How Can Aircraft Fleet Operation be Optimized to Capture Market Demand while Meeting Operational Constraints?

Final Exam - Home Assignment

Programming, Algorithms and Data Structures

Somnath Mazumdar

Student: Sebastian Uedingslohmann (175867)

Pages: 15
Character Count: 33,444
Date: December 20, 2024

Abstract

The efficient allocation of aircraft fleets and their crews is challenging for airlines, as they aim to maximize operational efficiency while considering volatile market demand, external factors, and operational constraints. This project develops a simulation-based program to optimize flight schedules by addressing fleet availability, crew working hours, night flight bans, and estimated market demand. The program generates flight schedules that minimize fleet downtime and assign crews while ensuring compliance with operational constraints. Every aircraft returns to its base, Frankfurt Airport, before the night flight ban at 23:00. Destinations are sorted according to estimated demand using a merge sort algorithm, and crews are dynamically assigned based on availability and feasibility constraints. The result demonstrates a system that efficiently utilizes fleet and crew to serve the routes with the highest demand first. It identifies profitable destinations and ensures that no crew works longer than 10 hours. If crew shortages occur, the aircraft remains grounded for the rest of the day.

Keywords: Flight Schedule Optimization, Fleet Assignment Problem, Resource Management, Merge Sort

GitHub: `https://github.com/seue24ab/Programming-Algorithms-and-Data-Structures-Assignment`

Contents

1 Introduction

Airlines face various challenges in creating flight plans such as the balance of market demand, the limited number of aircraft, and the availability of crew (Sherali et al., 2006). This includes managing the fleet considering ranges, capacities, profitabilities, crew working hours and operational rules such as turnaround schedules and night flight bans. These factors introduce significant complexity with operational and regulatory constraints. This program simulates decision making under less complexity, considering some variables to study and optimize real-world scenarios, providing insights into how limited resources can be managed efficiently. This project tackles the fleet assignment problem and aims to follow the four processes flight scheduling, fleet assignment, aircraft routing, and crew assignment (Kenan et al., 2018) to create flight plans that maximize profits while dealing with a limited number of aircraft and crew availability as well as the night flight ban at Frankfurt Airport. If resources are not managed correctly, flights may result in a loss of profits or employee-protection laws being violated. The relevance of this project is highlighted through the importance of efficient fleet and crew management which is a cornerstone of airline profitability and customer satisfaction. By optimizing flight schedules, airlines can significantly reduce operational costs and improve resource utilization which are crucial for maintaining competitiveness in the dynamic industry. The objective of this project is to simulate a day of fleet operation and to generate a flight schedule which allocates planes and crew based on demand and availability. The goal is to maximize the profit given the estimated demand on certain routes by using optimization techniques and a sorting algorithm. Furthermore, the schedule must ensure that crew working hours and operational flight constraints are respected. The simulation serves as a foundation for understanding and improving resource management in airline operations.

2 Requirement Analysis

2.1 Functional Requirements

There are a variety of factors a flight plan must consider, such as distances to destinations, capacities of the planes, booking classes, restrictions of the airport, availability of staff, operational restrictions, time slots provided by air traffic control, weather, and the list continues. This project should focus on the key considerations profitability, crew availability, and night flight ban. This results in five core functionalities of the program. **Sorting of destinations based on profitability.** The program should use a merge-sort algorithm to rank destinations based on their estimated demand. Demand is generated randomly for each destination to simulate real-world variations in market demand. The sorted list should prioritize the most profitable destinations, allowing the program to allocate aircraft and crews to routes with the highest demand first. This approach ensures that resources, such as flight hours and crews, are used optimally. Profitability should be calculated based on the estimated demand, the capacity of the aircraft, and a fixed profit-per-passenger value. **Crew management.** Crews must be assigned to each plane, while every crew may not work longer than ten hours. Another implication is that any crew operates only one plane during a working day. If the next flight assigned to the aircraft exceeds the crew's maximum working hours, the crew finishes their shift for the rest of the day. This means the crew cannot be reassigned to operate a shorter flight later, even if it fits within their remaining working time. Additionally, it must be ensured that turnaround times are included in the crew's total working time. As a result, a crew may not operate a flight in the morning and then another flight in the evening. On top of that, the user should be able to request the working hours of each crew member at the end of the working day. **Return of the fleet.** The flight schedule should be designed in a way that the whole fleet is parked at Frankfurt Airport before the night flight ban enters effect at 23:00 in the evening. This ensures that no aircraft violates operational regulations and avoids unnecessary penalties. **Calculation of profits.** The program calculates the expected profits based on the randomly generated demand for each route, the fixed profitability-per-head, and the capacity of the planes. The output provides the calculated profit for each leg and the total expected profit for the day. **Generation of a flight plan.** The final output should provide a chronological listing of each plane, its destination, return time, operating crew, and profit. The flight plan represents a unique and optimal solution designed to maximize aircraft utilization and profits while adhering to all constraints. These constraints include crew working hours, turnaround times, and the night flight ban. By ensuring compliance with these limitations, the program achieves efficient resource management and realistic flight scheduling.

