**The University of Southampton**

**Academic Year (2017/2018)**

**Faculty of Social, Human and Mathematical Sciences**

**Mathematical Sciences**

**MSc Dissertation**

A Heuristic Approach to Aircraft Stand Allocation

Paba Senadheera

A dissertation submitted in partial fulfilment of the MSc in Operational Research

This project is entirely the original work of Paba Senadheera. Where material is obtained from published or unpublished works, this has been fully acknowledged by citation in the main text and inclusion in the list of references.

Word Count: 17020 words

# EXECUTIVE SUMMARY

The infrastructure at an airport includes a variety of different buildings and facilities that cater to the passengers and the aircraft, passing through the airport. An integral infrastructure that caters to both the passengers and aircraft are the “Aprons” consisted of “Stands”, at the airport pavement. Aprons are the interface between the terminal building and the airfield; stands are the parking spaces of the aircraft.

There are different types of aprons and stands based on their location and distance away from the terminal building. Stands closest to the terminal building are called “Pier-served Stands” and they can be reached, by the passengers, through bridges. Stands further away are called “Remote Stands” – these require the passengers to use staircases to board and deplane the aircraft. The process of moving an aircraft between a remote and pier-served stand is called “Towing”.

The stand allocation problem is an optimisation problem with the goal of promoting an efficient use of the aircraft parking spaces. The stands are used by the flight during their “turnarounds” (i.e. the time an aircraft spends at the airport between its arrival and departure). The assigning of the turnarounds to the stands is restricted by many rules including the size of the aircraft and the origin/destination of the flight. Another restriction is the “Buffer Times” – this refers to the 25 or 30 minutes when the stand is empty, before and after an allocated flight.

**Problem Description**

The project focused on solving a stand allocation problem for Terminal 5 at Heathrow Airport, through a heuristic approach.

The 2 primary objectives for this project were:

1. Automate the data manipulation process to compute the turnaround times of flights.
2. Construct and implement a heuristic that allocates stands to aircraft such the number of pier-served passengers would be maximised.

The other aims of the project were to have:

* an automated data manipulation process that was feasible with yearly flight data;
* a constructed heuristic that assigned a feasible stand to all the flights;
* a heuristic that accounted for the potential delays or early arrivals of flights.

**Main Findings**

* The Python model for the data manipulation was successful – *Turnarounds table produced was accurate and applicable to yearly flight data.*
* Stand Allocation Heuristic was effective at increasing the number of pier-served passengers.
* Despite only having 52.4% successful allocations, the schedule from the heuristic solution had a very high Pier-Service Level.
* **Pier-Service Level = 81.1%**

**Methodology**

The flight turnarounds were calculated, and the turnarounds splitting was performed. This separated the flights into “overnight parking operations” and “stand operations” that arrive and leave on the same day.

The *Stand Allocation Heuristic* formulated was a hybrid heuristic that combined concepts from the Greedy heuristic and the Breakout Local Search heuristic. It followed a “first come, first assigned” policy because it is a common practise in the industry. Thus, flights with the earliest arrival times were chosen greedily to be allocated first. If more than one feasible pier-served or remote stand was available, the heuristic chose to allocate to the stand that had served the highest number of passengers.

The heuristic produced an initial schedule which was then checked to identify flights, in each stand, that didn’t adhere to the buffer times restriction. If any overlapping existed, these flights were removed from the schedule and reallocated to other feasible stands.

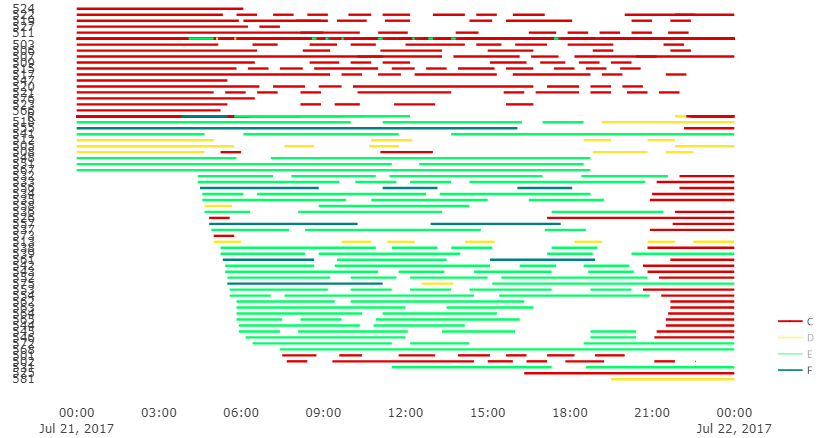
The procedure of the *Stand Allocation* heuristic segregated the flight data by fleet type (F, E, D, C) and made allocations one fleet at a time. This was because the compatibility of stands depended strongly on the size of the aircraft. This was because stands that could facilitate larger aircraft were big enough to hold smaller aircraft but not vice versa.

In industry, airport planners create the stand allocation schedules daily due to the high probability of changes in the flight schedule. For this reason, the *Stand Allocation* heuristic was also designed to work with daily flight data.

All the models, for data manipulation and the heuristic, were made using the programming language Python. The models were then tested on the flight schedule for the 21st July 2017.

