数据结构与算法课程设计报告

学　　号 19071125
姓　　名 卢雨轩
指导教师 杜永萍

2021 年 12 月 6 日

目录

1 需求分析

1.1 题目描述

设飞机场有四条跑道，四条都可以用于起飞，其中三条用于正常着陆，第四条用于紧急着陆。要求为飞机安排对应的跑道在规定的时间起飞或降落。当飞机出现时，则根据飞机航班号，燃油储备量等，将飞机排入队列。

在机场发生的事件为：

1. 每单位时间最多有 4 架飞机进入起飞队列，最多有 4 架飞机进入着陆队列；
2. 每条跑道在一个单位时间内只允许一次起飞或降落；
3. 在每个单位时间中，任何一个着陆队列里机载燃油接近最低储备量的飞机必需给与高于其他飞机的优先级，进行降落。如果仅有一架飞机出现这种状况，则使用第 4 条跑道，如果多于 1 架（最多 4 架）飞机出现此状况，则也要使用其他跑道。条件是保证安全，不能在跑道上发生撞机，不能因燃油耗尽发生坠机，充分利用跑道资源。

请设计程序系统模拟为各航班飞机安排跑道进行起飞或降落的管理模式。

1.2 需求分析

- 维护起飞、降落队列
- 维护紧急降落队列
- 每单位时间从队列中取出飞机，放入跑道
- 可视化飞机起飞、队列、航班时刻表

1.3 需求实现

1.3.1 使用类似『多级反馈队列』算法的方式调度飞机。

注意到：把飞机调度到跑到上，和操作系统中把进程调度到 CPU 的核心上，某种意义上是类似的。因此，我们可以使用类似『多级反馈队列』算法的方式来调度飞机。

令 W_i 为飞机 i 等待的时间，$fuel_i$ 为飞机 i 的剩余燃油量，N_i 为飞机 i 的调度优先级（nice 值），则我们可以给每个飞机计算一个当前优先级：

$$P_i = \frac{W_i}{F(i)} - 2 \times N_i$$

其中，$F(i)$ 为飞机 i 的燃油指数：

$$F(i) = \begin{cases} 1 & \text{飞机要起飞} \\ \frac{\exp(\text{FuelFactor} \times (F_i - 20))}{\exp(\text{FuelFactor} \times (F_i - 20)) + 1} & \text{飞机要降落} \end{cases}$$

下面，我们逐个分析以上的实现如何实现了每个需求：

- 维护起飞、降落队列：

  将起飞和队列的飞机放入优先队列中，通过以上的方式计算权重。

- 维护紧急降落队列

  由上述计算燃油指数的方式，我们可以得到：

  - 当飞机要起飞时，燃油指数为 1，单纯通过飞机的等待时间和调度优先级进行排序。
  - 当飞机要降落且剩余燃油量大于 20 时，燃油指数是一个接近 1 的数，同样单纯通过飞机的等待时间和调度优先级调度。
  - 当飞机要降落且剩余燃油量小于 20 时，燃油指数是一个接近 0 的数，飞机的优先级趋近于正无穷，一定在优先队列的前端。

    因此，通过以上方式就能保证紧急降落的飞机一定先出队，不需要单独维护紧急降落队列。

- 每单位时间从队列中取出飞机，放入跑道

  只需要每单位时间从队列中取出飞机，放入跑道。其中筛选出紧急着陆或起飞的飞机放入第四条跑道。

- 可视化飞机起飞、队列、航班时刻表

  读取队列内信息渲染即可。

可以看到，上述实现满足题目描述中的需求。

1.4 开发环境

考虑到需要要可视化的信息较多，且渲染较为复杂，课设使用 `TypeScript` 语言开发，并在浏览器内运行。

1.4.1 界面渲染

渲染图形界面较为复杂，很难抽象为 `HTML`，因此项目使用 `PIXI.js` 图形开发框架，通过 `WebGL` 或 `Canvas API` 绘制到 `canvas` 中。这样实现尽可能的兼容了更多平台，还可以使用 GPU 加速渲染界面。