2.2 Non-Functional Requirements

Besides the functional requirements, the program should also fulfill non-functional requirements, including flawless code execution, extensibility, and the ability to maintain consistent and accurate results. Additionally, the code should be clear and well-structured to ensure easy readability and maintainability for future modifications or extensions. For the development of this program, the focus should lay on the following non-functional requirements. **Scalability.** The design of the code should allow for more destinations, aircraft, and crew members to be added without requiring significant modifications. The modular structure of the program should ensure that new elements can be integrated seamlessly, such as additional flight routes with varying demands, larger fleets with different aircraft types, or expanded crews with specific working hour constraints. This flexibility makes the program adaptable to more complex scenarios and larger-scale simulations. **Clear and well-structured code.** This is essential to ensure readability, maintainability, and ease of debugging. By following consistent naming conventions, modular design, and object-oriented programming, the code becomes easier to understand and extend. This allows future developers to quickly grasp its functionality and make modifications or additions without introducing errors. **Usability.** The program should work intuitively and provide an output which is easy to interpret. The results, the flight schedules, crew assignments, and profit calculations, should be presented in a clear and structured format. This ensures that users can quickly understand the information without additional explanations.

2.3 Constraints

The project is being developed with certain limitations to keep it simple and manageable. Only a few factors were considered, such as demand, crew working hours, turnaround times, and night flight bans. The program does not include a long-haul fleet or complex flight networks. Instead, it focuses on a small set of destinations and a fleet of only three aircraft of the type Airbus A320. This reduced scale makes the simulation easier to understand and test. Additionally, simplified assumptions, like random demand values and fixed profitability per passenger, are being used. These choices ensure the program remains focused on its core functionality while meeting the project goals within the given time and resource constraints.

3 System Design

3.1 Architectural Design

The program follows a modular design to ensure clarity. Modular programming is a software design technique that involves the separation of functionalities into independent modules (Busbee & Leroy, 2018). To implement this, the program consists of different classes, each with a specific role. `Planes` class represents the fleet of aircraft with attributes like registration, capacity, range, and availability time. `Destinations` class contains different destinations as instances with attributes like distance from Frankfurt Airport in kilometers, flight time, and estimated demand. `Crew` class manages the crew members and their availability, ensuring working hours are not exceeding. This class is the parent class of the subclasses `Captain`, `FirstOfficer`, and `FlightAttendant`, which inherit from the `Crew` class. `Simulation` class is the central class that coordinates the entire simulation process. It manages the flight scheduling, checks constraints and calls the methods from the other classes. The relationship between these components is shown in the block diagramm below.

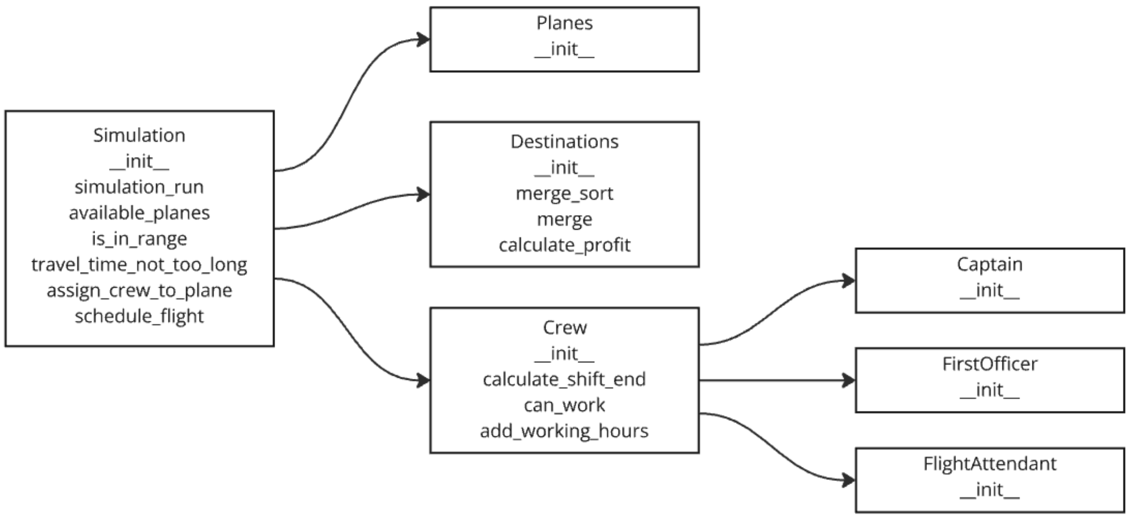


Figure 1: High-level block diagram of the system architecture.

Each class handles a specific responsibility, making the system both structured and efficient. Moreover, the modular structure improves readability, manageability, easier testing, reusability, and maintanance (Ricciardi, 2024).