**Results**

* The turnarounds table showed these statistics about the training flight data:
* The stand allocation schedule, from the heuristic is shown below. It resulted in a pier-service level of 81.1%. The results also showed that, for some fleets, there were no remote allocations except for the overnight parking.



The Stand Allocations Schedule for 21st July 2017

* The percentage of flights that were assigned to a stand was 52.4%. A summary of the number of flights and passengers with failed allocations is shown below.

**Recommendations**

Heuristic approaches are methods that provide solutions that are good enough when solving an extremely complex optimisation problem. The results suggested that the model created for the data manipulation and the heuristic model met the objectives and all the aims, except for 1. This was that not all the flights were allocated to a stand.

Recommendations to improve the this are:

* The heuristic proved to find a good solution for its objective. So, reformulate the allocation problem to have multiple objectives, including one to assign a stand to all flights.
* Include the BA bases and other stands that were excluded from this study due to its complexity.
* Reformulate the SAP and heuristic and model with both normal and short buffer times.
* Refine parts of the Python model to increase the overall processing speed of the model.

# ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr Athanassios N. Avramidis, my supervisor, who was always available to give me academic guidance during the entire course of this project.

I also want to thank Mr. Eloy Mora Vargas and Mr. Alberto Rivas Cid, from Arup, for all the support with data collection, helping me understand the scope of this project, and for the opportunity to work so closely with real-life data.

# TABLE OF CONTENTS

[EXECUTIVE SUMMARY 1](#_Toc536100550)

[ACKNOWLEDGEMENTS 6](#_Toc536100551)

[TABLE OF CONTENTS 7](#_Toc536100552)

[LIST OF FIGURES 9](#_Toc536100553)

[ABSTRACT 10](#_Toc536100554)

[INTRODUCTION 11](#_Toc536100555)

[1. THE AIRPORT AS A SYSTEM 13](#_Toc536100556)

[1.1 The Airport Apron/Gate Complex 13](#_Toc536100557)

[1.1.1 *Apron Sizing* 14](#_Toc536100558)

[1.1.2 *Apron Layouts* 16](#_Toc536100559)

[1.1.3 *Types of Aprons & Stands* 19](#_Toc536100560)

[1.2 Ground Handling Operations 21](#_Toc536100561)

[1.3 Managing of Resources 22](#_Toc536100562)

[2. LITERATURE REVIEW 23](#_Toc536100563)

[2.1 The Stand Allocation Problem (SAP) 23](#_Toc536100564)

[2.2 Feasibility of an SAP 25](#_Toc536100565)

[2.3 Approaches to solving an SAP 27](#_Toc536100566)

[2.3.1 *Breakout Local Search* 27](#_Toc536100567)

[2.3.2 *Metaheuristics* 30](#_Toc536100568)

[2.4 Software for Flight Stand Allocation 40](#_Toc536100569)

[*2.4.1*  *IBS Software for Airline Operations* 40](#_Toc536100570)

[*2.4.2*  *CAST Stand Allocation* 41](#_Toc536100571)

[3. METHODOLOGY 43](#_Toc536100572)

[3.1 Initial Flight Data 43](#_Toc536100573)

[3.1.1 *Flight Schedule Data* 43](#_Toc536100574)

[3.1.2 *Airport Details* 45](#_Toc536100575)

[3.1.3 *Ethical Approval* 45](#_Toc536100576)

[3.2 Heathrow Airport – Terminal 5 45](#_Toc536100577)

[3.2.1 *Stand Layout* 46](#_Toc536100578)

[3.2.2 *Operational Assumptions* 47](#_Toc536100579)

[3.2.3 *Training Data* 48](#_Toc536100580)

[3.3 Defining the Problem 49](#_Toc536100581)

[3.4 Data Manipulation 49](#_Toc536100582)

[3.4.1 *Data Cleaning* 49](#_Toc536100583)

[3.4.2 *Calculating Flight Turnarounds* 50](#_Toc536100584)

[3.4.3 *Modelling the Data Manipulation* 51](#_Toc536100585)

[3.5 Formulating the SAP 54](#_Toc536100586)

[3.5.1 *Identifying Variables* 54](#_Toc536100587)

[3.5.2 *The Stand Allocation Heuristic* 56](#_Toc536100588)

[3.5.3 *Modelling the Stand Allocation Heuristic* 61](#_Toc536100589)

[4. RESULTS 66](#_Toc536100590)

[4.1 Stand Allocation Gantt Chart 66](#_Toc536100591)

[4.2 Breakdown of Allocations for each Fleet type 69](#_Toc536100592)

[5. DISCUSSION 73](#_Toc536100593)

[6. CONCLUSION 75](#_Toc536100594)

[7. RECOMMENDATIONS 76](#_Toc536100595)

[REFERENCES 77](#_Toc536100596)

[APPENDIX I : Initial Data 79](https://sotonac-my.sharepoint.com/personal/ps1g13_soton_ac_uk/Documents/Disseration/writing/FINAL/Dissertation_Report_25282824.docx#_Toc536100597)

[APPENDIX II : Python Scripts 82](#_Toc536100598)