1.5 界面设计

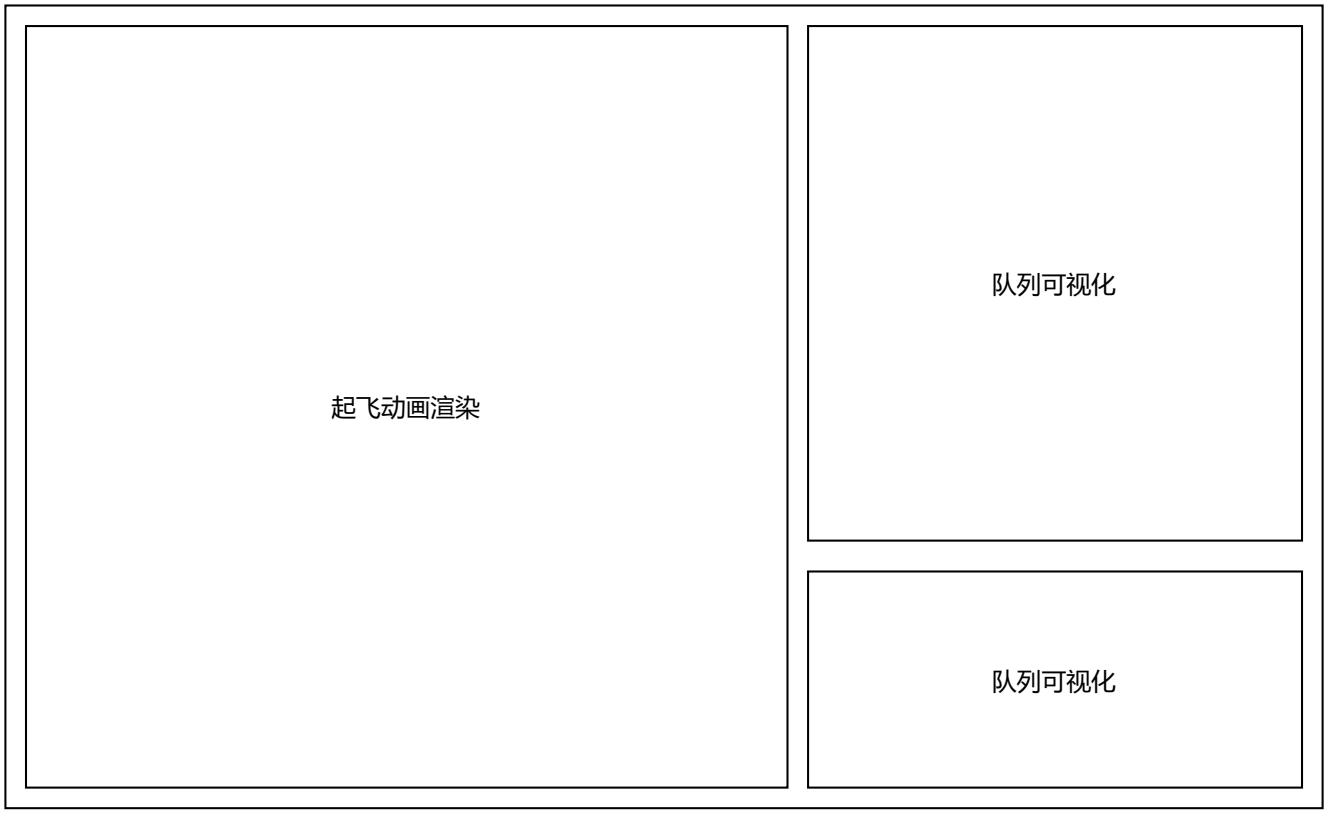

图 1: 机场管理系统界面设计

2 数据结构设计

2.1 队列实现

使用斐波那契堆（Fibonacci heap）维护起飞、降落队列。数据结构设计如下：

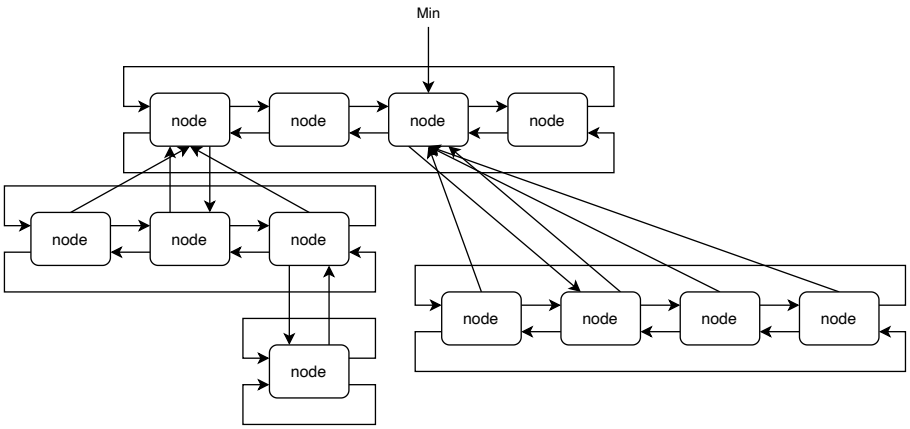

图 2: 斐波那契堆结构

2.2 其他数据结构

下面给出其他数据结构的定义。

2.2.1 飞机

```
class Plane {
    name: string;
    type: 'takeoff' | 'landing';
    time: moment.Moment;
    fuel: number; // will be ignored during takeoff.
    nice: number;
    waitTime: number;
    heapNode: FibHeapNode;
    status: 'waiting' | 'queueing' | 'served' | 'crashed';
    sprite: PIXI.Container;

    constructor(name: string,
        type: 'takeoff' | 'landing',
        time: moment.Moment,
        fuel: number,
        nice: number)

    getSprite(): PIXI.Container

    priority(): number

    toString(): string
}
```

2.2.2 机场

```
class Airport {
    planes: Array<Plane>;
    queue: FibHeap;
    takeoffLoad: number;
    landingLoad: number;
    currentTime: moment.Moment;
    planesOutQueue: Array<Plane>;
    servedCount: number;
    crashedPlanes: Array<Plane>;

    constructor()
```

```
    addPlane(options: addPlaneOption)

    interval(debug: boolean = false)
}
```

2.3 模块功能描述

```
├── assets                资源文件
│   ├── airport.png
│   ├── home.png
│   ├── load-hover.png
│   ├── load.png
│   ├── novem__.woff
│   └── plane.png
├── import-png.d.ts
├── index.ts              主程序代码
├── lib
│   ├── airport.ts        机场数据结构和调度算法
│   ├── config.ts         参数设置
│   ├── fibonacci.ts      fibonacci heap
│   ├── plane.ts          飞机数据结构和优先级计算
│   └── tree.ts           fibonacci heap visualize
├── stages                界面渲染相关
│   ├── loadStage.ts      加载界面
│   └── main.ts           主程序界面
├── style
│   └── main.scss         代码格式控制
├── test.ts
└── utils
    └── time.ts           虚拟时间戳
```

