README

所需库

numpy

queue

A*算法的基本思路

f：g+h

g：从起点到当前状态的实际代价

h：从当前状态到目标状态的预估代价

1. **算法初始化：** 将起点放入一个Open Set中，并将该状态的代价设为0。同时，将Closed Set置为空。
2. **循环直到找到路径或open set为空：**
    - **选择状态：** 从open set中选择一个状态，该节点是当前f值最小的状态。
    - **目标检查：** 检查当前状态是否为目标状态。如果是目标状态，则路径已找到，算法结束。
    - **生成子状态：** 对于当前状态，生成其子状态。
    - **更新状态信息：** 对于每个相邻状态，计算新的g值和h值，更新子状态的f值。
    - **状态分类：**
        - 如果子状态已经在Closed Set中，跳过。
        - 如果子状态不在Open Set中，将其加入Open Set，并记录其父状态为当前状态。
3. **构建路径：** 当目标状态被找到时，通过回溯从目标状态到起点的路径，并得到最终路径。

Q1：

---

f：g+h

g：从每个状态到其子状态都花费一步

h：启发式函数的定义为**不在其正确位置的数字**的个数

```
def heuristic(state):
    count = 0
    for i in range(9):
        if state[i] != goal_state[i]:
            count += 1
    return count
```

具体的设计思路

- **算法初始化：** 读入数据，将起点放入一个Open Set中，并将该状态的代价设为0。同时，将Closed Set置为空。
- **对于每个状态的数字排序，使用列表来存储，使用python中的字典来存储状态的f值和g值,以下是初始状态的样例**

```
{"f": heuristic(initial_state), "g": 0, "state": initial_state}
```

- **循环直到找到最短路径的g值或开放列表为空：**
  - **选择当前状态：** 从open set中选择一个状态，该状态是当前f值最小的状态。
  - **目标检查：** 检查当前状态是否为目标状态。如果是目标状态，则最短路径的g值已找到，返回g值，算法结束。如果不是目标状态，将其转移至closed set。
  - **生成子状态：** 对于当前状态，生成其子状态。在算法开头定义了元素0的移动方向，在算法内计算0的二维坐标，如果0移动后，其坐标在正常范围内，则可以生成这个子状态。
  - **更新状态信息：** 对于每个子状态，计算新的g值和h值，更新子状态的f值。
  - **状态分类：**

- 如果子状态已经在Closed Set中，跳过。
- 如果子状态不在Open Set中，将其加入Open Set。

测试样例输出

对于五个测试样例，其输出分别为

```
1
1
2
1
3
```

Q2:

---

f：g+h

g：从该层到其他层的花费

h：启发式函数的定义为**当前层所有通道中最小的那个通道长度**，这个数值是必定小于到达目标层的所需的长度的

```
def heuristic(floor, matrix):
    return min(matrix[floor][floor + 1:], default=np.inf)
```

具体的设计思路

明显的，这是个使用A*算法搜索图的问题。

- **算法初始化：** 将起点放入优先队列中，并将该状态的代价设为0。由于我们不需要输出具体的路径，所以不维护closed set。
- **对于每个状态的存储：** 使用列表来存储状态，第一个元素为f值，第二个为g值，第三个为层数

```
[0 + heuristic(1, matrix), 0, 1]
```

- **循环直到result中有K个元素或优先队列为空：**
  - **选择状态：** 从优先队列中选择当前f值最小的状态。
  - **目标检查：** 检查当前状态是否为目标状态。如果是目标状态，则将g加入到结果中，再检测result的个数是否满足K，满足则退出。
  - **生成子状态：** 对于当前状态，生成其子状态，加入优先队列。

测试样例输出

对于五个测试样例，其输出分别为

```
3
3
-1
-1
```

```
4
5
6
7
```

```
5
5
6
6
7
7
```

```
4
4
5
-1
-1
-1
-1
```

```
5
5
6
6
6
8
-1
-1
```