
        }
    }
    return true;
}

function value(A, G)//   计算每个数字当前状态与最终状态的曼哈顿距离 之和作为代价
{

    //count 记录所有棋子移动到正确位置需要的步数
    let count = 0;

    let begin = new Array(3);
    let end = new Array(3);
    for (let i = 0; i < begin.length; i++) {
        begin[i] = new Array(3)
        end[i] = new Array(3)
    }
    //将一维数组的值转赋值给二维数组
    for (let i = 0; i < begin.length; i++) {
        for (let j = 0; j < begin[i].length; j++) {
            begin[i][j] = A.state[3 * i + j];
            end[i][j] = G.state[3 * i + j];
        }
    }

    for (let i = 0; i < 3; i++)     //检查当前图形的正确度
        for (let j = 0; j < 3; j++) {
            if (begin[i][j] != end[i][j]) {
                for (let k = 0; k < 3; k++) for (let w = 0; w < 3; w++) if (begin[i][j] == end[k][w]) count =
(count + Math.abs(i - k * 1.0) + Math.abs(j - w * 1.0)); //累加计算每个数字当前状态与最终状态的曼哈顿距离
```

```
                } else {
                    continue;
                }
            }

    return count + A.depth;     //返回估计值
}

function createNode(S, G) {
    /*  计算原状态下,空格所在的行列数，从而判断空格可以往哪个方向移动  */
    let blank; //定义 0 的下标
    for (blank = 0; blank < 9 && S.state[blank] != 0; blank++) ;//找到 0 的位置
    let x = Math.floor(blank / 3), y = blank % 3; //获取 0 所在行列编号
    for (let d = 0; d < 4; d++) //找到 S 扩展的子节点，加入 open 表中
    {
        let newX = x, newY = y;//

        tempNode = new Node();

        /*移动数字 0*/
        if (d == 0) newX = x - 1;    //向左
        if (d == 1) newY = y - 1;    //向下
        if (d == 2) newX = x + 1;     //向右
        if (d == 3) newY = y + 1;     //向上

        let newBlank = newX * 3 + newY; //0 新的位置

        if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) //如果移动合法
        {
            /*  交换格子的内容*/
            for (let i = 0; i < S.state.length; i++) {
                tempNode.state[i] = S.state[i];
            }
            //                      tempNode = (Node)S.clone();

            // 将  0  和  要与  0  交换的那个值  进行交换
            tempNode.state[blank] = S.state[newBlank];
            tempNode.state[newBlank] = 0;

            if (S.parent != null && (S.parent).state[newBlank] == 0) //如果新节点和爷爷节点一样，舍弃该节
点
                continue;

            /*  把子节点都加入 open 表中  */
            tempNode.parent = S;
            tempNode.value = value(tempNode, G);
```

```
                tempNode.depth = S.depth + 1;
                openTable.add(tempNode);
            }
        }
    }


function main(star, end) {

    IsFind = false

    Path.clear()

    let time1 = new Date().getTime()

    S0 = new Node();
    Sg = new Node();

    for (let i = 0; i < star.length; i++) {
        S0.state[i] = star[i]
        Sg.state[i] = end[i]
    }

    // S0.state[0] = 0
    // S0.state[1] = 1
    // S0.state[2] = 2
    // S0.state[3] = 3
    // S0.state[4] = 4
    // S0.state[5] = 5
    // S0.state[6] = 6
    // S0.state[7] = 7
    // S0.state[8] = 8
    //
    // Sg.state[0] = 1
    // Sg.state[1] = 2
    // Sg.state[2] = 0
    // Sg.state[3] = 3
    // Sg.state[4] = 4
    // Sg.state[5] = 5
    // Sg.state[6] = 6
    // Sg.state[7] = 7
    // Sg.state[8] = 8

    Judge(S0, Sg);

    openTable.add(S0);
```

```
while (true) {

    // 获取 open 表中最小的 那个值 也就是 第一个元素
    // 加入到 close 表中 之后 弹出去
    // console.log(openTable.peek())
    closeTable.add(openTable.peek());

    openTable.poll();

    if (equal(closeTable.peek(), Sg) == false) {
        createNode(closeTable.peek(), Sg);
    } else {
        IsFind = true
        break;
    }
}