3 详细设计

3.1 优先队列实现

由于项目设计把两类飞机放入同一个队列中，并有大量的更改权重操作，如果使用朴素的二叉堆，每个时间周期的操作的时间复杂度会为 $O(n\log(n))$。因此，在有大量更新权重且只有少量 `push`、`pop` 操作时，应该采用 `fibonacci heap`。

3.1.1 `fibonacci heap` 简介

`fibonacci heap` 是一种可合并堆，可以支持高效的 `decrease key` 操作。与二叉堆的对比详见表 3.1.1 `fibonacci heap` 与二叉堆对比。

操作	fibonacci heap	二叉堆
MAKE-HEAP	$O(1)$	$O(1)$
INSERT	$O(\lg n)$	$O(\lg n)$
MINIMUM	$O(1)$	$O(1)$
EXTRACT-MIN	$O(\lg n)$	$O(\lg n)$
UNION	$O(1)$	$O(n)$
DECREASE-KEY	$O(1)$	$O(\lg n)$
DELETE	$O(\lg n)$	$O(\lg n)$

表 3.1.1: fibonacci heap 与二叉堆对比

3.1.2 fibonacci heap 结构

一个 fibonacci heap 是一个具有最小堆性质的有根树的集合（森林），也就是说，每个结点的关键字均大于或等于他父结点的关键字。我们维护每个节点的父亲、孩子（双向循环链表）、度数、标记（某结点自从上一次成为另一个结点的孩子后，是否失去孩子）。同时，我们维护整个堆的最小结点（也就是森林中的树的根节点中最小的）。如果 fibonacci heap 是空的，那么 min 为 NULL。

3.1.3 势函数

我们将要使用势方法分析 fibonacci heap 的时间复杂度。对于一个给定的 fibonacci heap H，用 $t(H)$ 表示 H 根链表中节点的数目，用 $m(H)$ 中表示 h 中已标记的节点的数目，然后定义 fibonacci heap H 的势函数如下：

$$\Phi(H) = t(H) + 2m(H)$$

3.1.4 最大度数

可以证明,在一个有 n 个结点的 fibonacci heap 中,任何结点的最大度数的上界 $D(n) = \lg_{(1+\sqrt{5})/2}(n)$

3.1.5 fibonacci heap 的操作

MAKE-HEAP

Algorithm 1 MAKE-HEAP

Output: fibonacci heap H

1: $H.min \leftarrow NIL$

2: $H.n \leftarrow 0$

不难看出，时间复杂度为 $O(1)$。

Algorithm 2 INSERT-NODE

Input: `fibonacci heap` H, `fibonacci heap node` x

1: $x.degree \leftarrow 0$
2: $x.p \leftarrow NIL$
3: $x.child \leftarrow NIL$
4: $x.marl \leftarrow False$
5: **if** $H.min = NIL$ **then**
6: Create a root list of H containing just x.
7: $H.min \leftarrow x$
8: **else**
9: insert x into H's root list.
10: **if** $x.key < H.min.key$ **then**
11: $H.min \leftarrow x$
12: **end if**
13: **end if**
14: $H.n \leftarrow H.n + 1$

INSERT-NODE

在插入结点的时候，我们直接将结点插入到 H 的 root list 中。操作的时间复杂度显然是 $O(1)$，势函数的变化量为

$$\begin{aligned}\Delta\Phi &= \Phi(H') - \Phi(H) \\ &= t(H) + 1 + 2m(H) - (t(H) + 2m(H)) \\ &= 1\end{aligned}$$

所以，实际平均代价为 $O(1) + 1 = O(1)$。

MINIMUM

显然，获取最小元素的时间复杂度为 $O(1)$。

UNION

合并两个堆，只需要将他们的 root list 合并，维护 min，并更新堆大小 n。势函数变化量为

$$\begin{aligned}\Delta\Phi &= \Phi(H) - \Phi(H_1) - \Phi(H_2) \\ &= t(H) + 2m(H) - (t(H_1) + 2m(H_1)) - (t(H_2) + 2m(H_2)) \\ &= 0\end{aligned}$$

显然，时间复杂度为 $O(1)$

EXTRACT-MIN

Algorithm 3 EXTRACT-MIN

Input: `fibonacci heap` H

```
1: z ← H.min
2: if z ≠ NIL then
3:     for child x of z do
4:         insert x into H's root list
5:         x.p ← NIL
6:     end for
7:     remove z from H's root list
8:     if H's root list is empty then
9:         H.min ← NIL                                   ▷ now H is empty
10:    else
11:        H.min ← next child of Z's root list           ▷ For now we set a random node as min
12:        CONSOLIDATE(H)                                ▷ Consolidate nodes if needed
13:    end if
14: end if
```

在移除最小结点的时候，我们首先将他的所有孩子插入到 root list 中，然后调用 CONSOLIDATE 合并具有相同度数的结点，保证每个度数至多只有 1 个结点在 root list 中。

下面计算 EXTRACT-MIN 操作的时间复杂度。首先，单次运行 EXTRACT-MIN 的时间复杂度显然为 $O(D(n))$。CONSOLIDATE 的第 19 – 31 行的工作需要的时间代价为 $O(D(n))$。第 2 – 17 行的时间复杂度为根链表的元素数量，也就是 $O(t(H)+D(H)-1)$，因此运行 EXTRACT-MIN 的时间复杂度为 $O(D(n)+t(H))$。势函数变化量为：

$$\begin{aligned}\Delta\Phi &= ((D(n)+1)+2m(H))-(t(H)+2m(H)) \\ &= D(n)+1-t(H)\end{aligned}$$

