CS203B Project

01 基本信息

姓名	学号
何其佳	12111211
邱俊杰	12111831
王越	12112610

02 `DSAA_Pro`文件基本结构

1. Jar包：lib文件夹下的`algs4.jar`，若使用`IDEA`进行编译，请先将其添加到Projcet Structure内
2. script：

- Solver：贪婪搜索算法，standard input输入`terminal`或`gui`分别进入终端或GUI模式（主算法）
- Solver2：BFS算法，无需额外standard input输入，默认进入`gui`模式（对比算法1）
- Sovler3：DFS算法，无需额外standard input输入，默认进入`gui`模式（对比算法2）
- Generator：随机棋盘生成器，能够根据合法的输入随机生成可解的棋盘，并可以自定义块的类型和数量
- Board：GUI辅助类
- MyGUI：GUI类

03 数据结构与算法解析（贪婪搜索）

3.1 数据结构

在数据结构的选取过程中，为了节省切片调用的时间，我们使用一维`int[]`数组对棋盘进行存储。我们使用了`HashMap`对棋子类型进行一一映射，并定义为`11 12 21 22 -1 0`四种类型的棋子，分别代表1x1，1x2，2x1，2x2block的pivot（左上角元素），以及非pivot[1]的block元素和空格。

对于每一个节点，我们创建了`Node`类，其中的主要参数有棋盘`int[] gridBoard`，父到子节点的操作`String lastOperation`，父节点`Node parent`，以及节点到最终状态的曼哈顿距离`int dist`，其中，曼哈顿距离的数学定义如下：

$$dist = \sum_{i,x_i \neq 0} |x_i - x_{it}| + |y_i - y_{it}|$$

其中(x_i, y_i)是棋盘中非0点的坐标，(x_{it}, y_{it})是棋盘为(x_i, y_i)在最终状态下的位置，通过这样定义，我们就可以令优先搜索的节点为dist较小的节点，从而让搜索能够在趋势上逐渐靠近最终答案，实践证明该优化策略效果显著。

> 思考：不能将空格与最终相对位置考虑进dist的原因
>
> 实践观察后我们发现，若将空格计算在内，很可能造成空格过早地归位，使得其很难上移去挪动上层尚未归位的棋子

为了实现全局节点的动态选择，而不是单独选择最深层子节点中dist最佳者，我们使用了`MinPQ`对全局节点进行了存储并设计了相应的算法提供支持。

为了判断节点是否已经遍历，我们使用了`HashMap`对节点进行查重。在`HashCode`的选取上，我们直接采用了了`Array.toStiring(gridBoard)`后的棋盘`String`的`HashCode`。

3.2 算法分析

1. 代码API

方法	功能
public Solver(Scanner sc)	将输入的数据处理，棋盘存入gridBoard，调用GreedySearch 方法
public static void main(String[] args)	主方法，调用scanner以及输入异常处理
public boolean GreedySearch()	内存溢出及剪枝处理，调用search方法，找到答案后回溯正解过程并返回true
private static boolean search(Node node)	对node内的棋盘进行检查与移动，如果得到了和答案一样的棋盘，返回true
private static void search1x1(int id, Node node)	对能移动的1*1的块（id）做处理，查看能否移动，能移动且移动后的棋盘未出现过则进行存储（2x2，1x2，2x1的search 方法类似）
private static boolean checkULBound(int id)	部分检查当前id是否在棋盘内部
public static int findIndex(int[] arr, int key)	找到数组arr中值等于key的键的索引
public int compareTo(Node o)	重写compareTo方法，比对值为node的曼哈顿距离
public Node(int[] gridBoard, String lastOperation, Node parent)	初始化node

2. 伪代码：

```
while (minPQ!=0) {
    nodesToCheck = search node(minPQ.delMin());
    for (nodesToCheck) {
        check if not passed & not the result {
            add to minPQ;
        } if is result {
            break;
        } else {
            if visited {
```

```
                continue;
            } else {
                add to minPQ;
            }
        }
    }
}

recursively find the whole path;
allocate GUI;
```

