**电梯调度系统设计报告**

**基本任务**

本电梯调度系统使用python语言，模拟在一个20层楼宇中，五部互联电梯之间的自动调度和运行过程。采用面向对象设计的调度算法与多线程技术，并结合 PyQt5 实现图形可视化用户操作界面。通过模拟用户在电梯内与楼层外发出请求，实现对电梯的合理分配与状态控制。

项目已托管于[github仓库](https://github.com/4768686/OS_elevator_dispatching_system)

**功能描述**

**1. 电梯内部功能**

每部电梯都配备了如下控制按钮和显示组件：

* **楼层按钮**：1-20层共计20个数字按钮，表示电梯内用户的目标楼层。
* **报警按钮：**模拟电梯故障状态，按下该按钮后电梯停止运行，不再响应，关闭后电梯重新恢复相应。
* **状态显示**：
  + 当前楼层数（LCD 数码管）。
  + 上升/下降/静止状态指示。
  + 报警状态。（“⚠”表示电梯报警）
  + 门状态（“<-- -->”表示开门，“--> <--”表示关门）。

**2. 楼层外部功能**

每层楼的电梯门口提供：

* **上行请求按钮**（▲）：表示楼层中用户希望上行。
* **下行请求按钮**（▼）：表示楼层中用户希望下行。
* **请求按钮状态反馈**：按钮变色表示请求已发出，处理后恢复默认颜色。

**实现方式**

**1. 技术栈**

* **语言**：Python 3.8.1
* **图形界面库**：PyQt5
* **线程管理**：Python QThread
* **调度核心**：自定义调度线程 + 状态检查线程

**2. 多线程模型**

* **5个电梯线程（WorkThread）**：负责每部电梯的楼层更新与请求处理。

class WorkThread(QThread):

    trigger = pyqtSignal(int)

    def \_\_init\_\_(self, the\_int):

        super(WorkThread, self).\_\_init\_\_()

        self.int = the\_int

        self.trigger.connect(check)