因此，总的时间复杂度为 $O(D(n)+t(H))+\Delta\Phi = O(D(n))$。后面可以证明，$D(n) = O(\lg n)$，因此 EXTRACT-MIN 的摊还时间复杂度为 $O(\lg n)$。

DECREASE-KEY

首先我们把 x 的 key 减小。接着，如果 x 是根节点或者新的 key 没有大于 x 的父亲的 key，则我们不需要做任何操作。否则，我们需要做很多操作。

首先，我们切断 x。CUT 过程将 x 从 y 的孩子中取出，使其成为根节点。

接着，我们通过 mark 来维护我们的时间复杂度。如果某个结点在成为别的孩子后已经失去一个孩子，那么此时它的 mark 应为 TRUE，我们将它继续切断。我们一直沿着树递归向上进行 CASCADING-CUT，直到某个结点的 mark 为 FALSE 或遇到根节点。

下面我们进行复杂度分析。首先分析它的实际代价。DECREASE-KEY 过程需要 $O(1)$ 的时间，还需要加上调用 CASCADING-CUT 的次数，我们设它为 c。这样，总的实际运行时间就是 $O(c)$。

Algorithm 4 CONSOLIDATE

Input: `fibonacci heap` H

```
1: Let A be an empty array with length D(H.n).
2: for each child w of H's root list do
3:     x ← w
4:     d ← x.degree
5:     while A[d] ≠ NIL do                          ▷ Consolidate nodes with same degree
6:         y ← A[d]
7:         if x.key > y.key then
8:             swap x and y
9:         end if
10:        remove y from H's root list
11:        make y a child of x, incrementing x's degree
12:        y.mark ← False                           ▷ y becomes other node's child, clear it's mark
13:        A[d] ← NIL                               ▷ A[d] is removed from root list
14:        d ← d + 1
15:    end while
16:    A[d] ← x                                     ▷ now, there arn't node with degree same with x.
17: end for
18: H.min ← NIL
19: for i ← 0 to D(H.n) do
20:    if A[i] ≠ NIL then
21:        if H.min = NIL then
22:            Create a root list of H containing just A[i].
23:            H.min ← A[i]
24:        else
25:            insert A[i] into H's root list.
26:            if A[i].key < H.min.key then
27:                H.min ← A[i]
28:            end if
29:        end if
30:    end if
31: end for
```

Algorithm 5 DECREASE-KEY

Input: `fibonacci heap` H, `fibonacci heap node` x, new key k

1: **if** $k \geq x.key$ **then**
2: **error** ▷ We can only decrease key
3: **end if**
4: $x.key \leftarrow k$
5: $y \leftarrow x.p$
6: **if** $y \neq NIL$ and $x.key < y.key$ **then**
7: CUT(H, x, y)
8: CASCADING-CUT(H, y)
9: **end if**
10: **if** $x.key < H.min.key$ **then**
11: $H.min \leftarrow x$
12: **end if**

Algorithm 6 CUT

Input: `fibonacci heap` H, `fibonacci heap node` x, y

1: remove x from y's child list, decrementing y's degree.
2: add x to the root list of H
3: $x.p \leftarrow NIL$
4: $x.mark \leftarrow FALSE$

Algorithm 7 CASCADING-CUT

Input: `fibonacci heap` H, `fibonacci heap node` y

1: $z \leftarrow y.p$
2: **if** $z \neq NIL$ **then**
3: **if** $y.mark = FALSE$ **then**
4: $y.mark \leftarrow TRUE$
5: **else**
6: CUT(H, y, z)
7: CASCADING-CUT(H, z)
8: **end if**
9: **end if**

接下来分析势函数的变化。

$$\begin{aligned}\Delta\Phi &= ((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H)) \\ &= 4-c\end{aligned}$$

因此摊还时间复杂度为

$$O(c)+4-c=O(1)$$

总结

可以看到，`fibonacci heap` 可以在 $O(1)$ 时间内进行 DECREASE-KEY 操作，十分适合本项目这样，有大量 DECREASE-KEY 但是有较少的 POP 和 PUSH 操作的应用情景。

3.1.6 最大度数的界

上述复杂度分析过程中，我们一直假设 $D(n)=O(\lg n)$。下面我们给出证明。

引理 3.1. *设 x 是 `fibonacci heap` 中的任意结点，并假设 $x.degree=k$。设 $y_1,y_2,\ldots,y_n$ 为 x 的孩子，并按照先后顺序排列，则 $y_1.degree\geq 0$，且当 $i\geq 2$ 时 $y_i.degree\geq i-2$。*

证明. 显然，$y_1.degree\geq 0$。对于 $i\geq 2$，当 y_i 进入 x 时，一定有 $x.degree\geq i-1$。此时，一定也有 $y.degree\geq i-1$（因为当且仅当 $y.degree=x.degree$ 时，会执行 CONSOLIDATE 操作，将 y_i 放到 x 的子树中）。

在这之后，由 CUT 操作中的 mark 属性保证了 y_i 不会连续是去两个孩子（否则就会被 CASCADING-CUT，进入 root list）。

综上，$y.degree\geq i-2$。 □

引理 3.2. *设 x 是 `fibonacci heap` 中的任意结点，并设 $k=x.degree$，则有 $size(x)\geq F_{k+2}\geq\phi^k$，其中 $\phi=(1+\sqrt{5})/2$，F_i 为 fibonacci 数列的第 i 项。*

证明. 设 s_k 为 `fibonacci heap` 中度数为 k 的结点的最小可能 size。显然，$s_0=1$，$s_1=2$，$size(x)\geq s_k$，且 s_k 随着 k 单调递增。