3.2 Module Design

To delve deeper into the architecture of the code, this section focuses on explaining the individual methods used that make up the outcome. Each method plays a specific role in ensuring that the system works effectively and meets all requirements. By breaking down each method in detail, the design of the program becomes more clear, and the logic behind its processes can be better understood. The following figure illustrates how the methods work together to optimize resources and produce a profit-maximizing flight schedule.

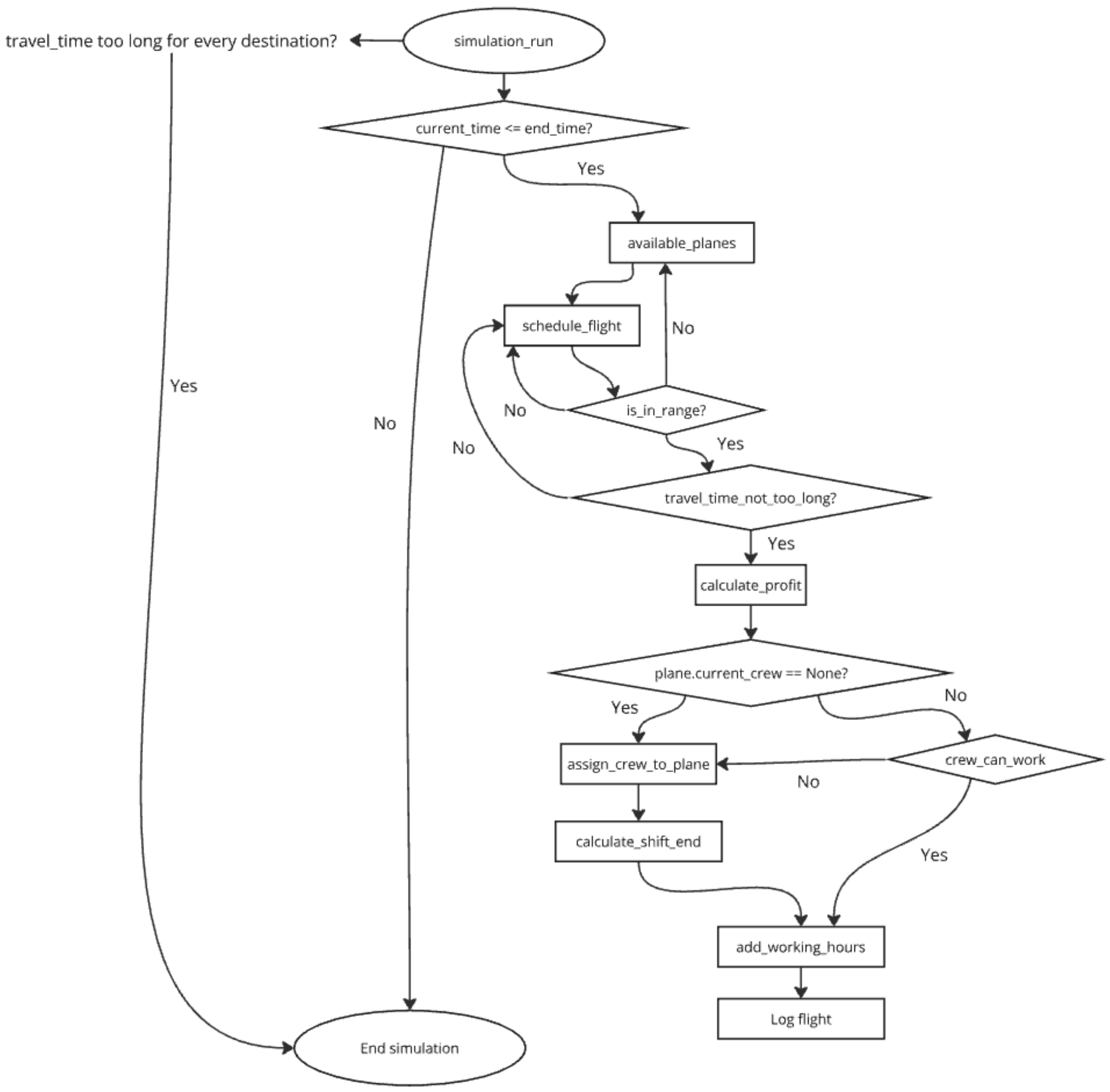


Figure 2: Process flowchart of the program.

The program begins by importing the required modules and libraries, including `random`, `datetime` and `timedelta` from the `datetime` module. Prior to defining the classes, the start time and the end time of the simulation are set. This step is essential because the `Plane` class relies on the `start_time` variable in its initializer.