// console.log("star")

//临时变量暂存队前数据
tempNode = closeTable.peek();   //back   返回队列的最后一个元素即最后入队的元素

while (tempNode.parent != null) {
    Path.add(tempNode);//入栈
    tempNode = (tempNode.parent);//指向父节点
}

Path.add(tempNode);

// console.log(Path.size())

// /* 输出方案 */
// while (Path.size() != 0) {
//     // console.log(Path.peek().state)
//     // for (let i = 0; i <= 8; i++) {
//     // console.log(Path.peek().state[i] + " ")
//     // if ((i + 1) % 3 == 0)
//     //     console.log()
//     // // System.out.print(Path.peek().state[i] + " ");
//     // // if ((i + 1) % 3 == 0)
//     // //     System.out.println();
//     // }
//     Path.pop();
// }
```

```
    let time2 = new Date().getTime()

    let time = time2 - time1

    let obg = {
        time: time, Path,IsFind
    }

    console.log(obg)

    return obg;
}


// main()
```

（4） 可视化实现

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>20 级计科二班彭彭清元</title>
    </head>

    <style>
        * {
            box-sizing: border-box;
        }

        #container {
            width: 1000px;
            /*height: 600px;*/
            margin: 20px auto;
            border: 1px solid red;
        }

        .state {
            display: flex;
            flex-direction: row;
            justify-content: space-around;
        }
```

```
        .header {
            width: 100%;
        }

        .select {
            margin-left: 20px;
        }

        td {
            text-align: center;
        }

        .DisplayMain {
            width: 80%;
            margin: 20px auto;
            display: flex;
            flex-direction: row;
            justify-content: space-around;
            flex-wrap: wrap;
        }

        .Table {
            width: 20%;
        }

        .footer {
            padding: 10px;
            width: 1000px;
            height: 100px;
            margin: 20px auto;
            border: 1px solid red;
        }

        input {
            width: 32px;
            height: 24px;
            text-align: center;
        }

    </style>
    <body>

        <div id="APP">
            <div id="container">
                <div>
```

```
                <h1 style="text-align: center; color: coral">欢迎使用由 20 级计科 2 班彭清元设计的搜索可视化展示</h1>
            </div>
            <div class="main">
                <div class="state">
                    <div class="left">
                        <h3>九宫重排的初始状态</h3>
                        <table border="1" height="50" width="100" cellspacing="0">
                            <tr>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[0]">
                                </td>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[1]">
                                </td>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[2]">
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[3]">
                                </td>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[4]">
                                </td>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[5]">
                                </td>
                            </tr>
                            <tr>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[6]">
                                </td>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[7]">
                                </td>
                                <td>
                                    <input type="text" width="100%" v-model="StartNumber[8]">
                                </td>
                            </tr>
                        </table>
                    </div>
                    <div class="right">
                        <h3>九宫重排的结束状态</h3>
                        <table border="1" height="50" width="100" cellspacing="0">
```

```
                    <tr>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[0]">
                        </td>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[1]">
                        </td>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[2]">
                        </td>
                    </tr>
                    <tr>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[3]">
                        </td>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[4]">
                        </td>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[5]">
                        </td>
                    </tr>
                    <tr>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[6]">
                        </td>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[7]">
                        </td>
                        <td>
                            <input type="text" width="100%" v-model="EndNumber[8]">
                        </td>
                    </tr>
                </table>
            </div>
        </div>
        <div class="display">
            <h3 style="text-align: center">以下是具体显示过程</h3>
            <div class="header">
                <div class="select">
                    <button class="btn1" @click="btn1">盲目搜索</button>
