Project Report: Gator Air Traffic Slot Scheduler

Saaketh Balachendil
UFID: 86294284
s.balachendil@ufl.edu
COP 5536 Fall 2025

1 Program Structure

This program simulates a non-preemptive, priority-based air traffic slot scheduler. At its core is the `GatorAirTrafficScheduler` class, which manages all operations and data.

The program is structured in four main parts:

1. **Main Execution (main function):** This is the entry point. It parses the command-line argument to get the input filename, generates the output file name, and reads the input line by line. Each command is parsed and routed to the appropriate method in the scheduler class.
2. **Scheduler Class (GatorAirTrafficScheduler):** The heart of the program. It holds all data structures and implements the 10 required operations (like `SubmitFlight`, `Tick`, `PrintActive`). The two most important internal methods are `_advanceTime` and `_rescheduleUnsatisfied`, which together implement the "two-phase update" logic: Phase 1 settles completed flights, and Phase 2 reschedules any unsatisfied ones before each operation.
3. **Custom Data Structures:** As required, I implemented the `MinHeap` (used for runway pool and timetable) and `MaxPairingHeap` (for pending flights) from scratch. These are self-contained and support all needed operations like `push`, `pop`, `updateKey`, and `erase`.
4. **Helper/Model Classes:** Several small classes store data:
   - `Flight`: Holds all info for a single flight: `flightID`, priority, state, assigned times, etc. [cite: 19-24].
   - `Runway`: Simple wrapper for runway items in the `runwayPool` heap; manages `nextFreeTime` and `runwayID` for comparison.
   - `HeapItem`: Generic wrapper for `MinHeap` to support tuple-based keys.
   - `PairingNode`: Node class used internally by `MaxPairingHeap`.

2 Data Structure Implementation

The six required data structures are implemented as follows:

1. **Pending Flights Queue (Max Pairing Heap):**
   - *Implementation:* Custom `MaxPairingHeap` class. [citestart]
   - *Purpose:* Stores flights in `PENDING` state, ordered by `(priority, -submitTime, -flightID)` so the highest-priority flight is always popped first [cite: 78-81].
2. **Runway Pool (Binary Min Heap):**
   - *Implementation:* Custom `MinHeap` class.
   - *Purpose:* Holds available runways, ordered by `(nextFreeTime, runwayID)` so the earliest available runway is always selected.
3. **Active Flights (Hash Table):**
   - *Implementation:* Standard Python `dict`.
   - *Purpose:* Master record of all flights in the system (Pending, Scheduled, InProgress). Maps `flightID` to `Flight` object for O(1) lookup.
4. **Timetable/Completions (Binary Min Heap):**
   - *Implementation:* Custom `MinHeap` class.
   - *Purpose:* Stores `SCHEDULED` and `INPROGRESS` flights, ordered by `(ETA, flightID)`. Lets `_advanceTime` efficiently find and process completed flights in correct order.
5. **Airline Index (Hash Table):**
   - *Implementation:* Python `defaultdict(set)`.
   - *Purpose:* Maps `airlineID` to a set of `flightID`s. Used by `GroundHold` to quickly find all flights in a given airline range [cite: 111-113].
6. **Handles (Hash Table):**
   - *Implementation:* Standard Python `dict`.
   - *Purpose:* Maps `flightID` to its `PairingNode` in the pending heap. Enables O(log n) `updateKey` (for `Reprioritize`) and `erase` (for `CancelFlight`).

3 Function Prototypes (Method Signatures)

Below are the key method signatures that define the program's interface.

3.1 GatorAirTrafficScheduler (Core Class)

This class contains the main logic and all public operations.

— Public API Operations —

```
def Initialize(self, runwayCount):
    """Starts the system with a given number of runways."""

def SubmitFlight(self, flightID, airlineID, submitTime,
                 priority, duration):
    """Adds a new flight request to the system."""

def CancelFlight(self, flightID, currentTime):
    """Removes a flight that has not yet started."""

def Reprioritize(self, flightID, currentTime, newPriority):
    """Changes a flight's priority before it starts."""

def AddRunways(self, count, currentTime):
    """Adds new runways to the system and reschedules."""

def GroundHold(self, airlineLow, airlineHigh, currentTime):
    """Removes unsatisfied flights in an airline ID range."""

def PrintActive(self):
    """Shows all flights still in the system."""

def PrintSchedule(self, t1, t2):
    """Shows Scheduled-but-not-started flights with ETAs in [t1, t2]."""

def Tick(self, t):
    """Advances the system clock to time t."""

def Quit(self):
    """Terminates the program."""
```

— Internal Core Logic Methods —

```
def _advanceTime(self, newTime):
    """
    Main two-phase engine:
    1. Settles completions (Phase 1)
    2. Promotes flights to InProgress
    3. Runs a pre-operation reschedule (Phase 2)
    """

def _rescheduleUnsatisfied(self):
    """
    Implements the Phase 2 greedy rescheduling logic.
    Rebuilds the schedule for all PENDING and SCHEDULED flights.
    """
```

— Internal Helper Methods —

```
def _write(self, message):
    """Writes a line to the designated output file."""

def _printChangedEtas(self, changedEtas):
    """Formats and prints the 'Updated ETAs' line."""

def _removeFlightFromSystemIndices(self, flightID):
    """Removes a flight from all secondary structures."""
```

3.2 MinHeap (Custom Data Structure)

```
def __init__(self):
    """Initializes an empty list to store heap items."""

def isEmpty(self):
    """Returns True if the heap is empty."""

def peek(self):
    """Returns the top item (minimum) without removing it."""

def push(self, item):
    """Adds a new item and sifts up to maintain heap property."""

def pop(self):
    """Removes and returns the top item (minimum) and sifts down."""

def clear(self):
    """Empties the heap."""

def _siftUp(self, index):
    """Private helper to move an item up the heap."""

def _siftDown(self, index):
    """Private helper to move an item down the heap."""
```

3.3 MaxPairingHeap (Custom Data Structure)

```
def __init__(self):
    """Initializes an empty root."""

def isEmpty(self):
    """Returns True if the heap is empty."""

def peek(self):
    """Returns the value of the root (maximum) without removing it."""

def push(self, key, value):
    """Adds a new key/value pair by merging a new node with the root."""

def pop(self):
    """Removes the root and performs a two-pass merge on its children."""

def updateKey(self, node, newKey):
    """Updates a node's key. Cuts and merges if the key increases."""

def erase(self, node):
    """Removes an arbitrary node from the heap."""

def clear(self):
    """Empties the heap."""

def _merge(self, node1, node2):
    """Private helper to merge two sub-heaps."""
```

```
27
28 def _mergeSiblings(self, firstSibling):
29     """Private helper for the two-pass merge strategy used in pop()."""
30
31 def _cutNode(self, node):
32     """Private helper to detach a node from its parent and siblings."""
```

3.4 Helper Classes

```
1  class Flight:
2      def __init__(self, flightID, airlineID, submitTime,
3                   priority, duration):
4          """Stores all data for a single flight."""
5
6  class Runway:
7      def __init__(self, runwayID, nextFreeTime):
8          """A wrapper for runway items in the MinHeap."""
9
10 class HeapItem:
11     def __init__(self, key, value):
12         """A wrapper for generic items in the MinHeap."""
13
14 class PairingNode:
15     def __init__(self, key, value):
16         """A node for use in the MaxPairingHeap."""
```