# LIST OF FIGURES

|  |  |
| --- | --- |
| **Figure 1.1(i)** *A satellite view of aprons from various airports, taken from*  *Google Earth……………………………………………………………….……....* | 14 |
| **Figure 1.1.1(i)** *The classification of the alphabetic ICAO codes based on*  *the size of the aircraft (Skybrary.aero, 2018)………………………….…….….* | 15 |
| **Figure 1.1.2(i)** *A diagram of the Linear stand layout for Aprons.*  *(Kazda and Caves, 2015)…………………………………………………………*. | 16 |
| **Figure 1.1.2(ii)** *A diagram of the Open stand layout for Aprons.*  *(Kazda and Caves, 2015)………………………………………………………….* | 17 |
| **Figure 1.1.2(iii)** *A diagram of the Pier stand layout for Aprons, arranged in different shapes (Kazda and Caves, 2015)………………………………..….…* | 18 |
| **Figure 1.1.2(iv)**  *A diagram of a Radial Satellite stands layout for Aprons. (Kazda and Caves, 2015)………………………………………………………….* | 18 |
| **Figure 1.2(i)** *A conveyer belt loading luggage onto the aircraft (Kazda and Caves, 2015)…………………………………………………………………...……* | 21 |
| **Figure 2.2(i)** *3 assemblies for the 3 stand operations, towing operations and buffer times (Guépet et al., 2015)……………………………………...…….* | 24 |
| **Figure 2.2(ii)** *2 sub-graphs connected to each other through the colour in the SAFP (Guépet et al., 2015)………………………………………….…….…* | 26 |
| **Figure 2.3.2b(i)** *The diagram that visualises the BBO heuristic (Uk.mathworks.com, 2018)…………………………………………………...……* | 35 |
| **Figure 2.3.2b(ii)** *The diagram that visualises the BCO heuristic*  *(Marinelli et al., 2015)…………………………………………………………...….* | 38 |
| **Figure 2.3.2b(iii)** *The diagram that visualises the B-BCO heuristic.*  *The BCO within the BBO (Marinelli et al., 2015)……………………………...…* | 39 |
| **Figure 2.4.2(i)** *The CAST user interface with the Gantt-chart view*  *(ARC, 2018)………………………………………………………………………....* | 42 |
| **Figure 2.4.2(ii)** *The CAST user interface with the apron simulation view (ARC, 2018)……………………………………………………………………..…..* | 42 |
| **Figure 3.2.1(i)** *Shows a diagram of the layout of the stands at*  *Terminal 5*........................................................................................................ | 46 |
| **Figure 4.1(i)** *The Stand Allocation Gantt chart for all the flights on*  *21st July 2017…………………………………………………………………..…..* | 66 |
| **Figure 4.1(ii)** *Section of C-code & D-code flight Stand Allocation Gantt chart (12am to 12pm on 21st July 2017*)……………………….………….……. | 67 |
| **Figure 4.1(iii)** *Section of E-code & F-code flight Stand Allocation Gantt chart (4am to 4pm on 21st July 2017)…………………………………….……..* | 68 |
| **Figure 4.2(i)** *A breakdown of no. of passengers serviced by each set of*  *C-code stands………………………………………………………...…………....* | 71 |
| **Figure 4.2(ii)** *A breakdown of no. of passengers serviced by each set of*  *D-code stands…………………………………………...………………...………..* | 71 |
| **Figure 4.2(iii)** *A breakdown of no. of passengers serviced by E-code and F-code stands…………………………………………………………………...…..* | 72 |

# ABSTRACT

The airport includes a variety of structures and facilities that cater to the passengers and the aircraft, passing through the airport. It is critical that these resources are managed properly to increase the efficiency of the airports, thus, providing a better quality of service.

This project explored the use of heuristic methods to manage the use of the aircraft parking spaces (i.e. stands). There are 2 types of parking spaces; ones that can be accessed directly from the airport terminal and the others need to be accessed by bus or mobile stairs. The task was to assign the daily flights, at Terminal 5 Heathrow airport, to the stands so that a maximised number of passengers can access the aircraft directly. A Stand Allocation heuristic was constructed, based on the Greedy and Breakout Local Search heuristics. It was modelled using “Python” and applied on to a set of training flight data.

The results helped conclude that the heuristic was able to efficiently assign the flights to stands. This was because, out of all the stands/flights pairings made, the proportion of passengers that were able to directly access their flight from the terminal was 81.1%. However, the heuristic was only able to successfully assign 52.4% of the flights to a stand.

# INTRODUCTION

Since its introduction during World War I, civil air transport has become increasingly popular. In the 1950s air transport was made into a commercial industry with the first fare structures and advertisements. By 1958, airlines were carrying more passengers across the Atlantic Ocean than ocean liners. Meanwhile, airports were being developed to serve the increasing numbers of commercial aircraft landings and take-offs (The Geography of Transport Systems, 2018). During the 1920s, airports started using lights to indicate the correct direction and angle of descent and take-off for aircraft. The colours of these lights and the flash intervals for the different indications were standardised by the International Civil Aviation Organisation (ICAO) in 1930. By the early 1960s, the rise in the number of aircraft meant more accidents related to air travel were occurring and the risk of mid-air collisions was increasing. This led to the establishment of the first national Air Traffic Systems. The development of airports was also booming with better infrastructure and facilities for the passengers and aircraft. Runways that ran 3000m long and jet bridge systems (commonly known as "Piers" in modern airports), that eliminated outdoor passenger boarding, were introduced (Schmitt and Gollnick, 2016).