3.复杂度分析：

对于单独一个结点的处理，因为棋盘中的空格的数量（k）是有限的，每个空格对应11个检查点[2]，而每个块只存在4种移动方式，故整体的遍历次数是常数级别 (k x 11 x 4)，此时时间主要受到MinPQ insert 的影响，时间复杂度为 O(mn log mn) （m,n分别是棋盘的长与宽）

对整体而言，每个单独的结点都是从minPQ中取出，如果minPQ中没有查找正确答案，循环会一直进行下去直到找到最终答案为止，在最坏的情况下，需要遍历的情况为 (m x n)!，整体的时间复杂度为 O((mn)! (mn) log(mn))，但在未剪枝的情况下，当 m x n 过大时，系统出现内存溢出的情况。

受到内存溢出的影响，当MinPQ的size大于50万时，只保留node的棋盘到达终态的曼哈顿距离最小的一万种，也就是截取MinPQ的前一万个元素，删除后面的元素，因为后面的49万个元素的曼哈顿距离较大，与终态的距离越远，故得出正确路径的概率几乎为0 (when n is sufficiently large)，能够做删除处理。

对于优化后的情况，因为MaxPQ的长度是不断变化的，故n值也不断变化，复杂度无法分析，介于O(0)~O((m*n)! mn log(mn))之间

04 数据结构与算法解析（BFS & DFS）

基于贪婪搜索的算法，更改存储结点的数据结构，并不再使用曼哈顿距离，即能够得到BFS，和DFS算法。具体情况如下：

BFS搜索

将MinPQ更改成queue，把生成的新棋盘，查重后将棋盘以结点的形式从队尾入队，查找新数据时从队头弹出结点，此时数据只有在遍历完父节点生成的全部子节点后才会遍历子节点生成的孙子节点，层层遍历，实现广度搜索

在没有实现贪心搜索的情况下，无法对queue进行合理的剪枝，最坏情况下的时间复杂度是O((m*n)!)

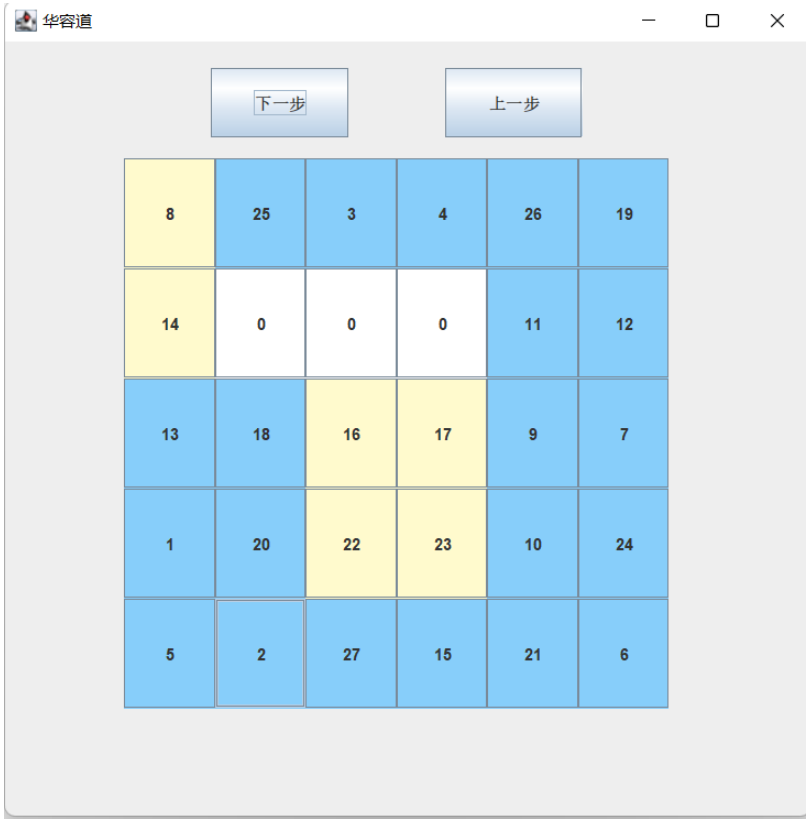
DFS搜索

将MinPQ更改成stack，把生成的新棋盘，查重后将棋盘以结点的形式丢入栈中，查找新数据时从栈中弹出结点，此时搜索时只会沿着一条分支不断往下查找，实现深度优先搜索

在没有实现贪心搜索的情况下，无法对stack进行合理的剪枝，最坏情况下的时间复杂度是O((m*n)!)