The class `Planes` is then being defined with the attributes `registration`, `aircraft_type`, `capacity`, `range`, `turnaround_time`, `available_from`, and `current_crew`. The `registration` attribute is used to identify a specific aircraft, while `capacity` represents the number of seats on the plane. The `range` indicates the maximum distance, in kilometers, that the aircraft can fly. `turnaround_time` is the required time for the plane to be dispatched at the airport and is defined as a `timedelta` object, meaning the input gets converted into a time format (HH:MM:SS). Moreover, `available_from` is initialized as the start time of the simulation, ensuring that every instance of the class `Planes` is available at the beginning of the day. Lastly, `current_crew` is set to `None` by default, as no crew is assigned to any plane at the start of the simulation.

Following `Planes` class, the `Destinations` class is initialized with the attributes `iata`, `city`, `distance`, `flight_time`, `estimated_demand`, `profit_per_head`, and `occupancy`. `iata` corresponds to the unique three-letter airport code issued by the International Air Transport Association. `distance` specifies the distance, in kilometers, from Frankfurt Airport to the destination. To facilitate calculations, `flight_time` is initialized as a `timedelta` object and represents the average flight duration to the destination. The attribute `estimated_demand` is initialized as a random integer between 30 and 100. this randomization introduces variability into the demand for routes, ensuring diverse and dynamic simulation outcomes. The `profit_per_head` attribute is fixed at 15 by default, allowing a better comparison of the profitability of each route. Finally, `occupancy` is set to zero as this value is calculated dynamically before every flight.

The `merge_sort` method in the code sorts the destinations based on their estimated demand using the merge sort algorithm. This method works by splitting the list of destinations into smaller parts until each part only contains one element. This is achieved by the line `lower = Destinations.merge_sort(destinations[:mid])` and `upper = Destinations.merge_sort(destinations[mid:])`. After splitting, the `merge` function is called to merge the smaller lists back together while sorting them based on the `estimated_demand` attribute. Inside the `merge` method, two lists (`lower` and `upper`) are compared element by element, and the element with the higher demand is added to the final sorted list. Merge sort is particulary useful for this program because it has a time complexity of O(n log n), making it efficient for sorting a potentially larger list of destinations. Furthermore, merge sort is a stable sorting algorithm which preserves the original order of equal elements and is easy to implement.

The `Crew` class is the main class for managing crew members in the simulation. It initializes the basic information of each crew member, like their ID and their position. It also defines the relevant attributes `total_hours` and `max_working_hours`, which are timedelta objects. This attributes are necessary for the tracking of the working hours and the compliance of crew working hour limitations. Moreover, the class defines the `available_from` attribute, which is required when it comes to determining which staff could operate an upcoming flight. The `assigned_plane`, `shift_start`, and `shift_end` attributes are set to `None` by default, since these attributes will be assigned with the relevant information as soon as a crew member is selected to operate a flight. The classes `Captain`, `FirstOfficer`, and `FlightAttendant` are child classes that inherit from the `Crew` class. each of these subclasses sets a specific position for the crew member by calling the parent class with the `super()` method. For exmaple. the `Captain` class passes the position as 'Captain', while the other two classes use 'FirstOfficer' and 'Flight Attendant'. This structure follows the desired modular structure and makes it easy to manage different roles while reusing the code from the `Crew` class.

Besides the initializer, `Crew` has three more methods. The method `calculate_shift_end` takes the parameters `crew` and `travel_time` and sets the attribute `shift_end` to the latest time a crew member needs to stop working, calculated by `shift_start` plus `max_working_hours`. This means that as soon as a crew member starts operating a flight, the crew member may not work for longer than the time of the first departure plus the maximum working hours. Another method of the class `Crew` is `can_work`, which returns a boolean value. The purpose of this method is to determine if a crew could still operate the next flight without exceeding the maximum working hours allowed. The function takes `current_time` and `flight_time` as attributes and returns `True` if the travel time of the next flight added to the current time does not exceed the value set in the attribute `shift_end`. The last method of the class is `add_working_hours`, which is used to track the working hours of each crew. If a crew operates a flight, the travel time, taken as the parameter `hours` is added to the attribute `total_hours`. This allows to print the total working hours of each crew at the end of the simulation.

The `Simulation` class is used to manage the entire simulation process. The `__init__` method takes `planes`, `destinations`, `start_time`, `end_time`, and `crew_member` as parameters and initializes these attributes. `planes`, `destinations`, and `crew_members` are lists of instances of the corresponding classes, which will be explained at a later point. Moreover, the method initializes the attribute `current_time`, which is set to `start_time` at the beginning. This attribute will be overwritten as the simulation iterates over the day. Another attribute is `total_profit`, which tracks the total expected profit generated by the operated flights during the day. Furthermore, `no_more_flights` is set to `False` by default. This attribute is meant to signalize that no destination can be served anymore, since the plane would not return to Frankfurt Airport in time. Finally, `available_crew` is a list of all the crew instances who have the attribute `total_hours` set to zero. This is relevant because only crew members who were not on duty this day may be selected to operate a flight.