设 $y_1,y_2,\ldots,y_n$ 为 x 的孩子，并按照先后顺序排列。则有：

$$size(x)\geq s_k\geq 2+\sum_{i=2}^{k}s_{y_i}.degree\geq 2+\sum_{i=2}^{k}s_{i-2}$$

其中最后一步由引理 3.1 得到。

下面我们使用数学归纳法。首先，我们有 $s_0=1$，$s_1=2$，因此 $s_i\geq F_{i+2}$ 在 $i\leq 1$ 时成立。

$$\begin{aligned}s_k &\geq 2+\sum_{i=2}^{k}s_{i-2}\geq 2+\sum_{i=2}^{k}F_i \\ &= 1+\sum_{i=0}^{k}F_i \\ &= F_{k+2} \\ &\geq \phi^k\end{aligned}$$

□

定理 3.1. 一个 n 个结点的 `fibonacci heap` 中任意结点的最大度数 $D(n)$ 为 $O(\lg n)$。

证明. 设 x 是 `fibonacci heap` 中的任意结点，并有 $k = x.degree$。依据 引理 3.2 ，有 $n \geq size(x) \geq \phi^k$。取以 ϕ 为底的对数，得到 $k \leq \log_\phi n$。

所以，任意结点的最大度数 $D(n)$ 为 $O(\lg n)$。[1] [1]

□

3.1.7 总结

综上所述，`fibonacci heap` 可以在 $O(1)$ 时间内实现 DECREASE-KEY，并在 $O(\lg n)$ 时间内完成 EXTRACT-MIN，十分适合我们的需求。

与传统的二叉堆实现对比，将每次操作的时间复杂度从 $O(n \lg n)$ 降低到 $O(n)$。

3.2 机场调度实现

每单位时间进行如下操作（见 算法 8 机场调度实现）。首先我们更新整个机场的起飞、降落的负载，然后更新队列中等待的飞机的负载情况并更新其优先级。接着，我们从队列中取出 3 个飞机。如果 3 个飞机中有起飞或者紧急降落的飞机，我们则可以把他安排到第四个跑道，然后任意取出一个飞机放入剩余的跑到。如果不存在起飞或紧急降落的飞机，我们则需要寻找一个飞机放入第四个跑道。对于这种情况，我们遍历整个堆中的所有飞机，判断是否可以放入第四个跑道。

3.3 渲染实现

为了实现流畅的渲染和计算，本项目分离逻辑帧与渲染帧，并提供调整时钟速率功能。在每个时间周期，首先进行一个或多个逻辑帧（与时钟速率有关），完成飞机状态、位置的运算，接下来进行一个渲染帧，完成跑道、飞机起飞动画、航班时刻表和 `fibonacci heap` 的可视化渲染。

4 测试

项目使用多组数据进行了不同的测试。

4.1 真实航班数据

项目爬取了北京大兴国际机场某天的航班时刻表，并进行了测试。可以发现，项目可以正确运行、安排跑道、获取飞机的航空公司并渲染对应的尾翼图标（见图 3 真实航班数据运行结果）。

4.2 格式错误数据

项目能正确识别格式错误的数据，并提示用户格式非法（见图 4 非法输入　运行结果）

[1] 上述证明过程参考了算法导论

Algorithm 8 机场调度实现

```
1: L_t ← LOAD_FACTOR × L_t + (1 − LOAD_FACTOR)× count of takeoff planes in queue.
2: L_l ← LOAD_FACTOR × L_l + (1 − LOAD_FACTOR)× count of landing planes in queue.
3: for plane p in queue do
4:     add p's wait time with load
5:     decrease p's fuel if landing
6:     if p.fuel ≤ 0 then
7:         add p to crashed list
8:         remove p from heap
9:     end if
10:    DECREASE-KEY(p, p.priority())
11: end for
12: for plane p to be add into queue do
13:    add p into heap
14: end for
15: E ← FALSE
16: for i ← 0 to 2 do
17:    pop a plane from heap
18:    if this plane is taking off or emgergency then
19:        E ← TRUE
20:    end if
21: end for
22: if not E then                          ▷ we havn't got a plane for lane 4
23:    while not E or heap not empty do
24:        pop a plane from heap
25:        if this plane is takeoff or emergency then
26:            break
27:        end if
28:    end while
29:    push pop-ed plane into heap
30: else
31:    pop a plane from heap               ▷ We have a plane for lane 4 now, so any plane will do.
32: end if
33: record served plane, update served count
```