Today, the trends in the growing popularity of air transportation have not changed. A study conducted in 2013 showed that the number of flights with over 6000 nautical miles had increased by 70% since 2008. It predicted that, by the year 2032, airlines around the world will carry over 29,220 new passengers and freighter aircraft. Forecasts were also made showing that the demand for air transport will increase by 5% per year, for the next 15 years (The Economic Times, 2018). This means that airports have become a main system within the air transport industry. Airports are now, more than ever, expected to improve their facilities and processes to ensure that the efficiency and comfort are greater for their daily passengers. For this reason, airports have invested greatly to focus on Airport Planning.

Airport Planning is defined as a systematic process that institutes guidelines for the efficient development of airports to allow them to achieve their local and national goals. Every airport constructs an individual airport ‘Master Plan’ which describes their blueprint for long-term advancements. One of the goals of a master plan is to set up a structure for a continuous planning process (Central Region Airports Division, 2018). This project takes a deeper look at creating a plan for airport operations – it concentrates on the operation of stand allocation for Terminal 5 at Heathrow Airport.

The sponsor for this project was Arup. Arup is a multinational company founded in London in 1946. It is formed of designers, planners, engineers and consultants who tackle complex challenges in all areas of the built environment. Within Arup, the Airport Development Team was overseeing the progress of this project. Arup has 50 years of experience in airport planning and has worked on several projects for Heathrow Airport. The ‘Terminal 5 Project’ consisted of 16 individual projects that dealt with various aspects of airport planning including the operation of the main terminal building, 60 aircraft stand and the associated airfield infrastructure (Arup.com, 2018).

There were 2 primary objectives for this project – the first was to automate the data manipulations implemented to compute the turnaround times of flights. The second objective was to construct a heuristic that allocates stands to aircraft in a manner that would maximise the number of pier-served passengers. There were several questions that the project aimed to answer – they were as follows:

1. Was the automated process of data manipulation feasible with flight data for a year?
2. Did the heuristic successfully assign a stand to all the flights?
3. Were the allocated stands compatible with the aircraft?
4. Did the heuristic account for the potential delays or early arrivals of flights?

This report introduces the main features of the airport related to flight stands and outlines the concept of a stand allocation problem. Several methods for tackling such a problem are reviewed and discusses the formulation of a heuristic to optimise the allocation of stands at Heathrow. The feasibility of using such a heuristic on a real-life problem is explored through modelling and the results are discussed to draw conclusions.

# 1. THE AIRPORT AS A SYSTEM

Within the infrastructure of air transport, the “Airport” is a key component. Every airport is made of 2 areas – the **Airside Area** and the **Landside Area.** Some examples of the infrastructures belonging to these areas are runways, taxiways and the apron/gate complex in the airside area; the passenger terminal/cargo complex and the ground access systems in the landside area.

The most important physical characteristic of an airport is its size. The size of an airport varies depending on the volume of traffic it accommodates during a given period of time, in terms of the passengers and airfreight cargos.

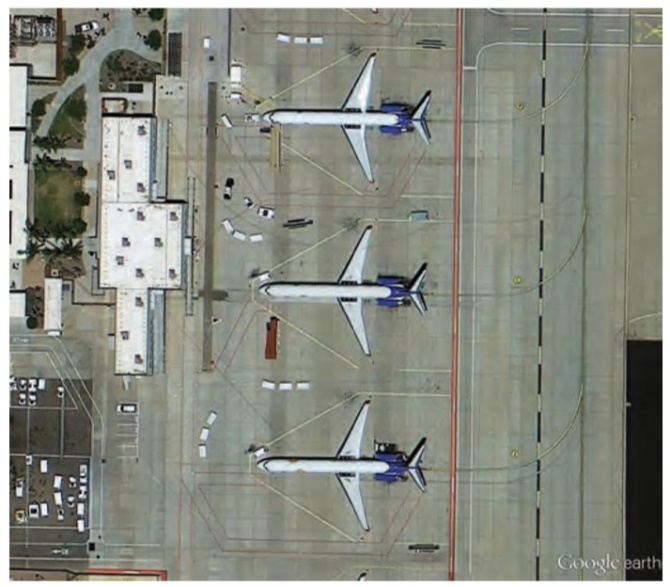
This is an important feature because it reflects the extent of the available infrastructures at the airport (Janić, 2013).

The capacity of an airport is the collective of its infrastructure and its available service facilities and equipment. The total amount of capacity required by an airport is dependent on the demand for services, over a given time. Therefore, the most important areas of operational performance are the **Demand** and the **Capacity** of an airport. The dynamic interaction between these performance aspects determines the airport’s **Quality of Service** (Janić, 2013)

For the scope of this project, the focus was on the Apron/Gate complex infrastructure in the airside area and the Ground Handling Operations that are a part of the airport operations in the landside area. This chapter introduces the details of the different configurations within an Apron and the tasks involved in Ground Handling.

## 1.1 The Airport Apron/Gate Complex

The Apron refers to the sections of the pavement area at the airport that has the aircraft parking stands and the necessary facilities and equipment to serve the aircraft during their **Turnaround** times (Janić, 2013). The turnaround time of an aircraft refers to the duration of time the aircraft spends at the airport between its arrival and departure flights. One of the basic requirements of an apron is to have the necessary permissions to allow the- on and off-loading of passengers, baggage and cargo, and the technical servicing of aircraft including refuelling (Kazda and Caves, 2015). The Figure 1.1(i) are images taken from Google Earth and shows the areas on the pavement that are the aprons.