The `simulation_run` method controls the entire simulation process by running a loop throughout the day. The loop starts by printing the simulation start message and iterates as long as `current_time` is less or equal to `end_time`. Within the loop, the method checks for available planes using `self.available_planes()`. If no more flights can be scheduled on the day, the loops breaks early and does not continue to iterate until the `end_time` is reached. If planes are available, meaning dispatched in Frankfurt, the program iterates over each plane in the `available_planes` list. For each available plane, the `schedule_flight` method is called to assign a flight. If all destinations were served on this day, a message is printed, and the simulation ends. No destination is served more than once per day. The `current_time` is increased by one minute after each iteration, simulating the passing of time. After the main simulation loop ends, the program asks the user if they would like to see the working hours of each crew member. It waits for a valid input, either ”yes” or ”no” (case insensitive). If the user enters ”yes”, the attribute `total_hours` is printed for every crew member.

As elaborated previously, the method `simulation_run` calls the method `available_planes`. This method takes no parameters and returns the list `available_planes`, which every instance of `planes` that is available in the moment of the simulation, is being appended to. The availability is determined by comparing the `available_time` attribute of the instance plane to `current_time`. This method is crucial to identify which aircraft are ready to operate to maximize fleet utility.

Both `is_in_range` and `travel_time_too_long` methods are designed to return boolean values to verify if a flight can be operated and does not violate operational constraints. `is_in_range` take two parameters, `plane` and `destination`, to check if the destination is within the range of the plane. `travel_time_not_too_long` checks if the plane can serve a route and return back to Frankfurt before 23:00. The method takes `travel_time` as a parameter. As described later, `travel_time` is the duration of the flight time forth, the turnaround time, and the return flight. If the current time plus the duration of the leg does not exceed 23:00, the method returns `True`.

The `assign_crew_to_plane` method takes the `plane` and the `travel_time` as parameters and assigns a crew to a plane based on the required roles for the operation. It starts by creating an empty list `crew_plane` to hold the assigned crew members. The method then iterates through the `required_roles` list. For each role, it loops through `self.available_crew`, which contains the crew members that are still available on this day. If a crew member’s `position` matches the role, that crew member is assigned to the plane. The crew member’s shift begins at the `current_time`, and the method calculates the `shift_end` by calling the `calculate_shift_end` method. The crew member is removed from `self.available_crew` to ensure that it does not get assigned to a different plane later that day, and their working hours are updated using `add_working_hours`. As soon as all required roles are staffed, the loop stops. If no crew can be assigned because there are not enough available crew members, a `ValueError` is raised, indicating that no crew could be assigned to the plane. Finally, the `plane.current_crew` is updated with the assigned crew, and the method returns `True`, confirming that the crew was successfully assigned to the plane. This ensures that each plane has the correct crew constellation to operate.

The `schedule_flight` method is responsible for assigning flights to planes while checking for destination availability, travel time constraints, and crew assignments. It begins by iterating over the sorted list of destinations. For each destination, the method first checks if the plane can reach the destination using the `is_in_range` method. If the destination is within range, it calculates the total travel time, including the flight to the destination, the turnaround time at the airport, and the return flight back to Frankfurt. Next, it verifies that the total travel time does not exceed the maximum allowed time using the `travel_time_not_too_long` method. If the plane does not already have a crew assigned, it calls the `assign_crew_to_plane` method to assign a crew to operate the flight. If no crew can be assigned, the plane remains grounded, and the method stops further processing for that destination. If the plane already has a crew assigned, it checks if the crew can operate the flight without exceeding their working hours using the `can_work` method. If is returns `False`, the `assign_crew_to_plane` method is called again to replace the crew. If assigning a crew fails, the method stops further processing for that flight, and the plane remains idle. Once all conditions are met, the flight is successfully scheduled. The `calculate_profit` method is called to compute the profit for the route, and this profit is added to `total_profit`. The current destination is then removed from the list of destinations to ensure it is not scheduled again. Finally, the method prints the details of the flight, such as the plane registration, aircraft type, destination city, departure time, and the expected return time to Frankfurt. If no destinations are reachable, the method sets the `no_more_flights` attribute to `True`, indicating that no further flights can be scheduled for the day. This ensures the simulation ends when all options have been exhausted.

Upon defining the lists `planes`, `destinations` and `crew_members`, which contain instances for the corresponding classes and hand over the relevant attributes, an instance for the `Simulation` class is created. Lastly, the method `simulation_run` is called to initiate the simulation.

4 Implementation

The project uses modules and libraries to enhance the functionality. The main programming language is Python, but additional libraries like `datetime` and `random` are used to manage time calculations and generate random values to add variation to the demand for routes. The `timedelta` class helps calculating time differences, which is particurlarly relevant for travel times and crew working hours.

To handle the sorting of destinations by their estimated demand, the program uses a merge sort algorithm, which organizes lists efficiently to determine the most profitable destinations. The implementation of the `merge` method is a key step in the development of the program and will be further explained in more detail below.