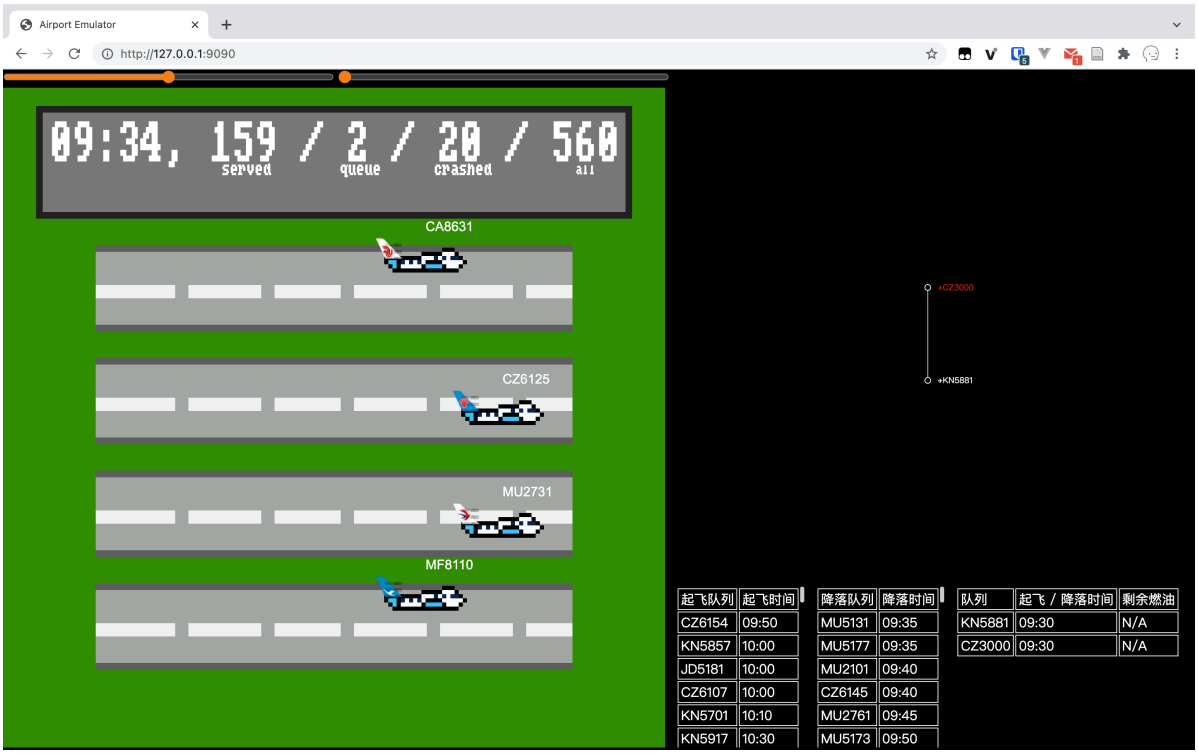

图 3: 真实航班数据运行结果

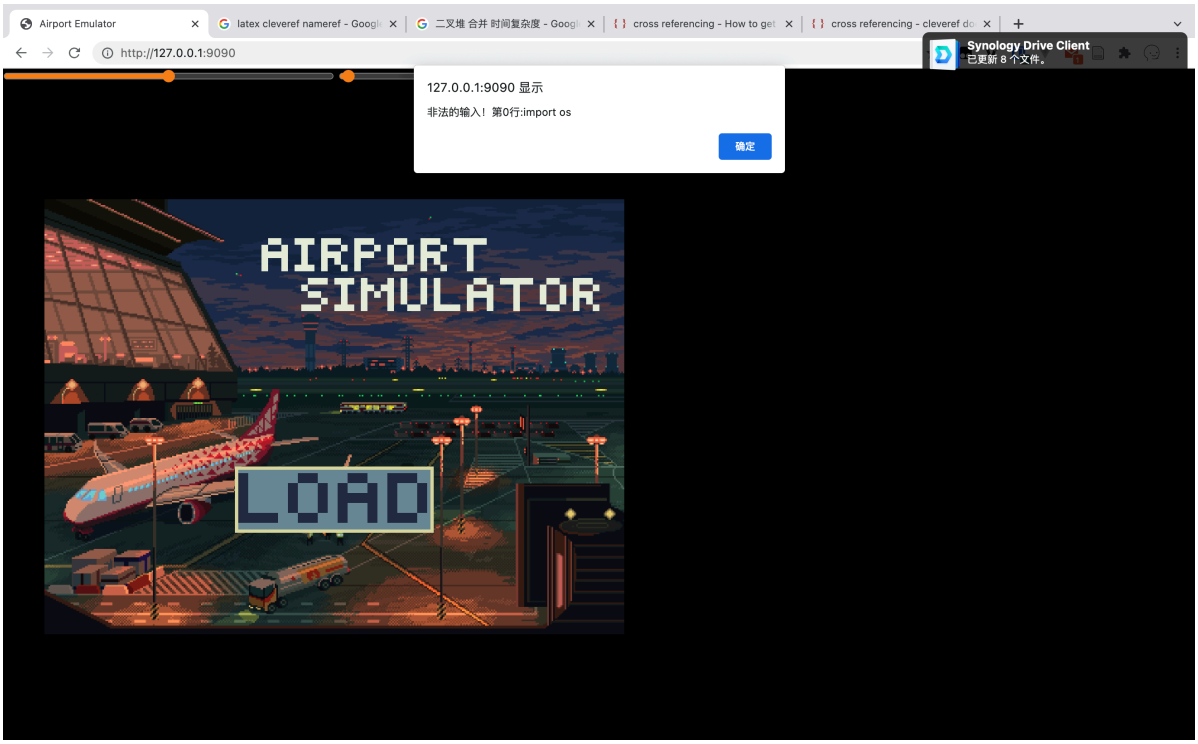

图 4: 非法输入　运行结果

4.3 极端情况数据

项目可以正确处理飞机数量极多、不得不坠机的数据。对此，我们构造了一组必定会有飞机坠毁的数据。可以看到，最开始飞机没有使用紧急降落跑道，燃油量也可以正确计算（图 5 正常降落、记录燃油　运行结果）。稍后，等到有飞机的燃油量小于 10 时，第四条跑道被占用（图 6 使用紧急降落跑道　运行结果）。最后，当飞机的燃油量小于 0 时，飞机被从队列中移除，并增加了坠毁计数器（图 7 坠毁计数　运行结果）。