*Figure 1.1(i): A satellite view of aprons from various airports, taken from Google Earth.*

### 1.1.1 *Apron Sizing*

Since the apron consisted of the aircraft parking spaces (stands), the suitable size of any given apron depends on the types of aircraft that intend to park at that apron. Here, the type of aircraft is defined by the size of the aircraft and is identified using the ICAO code (Kazda and Caves, 2015).

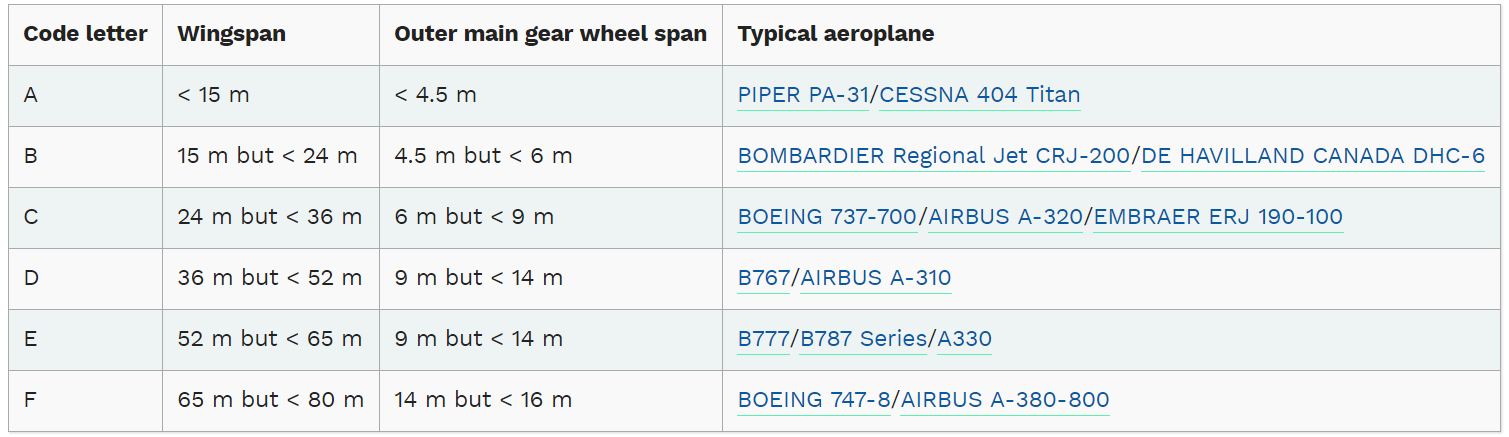
**The ICAO Codes**

The ICAO Aerodrome Reference Code is a two-part identification method that can be used to categorise the aircraft by type. The ICAO codes help to simplify the process of establishing if an aircraft of a specified type is able to use a particular aerodrome (Skybrary.aero, 2018).

An aerodrome is a place where aircraft operations are able to take place fully, for all flights carrying passengers or cargo. Anything from an airfield to a commercial airport or military base is considered an aerodrome.

The first part of the ICAO code is a numeric code from “**1 to 4**”. It identifies and categorises the aircraft based on the minimum field length required for take-off at maximum certificated take-off mass, at sea level, in [ISA](https://www.skybrary.aero/index.php/ISA) conditions in still air and with zero runway slope (i.e. the **Reference Field Length**).

The second part of the ICAO code is alphabetic – a code between “**A to F**”. These identify the aircraft based on their **Wing Span** or the **Outer Main Gear Wheel Span**. The classification of these alphabetic codes is shown in Figure 1.1.1(i)



*Figure 1.1.1(i): The classification of the alphabetic ICAO codes, based on the size of the*

*aircraft (Skybrary.aero, 2018).*

In the case of sizing an apron, only the alphabetic ICAO code is used. The aircraft wingspan determines the size of the stand, hence the size of the apron. The stands in the apron need to a minimal distance apart – this is determined using the wingspan and the wing tip clearance of the aircraft. Therefore, the required size of stands varies from **2,200 m2 for B-code** aircraft to **15,000 m2 for F-code** aircraft. The depth of the stands varies from **30 to 85 m** for codes **B to F**, respectively (Kazda and Caves, 201).

Additionally, the apron must have a space dedicated to the ground equipment, vehicle placement, taxiways, etc. that is needed for the airport operations that take place there. Thus, the total size of the apron depends on both stand size and stand operations required.

### 1.1.2 *Apron Layouts*

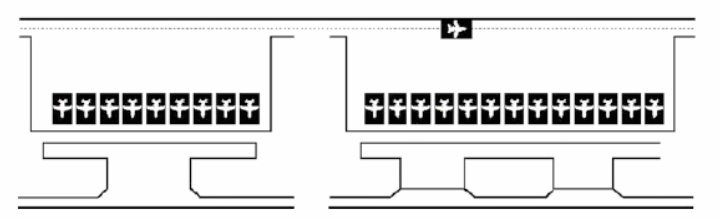
The size and manoeuvring operations of aircraft complicate the apron environment. During the periods of peak air traffic, they make it impossible for all the stands required to be placed close to the terminal building. Therefore, it is important that the stands are laid out in a proper manner, based on the size of all the aircraft.