                    <button class="btn2" @click="btn2">宽度搜索</button>
                    <button class="btn3" @click="btn3">A*搜索</button>
                </div>
            </div>
            <div class="DisplayMain">
```

```
                                <div class="Table" v-for="item in TableList">
                                    <h4>{{item.title}} <span v-show="bt3 ==
true">{{item.depth}},{{item.value}}</span></h4>
                                    <div v-for="(it,index) in item.state">
                                        <span>{{item.state[index * 3 + 0]}}</span>
                                        <span>{{item.state[index * 3 + 1]}}</span>
                                        <span>{{item.state[index * 3 + 2]}}</span>
                                    </div>
                                </div>
                            </div>
                        </div>
                    </div>
                    <div class="footer">
                        <span>性能分析：</span>
                        <div>时间：{{time}}</div>
                        <div>步骤：{{step}}</div>
                        <div>最终结果：{{MSG}}</div>
                    </div>
                </div>


                <script src="vue.js"></script>
                <script src="A.js"></script>
                <script src="BFS.js"></script>
                <script src="DFS.js"></script>
                <script>

                    const vm = new Vue({
                        el: "#APP",

                        data: {
                            list: [],
                            TableList: [],
                            step: "",
                            time: '',
                            StartNumber: [],
                            EndNumber: [],
                            IsFind: "",
                            MSG: "",
                            bt1:false,
                            bt2:false,
                            bt3:false,
                        },

                        methods: {
```

```
btn3() {
        this.bt1 = false
        this.bt2 = false
        this.bt3 = true
        this.list = []
        this.list = main(this.StartNumber, this.EndNumber)
        // this.list = main(StartNumber,EndNumber)
        // console.log(this.list)
        // console.log(main([2,8,3,1,6,4,7,0,5],[1,2,3,8,0,4,7,6,5]))
        this.time = this.list.time
        this.TableList = []
        let items = this.list.Path.items
        let len = items.length - 1
        for (let i = items.length - 2; i >= 0; i--) {
            this.TableList.push({
                title: "第" + (len - i) + "步",
                state: items[i].state,
                depth: "h(" + (len - i) + "): " + items[i].depth,
                value: "g(" + (len - i) + "): " + items[i].value,
            })
        }

        this.time = "时间花费（单位:时间戳）: " + this.time
        this.step = "一共花了" + len + "步"
        this.IsFind = this.list.IsFind
        if (this.IsFind == true) this.MSG = "成功找到结果"
        else this.MSG = "未成功找到结果"
},
btn2() {
        this.bt1 = false
        this.bt2 = true
        this.bt3 = false
        this.TableList = []
        this.list = []

        let set = []

        set = main1(this.StartNumber, this.EndNumber)
        // set = main1([2, 8, 3, 1, 6, 4, 7, 0, 5], [1, 2, 3, 8, 0, 4, 7, 6, 5])

        // console.log(set)
        set.set.forEach(value => {
            this.list.push(value.split(""))
        })

        console.log(this.list)
```

```
        let items = this.list
        let len = items.length
        for (let i = 0; i < len; i++) {
            this.TableList.push({
                title: "第" + (i + 1) + "步",
                state: items[i],
            })
        }

        this.time = "时间花费（单位:时间戳）: " + set.time
        this.step = "一共花了" + len + "步"

        this.IsFind = set.IsFind
        if (this.IsFind == true) this.MSG = "成功找到结果"
        else this.MSG = "未成功找到结果"
    },
    btn1() {
        this.bt1 = true
        this.bt2 = false
        this.bt3 = false
        this.TableList = []
        this.list = []

        let map = []

        // set = main1(this.StartNumber, this.EndNumber)
        map = main2(this.StartNumber, this.EndNumber)
        // map = main2([2, 8, 3, 1, 6, 4, 7, 0, 5], [1, 2, 3, 8, 0, 4, 7, 6, 5])

        // console.log(map)

        map.ansString.forEach(value => {
            this.list.push(value.split(""))
        })

        console.log(this.list)

        let items = this.list
        let len = items.length
        for (let i = 0; i < len; i++) {
            this.TableList.push({
                title: "第" + (i + 1) + "步",
                state: items[i],
            })
        }
```

```
                this.time = "时间花费（单位:时间戳）: " + map.time
                this.step = "一共花了" + len + "步"
                this.IsFind = map.IsFind
                if (this.IsFind == true) this.MSG = "成功找到结果"
                else this.MSG = "未成功找到结果"
            }
        }
    })
</script>
</body>
</html>
```

实验的过程不是消极的观察，而是积极的、有计划的探测，一个成功的实验需要的是眼光、勇气和毅力。

——曲阜师范大学名誉校长

诺贝尔奖获得者丁肇中