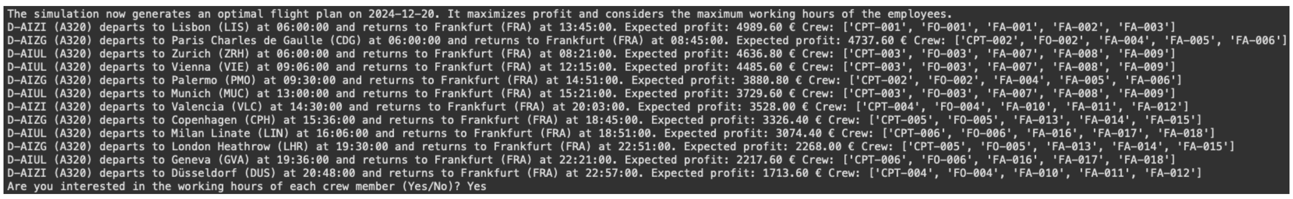
The method takes two lists, `lower` and `upper`, as inputs. These lists represent smaller chunks of destinations and the goal of the `merge` method is to combine these two lists into a single sorted list. To start, an empty list `destinations_sorted` is created. this list will store the final sorted destinations. The `while` loop runs as long as both `lower` and `upper` lists are not empty. In each iteration, the algorithm compares the `estimated_demand` value of the first destination in the `lower` list (`lower[0]`) with the first destination in the `upper` list (`upper[0]`).

- If `estimated_demand` of the destination in the `lower` list is greater than the one in the `upper` list, the destination from the lower list is removed using the `pop(0)` method and added to the `destination_sorted` list.
- Else, the destination from the `upper` list is removed with `pop(0)` and added to `destinations_sorted`.

This process ensures that the largest `estimated_demand` values come first, creating a descending order. The algorithm keeps removing and appending the greater value from the two lists until one of the lists becomes empty. At this point, the remaining items in the non-empty list (either `lower` or `upper` are added directly to `destinations_sorted`. In summary, the `merge` method picks the larger `estimated_demand` from two lists, one element at a time, and combines them into a single sorted list.