Introduced below are some of the basic layouts for stands to be placed in the apron. Each layout has its own advantages and disadvantages; hence it is common to see a hybrid of these concepts used in industry (Kazda and Caves, 2015).

**The Linear Layout**

This is a concept where the individual stands are placed in a line, alongside the terminal building. A diagram of the layout is shown on Figure1.1.2(i) Modified versions of this layout are very commonly seen at many airports.

The advantage of this layout is that it provides a direct line of access between the terminal building and the aircraft. The installation of this layout and the required passenger loading bridges is simple. It also provides enough space for the handling equipment and staff, required at the apron. However, when this layout is applied to very large airports, the distance between the stands at the ends of the line and the central terminal building is too large for the convenience of passengers transferring between airlines.

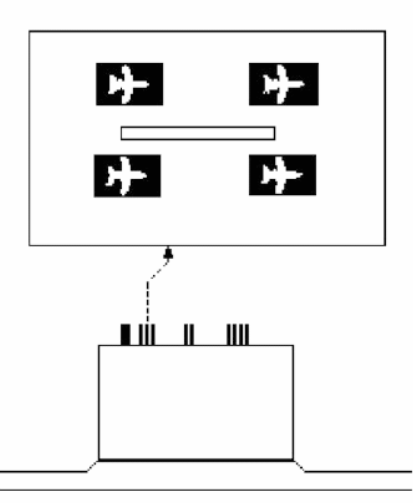


*Figure 1.1.2(i): A diagram of the Linear stand layout for Aprons. (Kazda and Caves, 2015).*

**The Open Layout**

As shown in Figure 1.1.2(ii), this layout has the stands located in rows, in front of the terminal building. The features of this layout allow many aircraft to be served, on a small space in front of the terminal building. Aprons in this layout can be optimised in terms of airport operations.

This layout requires to transportation to be provided to all passengers from distant stands. This is disadvantageous because the bus trips providing the additional transport can be unreliable. Moreover, the additional apron movements introduced by the bus trips can cause accidents.



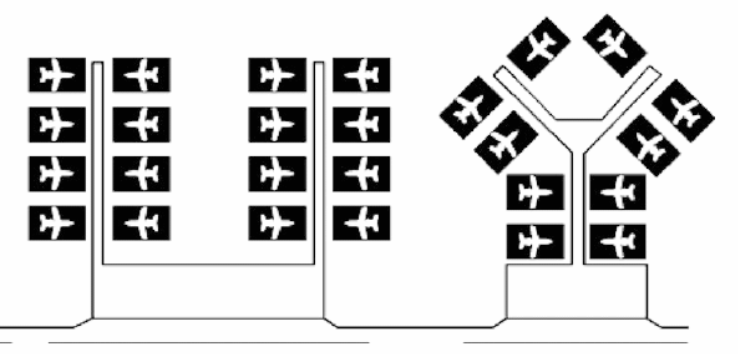
*Figure 1.1.2(ii): A diagram of the Open stand layout for Aprons. (Kazda and Caves, 2015).*

The Terminal Building

**The Pier Layout**

The concept for this layout is to have piers coming out of the terminal building and have the stands lined along the pier. This layout is the most suitable for large airports because it is the most efficient way of increasing the airport’s capacity. The shapes in which the piers are arranged depends on the space availability at the airport. The figure (Figure 1.1.2(iii)) shows the arrangement of the pier layout apron, in 2 different shapes.

The biggest advantage of the piers is that it keeps all the stands together. This allows transferring passengers to navigate easily and have direct access to the central processing area. On the other hand, this layout leads to the apron area appearing very complex and does not provide a lot of room for the ground handling equipment and vehicles on the apron.



*Figure 1.1.2(iii): A diagram of the Pier stand layout for Aprons, arranged in different shapes (Kazda and Caves,*

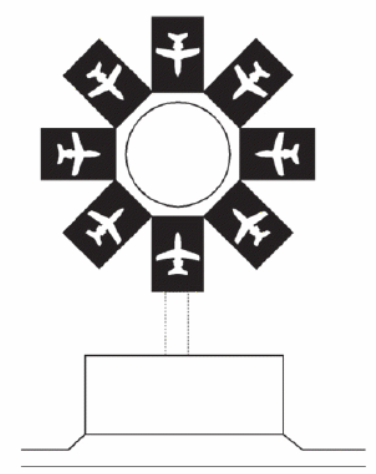
*2015).*

**The Satellite Layout**

The concept behind this layout is to have several clusters of stands, away from the terminal building, as remote passenger loading satellites. Each of the satellites will be connected to the terminal building through underground or overhead passages. The satellites are typically circular in shape and can accommodate 4 to 8 aircraft, depending on the size of the circle. If the satellites are in a linear shape, it can hold around 20 stands per side.

The satellite layout provides abundant space for all the equipment and vehicles required at the apron, and for manoeuvring the aircraft. However, the size of the satellites and the spacing between them means that the layout can only be used where the apron area is large.

Figure 1.1.2(iv) shows the set-up of a radial satellite with 8 stands.