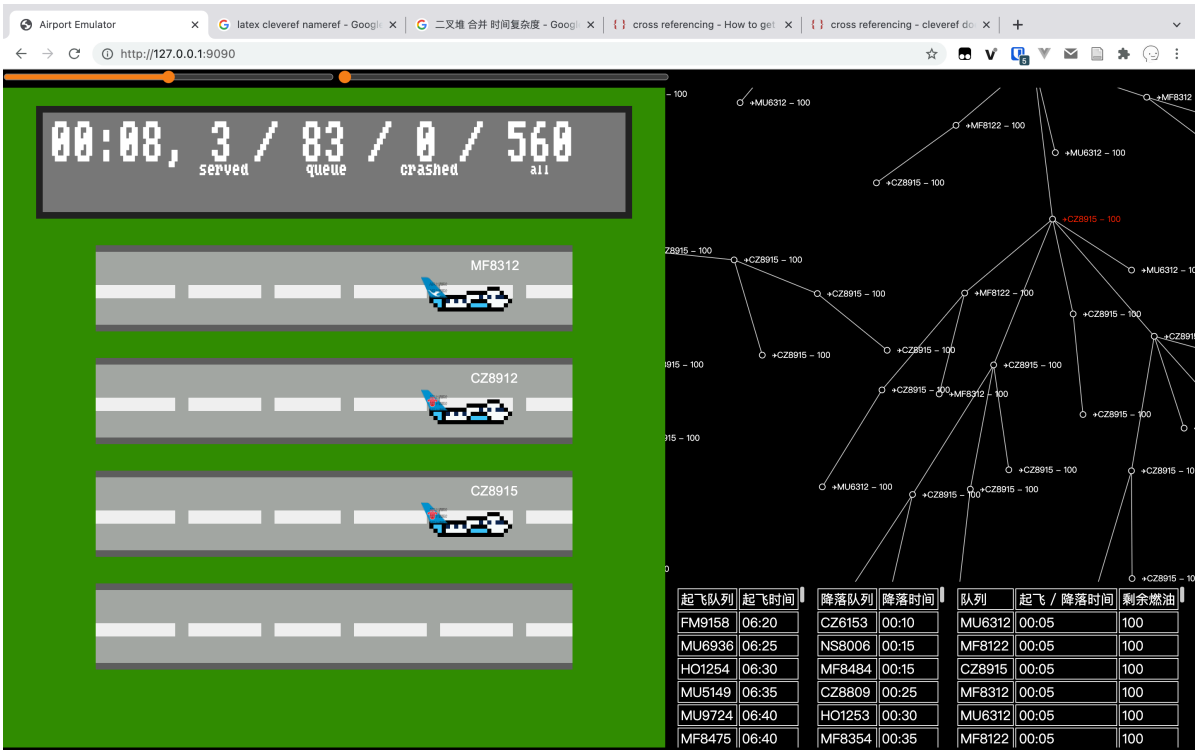

图 5: 正常降落、记录燃油　运行结果

5 总结与提高

本项目具有以下特色：

- 创新性的使用『多级反馈队列』算法调度飞机
- 实现并应用了 `fibonacci heap` ，在不提升时间复杂度的情况下实现动态优先级的计算与更新
- 实现了如下拓展功能：
  - 飞机起飞、降落动画的渲染
  - 飞机尾翼图标的动态渲染
  - 逻辑帧与渲染帧分离的时钟速率系统
  - `fibonacci heap` 的可视化、力反馈图展示