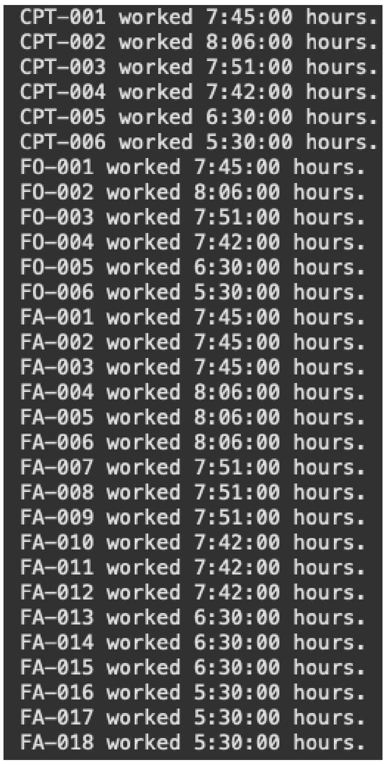
5 Results and Discussion

The output of the program is a chronological flight schedule showing each plane's departure and return times, along with the destination and expected profit. Each flight lists the assigned crew members, including captain, first officer, and flight attendants. Additionally, the total working hours for each crew member can be displayed upon request by the user.



```
The simulation now generates an optimal flight plan on 2024-12-20. It maximizes profit and considers the maximum working hours of the employees.
D-AIZI (A320) departs to Lisbon (LIS) at 06:00:00 and returns to Frankfurt (FRA) at 13:45:00. Expected profit: 4989.60 € Crew: ['CPT-001', 'FO-001', 'FA-001', 'FA-002', 'FA-003']
D-AIZG (A320) departs to Paris Charles de Gaulle (CDG) at 06:00:00 and returns to Frankfurt (FRA) at 08:45:00. Expected profit: 4737.60 € Crew: ['CPT-002', 'FO-002', 'FA-004', 'FA-005', 'FA-006']
D-AIUL (A320) departs to Zurich (ZRH) at 06:00:00 and returns to Frankfurt (FRA) at 08:21:00. Expected profit: 4636.80 € Crew: ['CPT-003', 'FO-003', 'FA-007', 'FA-008', 'FA-009']
D-AIUL (A320) departs to Vienna (VIE) at 09:06:00 and returns to Frankfurt (FRA) at 12:15:00. Expected profit: 4485.60 € Crew: ['CPT-003', 'FO-003', 'FA-007', 'FA-008', 'FA-009']
D-AIZG (A320) departs to Palermo (PMO) at 09:30:00 and returns to Frankfurt (FRA) at 14:51:00. Expected profit: 3880.80 € Crew: ['CPT-002', 'FO-002', 'FA-004', 'FA-005', 'FA-006']
D-AIUL (A320) departs to Munich (MUC) at 13:00:00 and returns to Frankfurt (FRA) at 15:21:00. Expected profit: 3729.60 € Crew: ['CPT-003', 'FO-003', 'FA-007', 'FA-008', 'FA-009']
D-AIZI (A320) departs to Valencia (VLC) at 14:30:00 and returns to Frankfurt (FRA) at 20:03:00. Expected profit: 3528.00 € Crew: ['CPT-004', 'FO-004', 'FA-010', 'FA-011', 'FA-012']
D-AIZG (A320) departs to Copenhagen (CPH) at 15:36:00 and returns to Frankfurt (FRA) at 18:45:00. Expected profit: 3326.40 € Crew: ['CPT-005', 'FO-005', 'FA-013', 'FA-014', 'FA-015']
D-AIUL (A320) departs to Milan Linate (LIN) at 16:06:00 and returns to Frankfurt (FRA) at 18:51:00. Expected profit: 3074.40 € Crew: ['CPT-006', 'FO-006', 'FA-016', 'FA-017', 'FA-018']
D-AIZG (A320) departs to London Heathrow (LHR) at 19:30:00 and returns to Frankfurt (FRA) at 22:51:00. Expected profit: 2268.00 € Crew: ['CPT-005', 'FO-005', 'FA-013', 'FA-014', 'FA-015']
D-AIUL (A320) departs to Geneva (GVA) at 19:36:00 and returns to Frankfurt (FRA) at 22:21:00. Expected profit: 2217.60 € Crew: ['CPT-006', 'FO-006', 'FA-016', 'FA-017', 'FA-018']
D-AIZI (A320) departs to Düsseldorf (DUS) at 20:48:00 and returns to Frankfurt (FRA) at 22:57:00. Expected profit: 1713.60 € Crew: ['CPT-004', 'FO-004', 'FA-010', 'FA-011', 'FA-012']
Are you interested in the working hours of each crew member (Yes/No)? Yes
```

Figure 3: Output of the simulation.



```
CPT-001 worked 7:45:00 hours.
CPT-002 worked 8:06:00 hours.
CPT-003 worked 7:51:00 hours.
CPT-004 worked 7:42:00 hours.
CPT-005 worked 6:30:00 hours.
CPT-006 worked 5:30:00 hours.
FO-001 worked 7:45:00 hours.
FO-002 worked 8:06:00 hours.
FO-003 worked 7:51:00 hours.
FO-004 worked 7:42:00 hours.
FO-005 worked 6:30:00 hours.
FO-006 worked 5:30:00 hours.
FA-001 worked 7:45:00 hours.
FA-002 worked 7:45:00 hours.
FA-003 worked 7:45:00 hours.
FA-004 worked 8:06:00 hours.
FA-005 worked 8:06:00 hours.
FA-006 worked 8:06:00 hours.
FA-007 worked 7:51:00 hours.
FA-008 worked 7:51:00 hours.
FA-009 worked 7:51:00 hours.
FA-010 worked 7:42:00 hours.
FA-011 worked 7:42:00 hours.
FA-012 worked 7:42:00 hours.
FA-013 worked 6:30:00 hours.
FA-014 worked 6:30:00 hours.
FA-015 worked 6:30:00 hours.
FA-016 worked 5:30:00 hours.
FA-017 worked 5:30:00 hours.
FA-018 worked 5:30:00 hours.
```

Figure 4: Output of the working hours upon request by the user.

The output of the program effectively addresses the challenges outlined in the problem statement by optimizing the aircraft fleet utilization to capture market demand while respecting operational constraints. The generated flight schedule ensures that planes are effectively assigned to routes based on profitability, considering turnaround times and the night flight ban. Additionally, the program respects the working hour constraints of crew members, ensuring that no crew exceeds their maximum allowable hours, which guarantees realistic flight operations. The inclusion of crew working hours further supports resource management by providing clear insights into crew utilization. By balancing profitability, crew constraints, and operational feasibility, the program offers a practical tool for optimizing fleet operations, making it a valuable to gain an understanding of airlines' approach to maximizing efficiency.

Developing this output came along with several challenges. One of the main difficulties was maintaining a modular code structure and breaking down the methods into the smallest possible components. A major challenge was implementing the `Crew` class and developing the working hour constraint. Initially, the approach was to constantly check if the `total_hours` plus the `travel_time` exceeded `max_working_hours`. However, after extensive debugging, a new issue was discovered where a crew that could not operate the next flight assigned to their plane was incorrectly selected for a shorter flight later in the day, while waiting time at the airport should also being counted as working time. This does not reflect how crew operations work in reality, as a single crew is assigned to only one plane per day. To solve this issue, the functionality was redesigned from scratch using a new approach. Instead of constantly checking the total hours, a `shift_end` attribute was introduced for each instance of the `Crew` class. The program now checks if the `current_time` plus the `travel_time` is still within `shift_end`. This effectively determines whether a crew can operate the next flight, resolving the earlier problem and ensuring realistic crew operations.