*Figure 1.1.2(iv): A diagram of a Radial Satellite stands layout for Aprons. (Kazda and Caves, 2015).*

The Terminal Building

### 1.1.3 *Types of Aprons & Stands*

The airport system consists of different types of apron areas and stands within them. The different types of aprons have been categorised based on the distance between the apron and the terminal building, and the operations taking place at the apron (National Academies of Sciences, Engineering, and Medicine, 2013).

Some of the main types of aprons used for passenger and cargo flights have been discussed below.

**Terminal Area Apron**

The region between the terminal building and the runway one of the most critical aprons – it is the busiest area at commercial airports. This region is the ‘*Terminal Area Apron’* (National Academies of Sciences, Engineering, and Medicine, 2013)*.*

These aprons are where passengers enplane and deplane the aircraft. During the time of passenger movement to and from the aircraft, the ground handling equipment is used to service the aircraft. Some of these services include catering, de-icing, refuelling and loading or unloading luggage.

The stands within the terminal area aprons can be recognised as **Contact/Pier-served Stands** and **Non-contact/Remote Stands**. The contact, or pier-served, stands are those that that located directly along the terminal building of the airport. When the aircraft is parked here, passengers use bridges to enplane and deplane. The non-contact or remote stands are those that are located further away from the terminal building. However, these stands are still sufficiently close enough to allow passengers to enplane and deplane through stairs or ramps. Remote stands do not have a direct link to the terminal building, like the pier-served stands do.

**Remote Aprons**

The Remote, or RON, aprons are those that are located at a considerable distance away from the terminal building. If these aprons were to be accessed by passengers, it would require a bus service.

The main purpose of the RON aprons is the storing of aircraft. Most passenger aircraft do not operate at overnight so commercial airports need to accommodate the large numbers of aircraft that need overnight parking. These numbers often exceed the number of stands available, therefore the RON aprons are used to store the overnight aircraft. During the daytime, the RON aprons are used by aircraft with exceptionally long turnaround times or by aircraft that require some light maintenance (National Academies of Sciences, Engineering, and Medicine, 2013).

Ideally, the stands at these aprons are not used to passenger loading, however during periods of high air traffic, airports use bus services to connect passengers to the RON aprons. The process of moving an aircraft from a RON apron to a terminal area apron is called **Towing.**

The terminal area and remote aprons are the aprons that are most important for the quality of service provided by the airports to the passengers. This is because they are the aprons responsible for interacting with the passengers. In terms of the services provided for the aircraft/airlines, the most important apron type is the maintenance apron.

**Maintenance Aprons**

As the name suggests, the maintenance aprons are where any maintenance procedures are conducted on the aircraft. The aircraft maintenance activities that need to be performed are different for aircraft from each airline – the guidelines for this are specified by the airlines. The work done at these aprons are very important to ensure that all the aircraft are safe for flight. The actual maintenance facilities available depends on the size and quality of the airports.

The maintenance aprons are usually consisting of a hangar building, large enough to fit aircraft from all fleet types. Within the maintenance airports, there are no individual stands for the aircraft – this is because ground handling operations or manoeuvres are not performed here (National Academies of Sciences, Engineering, and Medicine, 2013).

## 1.2 Ground Handling Operations

The ground handling operations refer to processes that are performed on the aircraft, during the enplaning or deplaning of passengers, to prepare it for flight. Some examples here (National Academies of Sciences, Engineering, and Medicine, 2013) these processes are described below :

* **Fuelling:** This is the filling of the aircrafts' fuel tanks with a predetermined amount of fuel. The amount of fuel carried is dependent on the requirements of the scheduled flight. This task is completed by fuelling trucks or through a hydrant fuelling system.
* **Baggage handling:** The luggage is screened and sorted at the terminal building before they are loaded onto departing aircraft. After the arrival of an aircraft, the baggage is transported to the terminal building for claiming. The method used for loading and unloading the baggage depends on the size of the aircraft – large aircraft conveyor belts while smaller aircraft manpower. Figure 1.2(i) shows the loading of luggage using a conveyer belt.



*Figure 1.2(i): A conveyer belt loading luggage onto the aircraft (Kazda and Caves, 2015).*

* **Preconditioned air:** The aircraft uses air that has been pre-cooled or heated to maintain the ambient temperature within the aircraft. sometimes this preconditioned air is used to regulate the temperature of the passenger loading bridges.
* **Catering:** This procedure is where the aircraft is restocked with the necessary food, close to the scheduled flight time. The catering process also concerns the removal and disposal of the waste from the previous flight.
* **De-icing:** The de-icing process is where any ice, snow or frost on the body of the aircraft is removed. Once it's removed, the residual fluids must be removed from the apron. To do this, aprons are equipped with de-icing fluid collection systems. Otherwise, de-icing fluid recovery vehicles or glycol recovery vehicles are used to remove the de-icing fluids on the apron.

In addition to these, there are many more ground handling operations that need to be performed before and after each flight. Performing these operations take time – these periods of time before and after a flight is referred to as the **Buffer Times.** The buffer times required by an aircraft depends on the size of it – i.e. fleet type.

## 1.3 Managing of Resources

All the components of the infrastructures in an airport are important resources. These resources facilitate the essential airport operations that need to take place in order for the airport to function as a system. It is, therefore, very important to manage these resources in such a way that their use is optimised.

This is done through Airport Planning where airports are able to identify their problems, set their goals for optimisation and create a plan to reach these goals.