05 GUI模块

运行结束后，如果存在正解，弹出gui模块如下：下一步出现下一步棋，上一部返回上一步棋，蓝色块为1x1模块，白色为空格，米黄色为1x2, 2x1, 2x2模块



06 实验

批注：对于样例4等大于3*3的情况，Solver2 和 Solver 3 可能会出现内存溢出（BFS & DFS），Solver（minPQ）能够运行

样例1

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

```
2 2
1 2
0 3
0
Read In Successfully!
Yes
1
3 L
```

样例2

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

```
2 2
2 1
0 3
0
Read In Successfully!
No
```

样例3

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

```
3 3
1 2 3
4 0 0
7 5 6
1
5 1*2
Read In Successfully!
Yes
1
5 U
```

样例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
```

```
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
Read In Successfully!
Yes
30
11 L
8 D
4 R
7 R
3 R
15 D
11 D
7 D
3 R
1 U
9 U
13 U
17 L
11 L
10 L
12 U
```

07 Bonus1：随机棋盘生成器

棋盘生成器通过输入来确定随机棋盘的大小以及特殊块的个数、位置等。运行`Generator`的main方法，输入合法的棋盘参数，就可以自动生成棋盘以及计算结果并在GUI上进行展示

> 示例：

```
5 6
2
1 2*2
11 2*1
Read In Successfully!
----------Successfully Generated Data----------
5 6
1 2 3 0 5 6
7 8 9 4 12 18
13 14 21 10 11 0
20 25 27 16 17 22
19 26 28 15 23 24
2
1 2*2
11 2*1
-----------------------------------------------------
```

其中，第一行输入的为纵轴长`m`和横轴长`n`两个参数

第二行输入的是特殊block的个数

后面两行输入对应的特殊block以及其在结果棋盘上的位置，输入格式类似题目设置的标准

08 Bonus2: 算法对比

我们一共采用了三种算法，分别是贪婪搜索, BFS 以及 DFS。分别对应的 class 文件为Solver, Solver2, Solver3 具体的实现方式在03和04模块中做出介绍，下面是三种算法的对比关系表：

性质\算法	**贪婪搜索**	**广度优先搜索**	**深度优先搜索**
待遍历棋盘存储结构	minPQ	Queue	Stack
下一步优先移动顺序	曼哈顿距离	层序	深度
运行速度	快	慢	慢
可运行规模	7*7	3*3	3*3
最坏情况下的时间复杂度(遍历所有情况)	$O((mn)! * mn * log(mn))$	$O((mn)!)$	$O((mn)!)$
实验样例4的运行时间	8ms	4396ms	Exception in Heap

其中，Solver2和Sovler3的输入格式同Solver，均为题目所设标准

---

1. pivot指的是每种block最左上角的元素 ↩

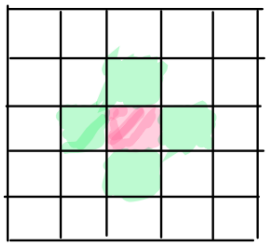
"□"为空格

"□"为可能在移动的块左上角的块 

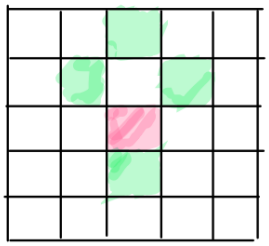
四种类型的块所对应的检索区

2.

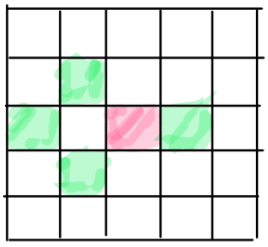
1×1 块 "□"



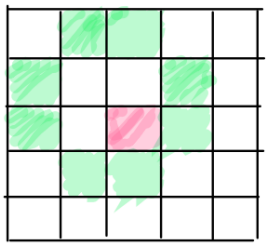
2×1 块 "日"



1×2块 "凵"



2×2块 "田"



所有情况