在实现项目的过程中，我也收获了许多。

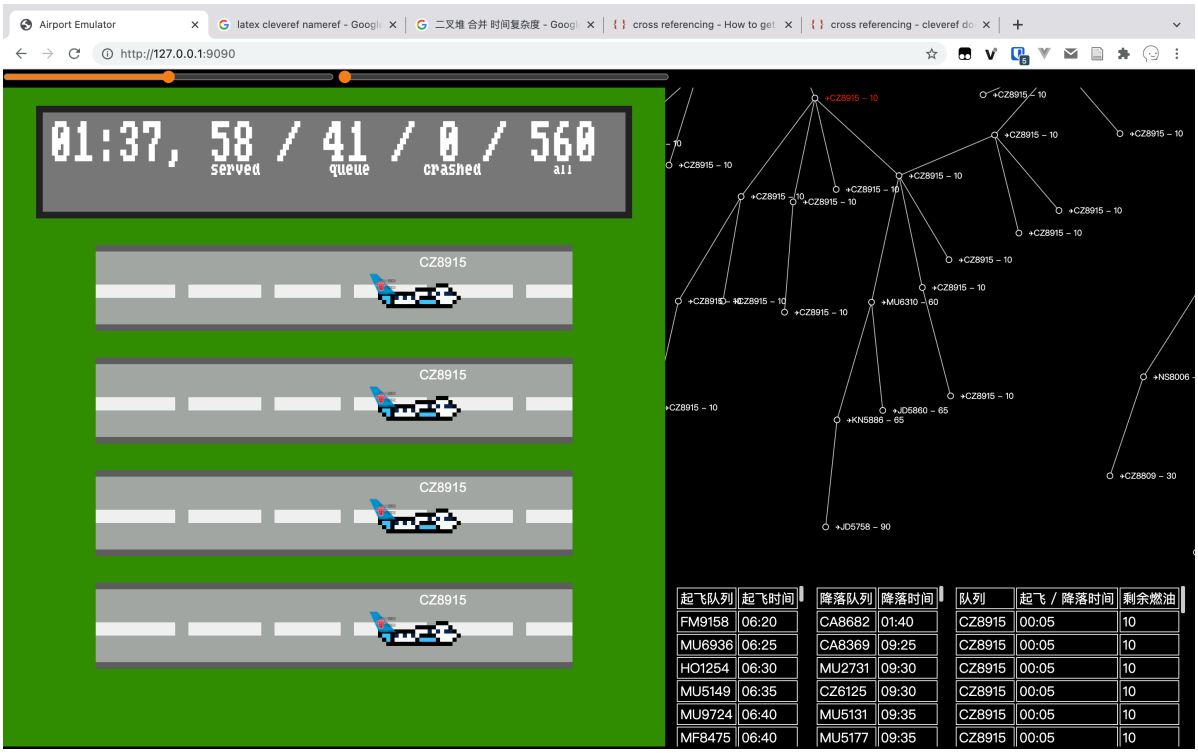

图 6: 使用紧急降落跑道　运行结果

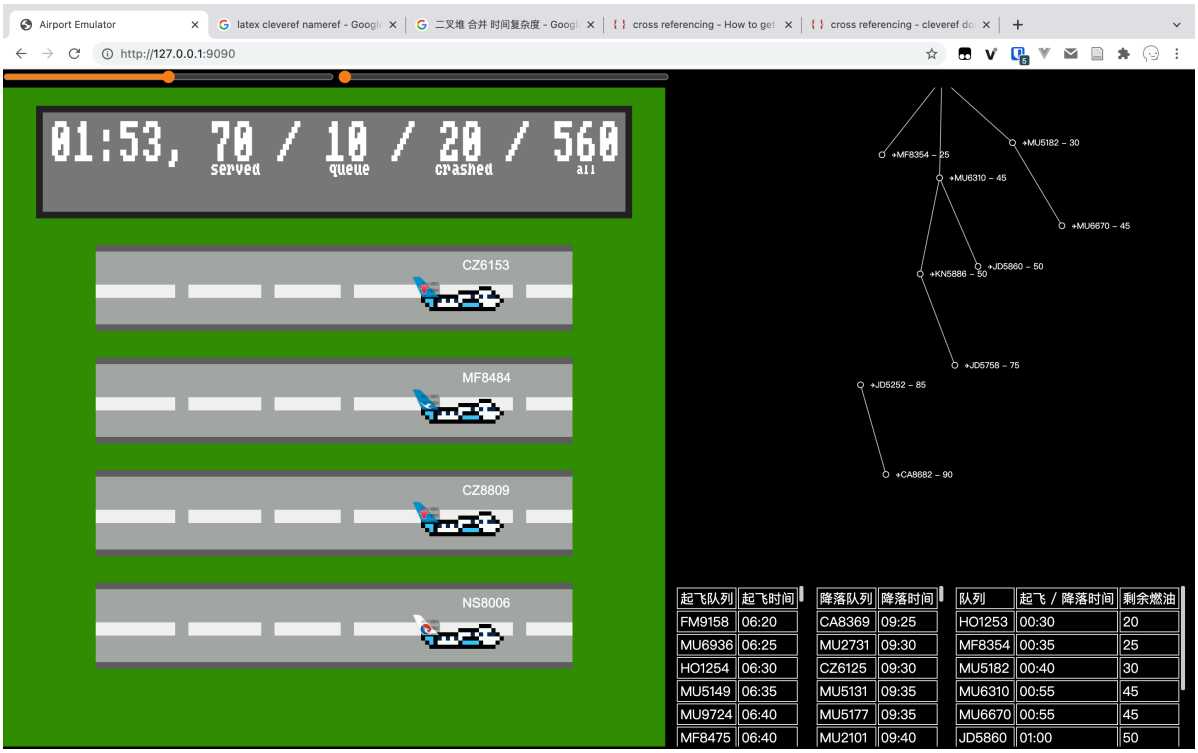

图 7: 坠毁计数　运行结果

1. 在理解并实现 `fibonacci heap` 的过程中，我加深了对于使用势函数分析算法的均摊复杂度的方法，并理解了 `fibonacci heap` 的势函数的设计的原因。

   在之前分析简单算法的时间复杂度时，某一步的操作与下一步操作的时间没有关系，只需要分别分析每一步子问题的数量和子问题消耗的时间。但是，对于 `fibonacci heap` 这样的复杂算法，每一步操作的时间都会影响下一步操作的时间。如：EXTRACT-MIN 时，我们首先将最小值的孩子放入根链表中，增加了整体的势，使得之后的操作变慢。接着，我们合并了根链表中度数相同的节点，使得其势显著减小。这样，整体来看，一次 EXTRACT-MIN 的时间复杂度仍为 $O(\lg n)$。

2. 在设计时钟速率功能的过程中，最开始我没有分离逻辑帧和渲染帧。这就导致了速率过快的时候，渲染速率较慢，成为系统最大的瓶颈。

   在调查了其他项目的实现后，我决定分离逻辑帧和渲染帧，也就是在一个渲染帧中执行多个逻辑帧。这样，系统的渲染速度就不再成为系统的瓶颈，时钟速率也可以设置更高的最大值。

3. 在不同的应用场景下，对于同一个数据结构要采用不同的实现。如，在本项目中，由于有大量的 DECREASE-KEY 操作，选用 `fibonacci heap` 可以加快整个系统的运行速度和时间复杂度。类似的，对于求图的最短路的 dijkstra 算法而言，每次松弛操作也有大量的 DECREASE-KEY 操作，如果选用 `fibonacci heap` 就可以把时间复杂度从 $O(n \lg m)$ 降低到 $O(n \lg n)$。

参考文献

[1] MAN K, PING Y J. Suan fa dao lun[M]. [S.l.]: Ji xie gong ye chu ban she, 2013.