* **1个调度线程（DispatcherThread）**：负责将全局的上行/下行请求分配给合适的电梯。

**3. 关键全局变量**

elevator\_target: 每部电梯的目标楼层集合

floor\_requests: 所有楼层的上下请求及分配状态

state: 每部电梯的当前运行状态（上=1，停=0，下=-1）

should\_pause: 电梯是否需要开门（开门 = 1）

pause: 电梯是否启用（默认 = 1）

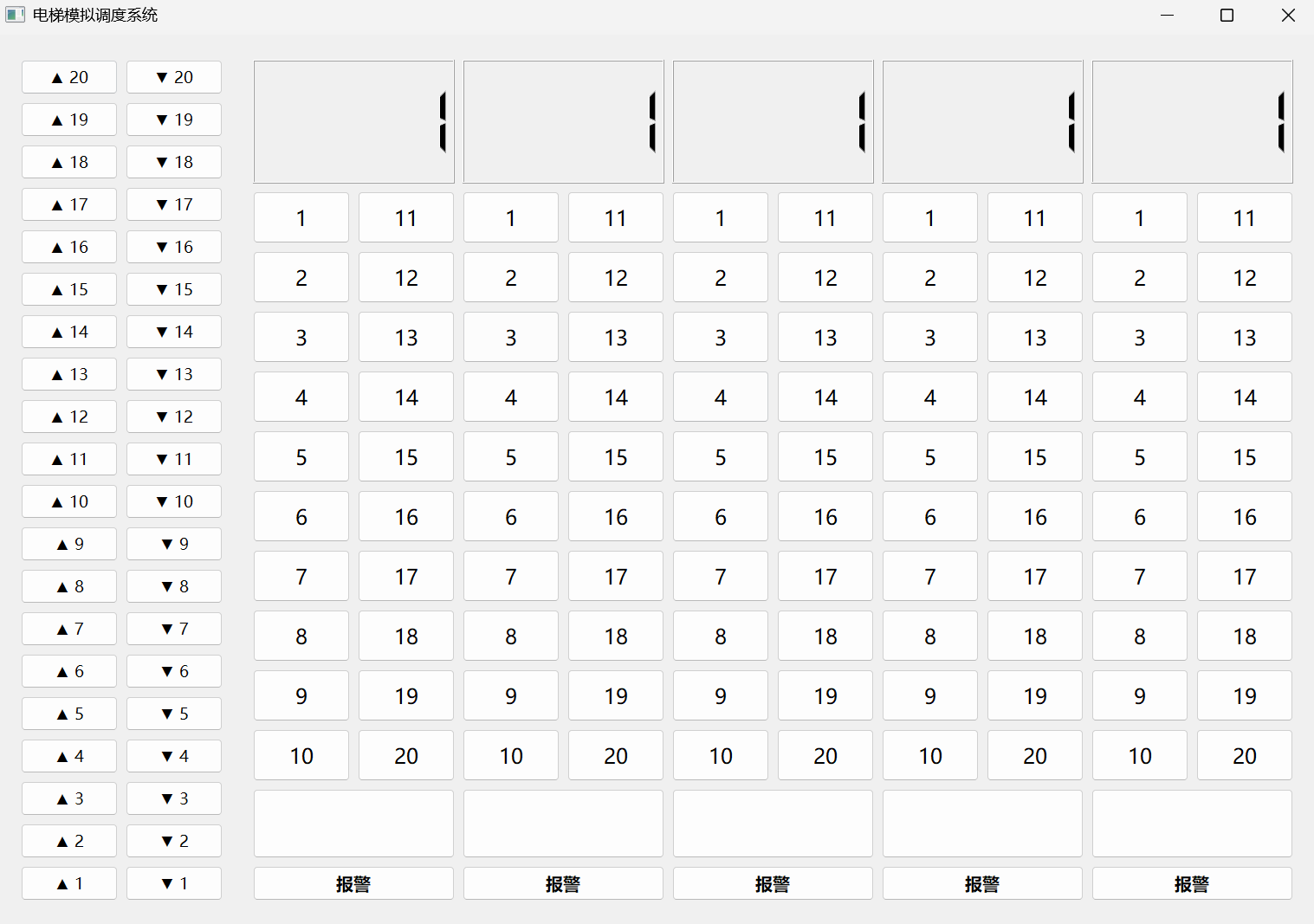
alarm\_state:电梯是否报警(默认 = false)

**界面设计**

通过 PyQt5 创建图形化交互界面：

* **界面左右布局**：
  + 左侧：每层楼层之间的上/下按钮。
  + 右侧：每部电梯的楼层控制按钮、报警按钮、状态与楼层显示。
* **交互体验**：
  + 按钮点击即发送请求，并改变颜色表示已发出。
  + 电梯经过楼层后，自动响应开门/关门、按钮状态复位等。

示意图（程序界面）：



楼层按钮

楼层显示

电梯按钮

电梯开关门状态显示

报警按钮

**调度算法**

* 1. **楼层请求处理流程**

用户点击 ▲ / ▼ → floor\_requests 添加请求 → 调度线程分配 → 对应电梯响应

* 1. **调度策略核心逻辑**

该电梯调度系统程序使用的是一种**基于方向与距离优先的贪心调度算法**，它的主要目的是将楼层请求（上下行）合理分配给当前最适合的电梯，从而实现响应速度快、减少电梯空跑的目标，调度核心定义在DispatcherThread类中。

**dispatch()**

定时检查未被分配的楼层请求，并尝试调用 select\_elevator() 为其分配最优电梯。

    def dispatch(self):

        unassigned = {f: d for f, (d, e) in floor\_requests.items() if e is None}

        '''遍历所有未分配的楼层请求'''

        for flr, direction in unassigned.items():

            best = self.select\_elevator(flr, direction)

            if best is not None:

                elevator\_target[best].add(flr)  *# 分配给最合适的电梯*

                floor\_requests[flr] = (direction, best)  *# 更新分配信息*

**select\_elevator(flr, direction)**

为给定楼层 flr 和方向 direction（1=上，-1=下）选择最优电梯：

    def select\_elevator(self, flr, direction):

        candidates = []

        '''state[]:电梯状态,0空闲,1向上,-1向下'''

        for i in range(5):

*# 只考虑未处于报警状态的电梯*

            if not alarm\_state[i]:

                if state[i] == 0:

                    candidates.append((abs(floor[i] - flr), i))  *# 计算电梯与楼层之间的距离*

                elif state[i] == direction:

                    if direction == 1 and floor[i] <= flr:

                        candidates.append((flr - floor[i], i))

                    elif direction == -1 and floor[i] >= flr:

                        candidates.append((floor[i] - flr, i))

        if candidates:

            return min(candidates)[1]  *# 返回最优电梯*

        return None

* 1. **状态机切换**

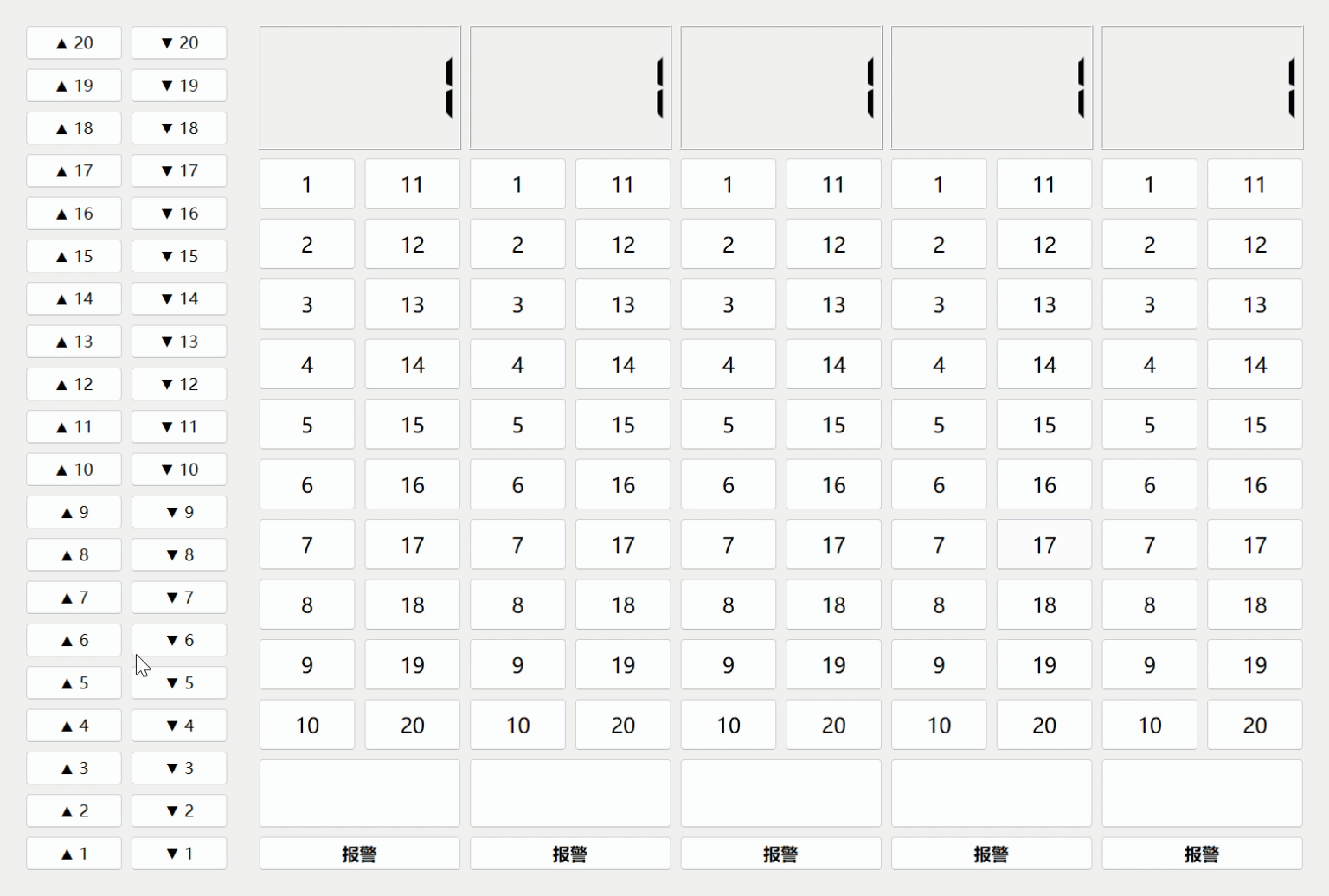
电梯根据当前目标楼层集合及当前位置自动判断下一步行动：

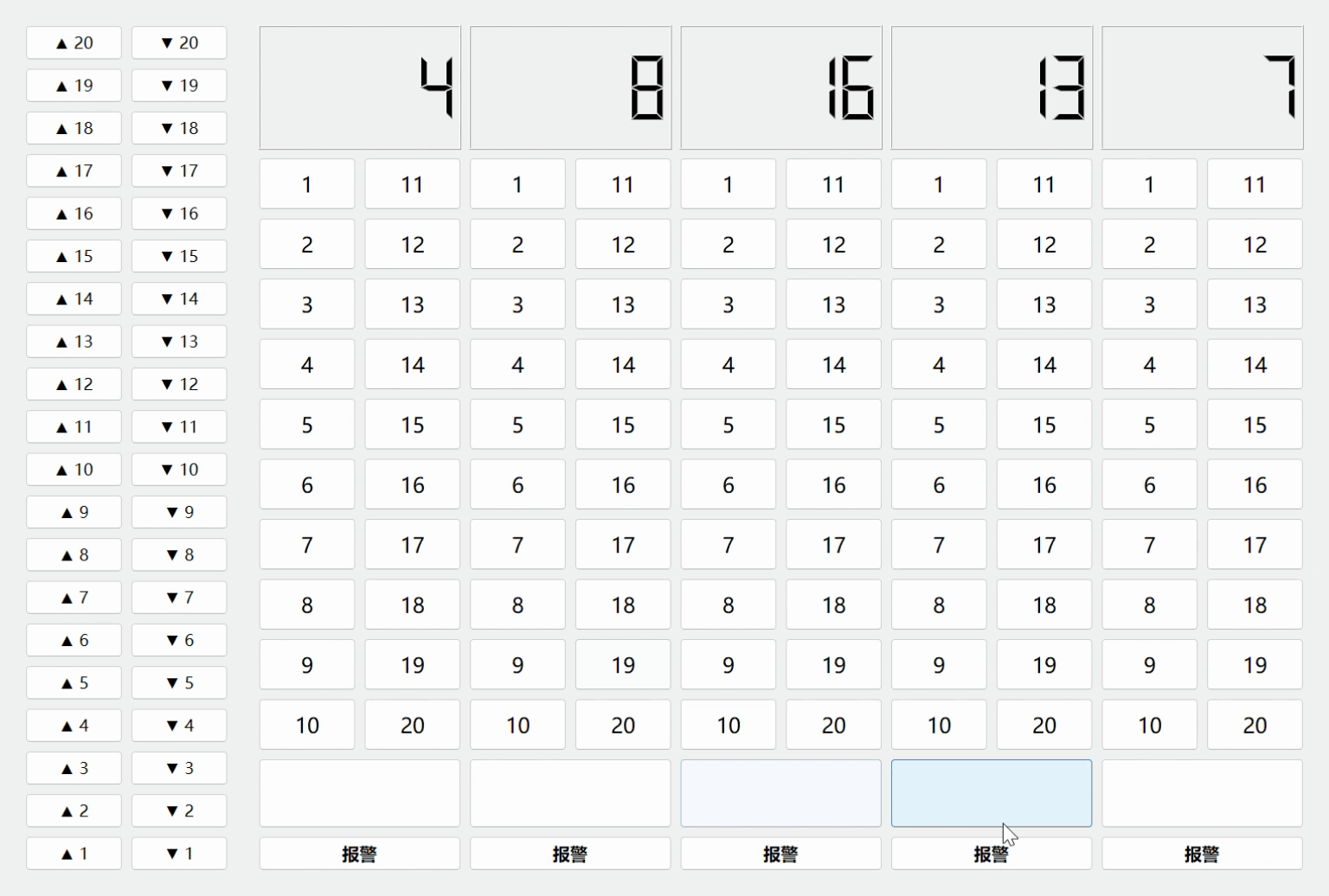
* 如果目标在上方 → 向上
* 如果目标在下方 → 向下
* 无目标 → 静止

|  |  |
| --- | --- |
| 条件 | 意义 |
| state[i] == 0 | 电梯空闲，优先调度 |
| state[i] == direction | 电梯运行方向与请求方向一致 |
| floor[i] <= flr（向上） | 电梯尚未经过请求楼层（顺路） |
| floor[i] >= flr（向下） | 同上 |
| min(candidates) | 从候选电梯中选择“最靠近”的那个 |

**附录：部分情况展示**

**1. 正常状态**

****

**2．部分电梯警报状态**