A key learning derived from developing this program and the challenges was that breaking down code into smaller, modular methods makes it much easier to follow along the functioning of the code. Through the problems with the `Crew` class, it was proven that a clear and structured approach helps identify issues early and enables easier implementation of alternative solutions when problems arise. Moreover, a key learning was that debug prints are the most useful way to isolate issues of the code.

6 Conclusion and Future Work

In conclusion, the program successfully achieves its goal of generating an optimal flight schedule for a fleet of planes. It ensures that every aircraft of the fleet gets a crew and a route assigned, while respecting operational constraints, such as the maximum working hours for the crew and the night flight ban. The implementation calculates the departure and return times for each plane and determines the expected profit for every route. Additionally, the program allows users to view the total working hours of each crew member. Overall, this project demonstrates an effective way to optimize the utilization of resources while meeting the defined requirements.

Throughout the semester, and this project in particular, I have successfully achieved most of the learning objectives. Coming from a non-technical background, the Python programming language was entirely new to me. However, upon completing this course and this project, I can state that I gained a solid understanding of the fundamentals of Python programming concepts and techniques. I applied my knowledge to this project by using different data types, variables, operators, conditional statements and loops. Moreover, I demonstrated my ability to work with Python collections such as lists, tuples and dictionaries. I showcased my proficiency in implementing advanced list operations, such as list comprehensions and list slicing, thereby highlighting my understanding of core Python programming concepts.

In addition, I have demonstrated that I am able to identify a problem and come up with a running solution. It cannot remain unmentioned that I occasionally relied on research for certain methods, especially when working with external libraries such as `datetime`. I learned to foster a modular approach when developing a framework for a solution, which had the greatest learning effect on me. While understanding Python syntax was not particularly challenging, the logical structure of the code was a greater difficulty. Nevertheless, having successfully designed and implemented the code for this project, I can confidently affirm that I met the second learning objective.

The code of this program incorporates imperative Python language features, such as loops and conditional statements, which focus on command-driven execution. It also integrates declarative features, like list comprehensions, which emphasize a result-oriented approach. Additionally, the entire project is coded in an object-oriented programming style. I can conclude that I have developed a solid understanding of these three Python programming paradigms and demonstrated my ability to apply them.

Although this project does not make use of an API, I utilized `random` and `datetime` as external libraries for this project. While I implemented relatively straightforward methods from these libraries and did not develop advanced expertise in their usage, I would conclude that I have only partially achieved this particular learning objective. Nonetheless, the use of APIs is an area that greatly interests me, and I aim to enhance my skills in this field in the future.

By considering which algorithm to use for the sorting of the list of destinations, I learned about the advantages and disadvantages of various sorting algorithms and their respective complexities. While the merge sort algorithm was chosen for scalability reasons, as it is particularly suitable for handling larger data, I was able to implement and integrate it into the program successfully. Through this project, I have proven my capability to apply a sorting algorithm on my own. Thus, I am convinced that I met this learning objective, although I would not be able to implement other sorting algorithms such as binary search or bubble sort into a program without additional research. That being said, the learning goal was met, but I acknowledge that there is still room for further improvement.

Moving forward, several features could be added to enhance the program's functionality and usability. The next step would be to implement long haul flights. The complexity of this functionality lies in the fact that these flight operations differ significantly from short haul flights, as planes operating these routes do not return to Frankfurt Airport in the evening. Another challenge is the need for extended crew rest periods, as well as the fact that one aircraft engages multiple crews simultaneously. Additionally, incorporating planes with different capacities into the fleet would allow routes with lower demand to be served more efficiently. The aim would be for the flight plan to take aircraft occupancy into account when assigning planes to specific routes. On a higher level, the ultimate goal of the simulation would be to incorporate disruptions and to simulate aircraft unavailability due to technical issues. The objective would be to develop an optimized flight schedule that is robust against unexpected events and operational disruptions.

7 References

Busbee, K. L., & Braunschweig, D. (2018). Modular Programming. https://press.rebus.community/programmingfundamentals/chapter/modular-programming/

Kenan, N., Jebali, A., & Diabat, A. (2018). An integrated flight scheduling and fleet assignment problem under uncertainty. Computers & Operations Research, 100, 333–342. https://www.sciencedirect.com/science/article/pii/S0305054817302216

Ricciardi, A. (2024, December 7). Modular Programming: Benefits, Challenges, and Modern Applications. DEV Community. https://dev.to/alex_ricciardi/modular-programming-benefits-challenges-and-modern-applications-5395

Sherali, H. D., Bish, E. K., & Zhu, X. (2006). Airline fleet assignment concepts, models, and algorithms. European Journal of Operational Research, 172(1), 1–30. https://www.sciencedirect.com/science/article/pii/S0377221705002109