Aprons are such a resource that facilitate the airside operations required at an airport. The stands in the apron are one of the limited resources that provide the platform for all the ground handling operations to occur. This is one of the most critical resource-operation combinations because it is what ensures the flights take place normally and safely. The use of the apron can be optimised by the airport planning processes like the Stand Allocation Problem and the Gate Allocation Problem.

The successful optimisation of the use of airport resources helps the airports to become more cost and time efficient. Thus, increasing the quality of services provided to the passengers, the aircraft and the airlines.

# 2. LITERATURE REVIEW

This chapter reviews several research papers conducted, in the past, to tackle the stand allocation problem (SAP). It also reviews some software currently used in the industry for stand allocations.

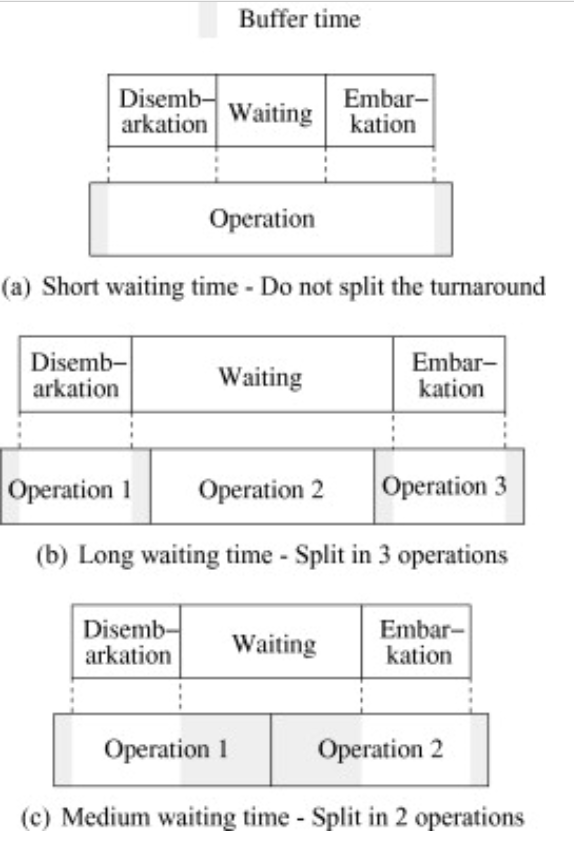
## 2.1 The Stand Allocation Problem (SAP)

The SAP deals with the assignment of aircraft to parking spaces; referred to as ‘Stands'. The objective of the SAP is usually to maximise the number of passengers that enplane and deplane from contact stands (Guépet et al., 2015). Airports often choose to set multiple objectives for their SAPs. Some other objectives of SAPs include the minimisation of towing time, the minimisation of taxi delays due to push-back or the increase in the robustness of the stand allocation schedule (Gate Assignment Solution (GAM) Using Hybrid Heuristics Algorithm, 2009). The SAP can be oriented to optimise passenger related features or airport-related features, depending on which objectives are prioritised by the airport planners (Justesen, Larsen and Dohn, 2014).

Every SAP considers the flight turnarounds as 3 ‘Stand Operations’ – the *Arrival* of the flight, the time spent *Waiting* on the stands and the *Departure* of the flight. The arrival and departure operations consist of the procedures for enplaning and deplaning the passengers and luggage, and for aircraft ground handling. The waiting period of the aircraft is not considered as a separate stand operation for flights with a short turnaround time. In this case, the arrival and departure of the aircraft can take place from the same stand. Otherwise, the aircraft would be towed off to another stand, often away from the terminal building, during the waiting period. This counts as 2 further stand operations, for towing the aircraft after arrival and before departure, to be added to the SAP. The airports usually consider the towing operations as expensive costs (Guépet et al., 2015).

There is always a chance that the departure and arrival of flights are delayed or early. The probability of such events depends on various factors. As listed by the EUROCONTROL (Eurocontrol, 2018), some of these factors include delays in boarding; technical defects; problems at the airport from which the flight is arriving; baggage handling and severe weather conditions. In order to account for these, a ‘Buffer Time’ is added before the arrival and after the departure stand operations (Guépet et al., 2015). The duration of the buffer times depends on the type of aircraft; it also varies between the arrival buffers and departure buffers.

The different assemblies for these 3 stand operations, towing operations and buffer times are displayed in Figure 2.2(i).



*Figure 2.2(i): 3 assemblies for the 3 stand operations, towing operations and buffer times (Guépet et al., 2015).*

The SAP is very closely related to the Gate Allocation Problem hence there are many works of literature that interchange the techniques used to solve these 2 problems. The gate is where the passengers get ticketed and checked in, while the stand is the parking position of the plane at the airport’s apron (Gate Assignment Solution (GAM) Using Hybrid Heuristics Algorithm, 2009).

## 2.2 Feasibility of an SAP

SAPs consist of many hard and soft constraints that are needed to ensure that the objectives are fulfilled. These constraints define the compatibility between the available stands and the flights. The most commonly used constraints include: the arrival-departure times, the size and type of the aircraft, the origin and destination of the flights and airline preferences (Chun et al., 2000). Due to these constraints, several studies have shown that the SAP is an NP-hard optimisation problem.

The study, “*Exact and heuristic approaches to the airport stand allocation problem”* (Guépet et al., 2015), used a Graph Colouring Problem (GCP) to prove the feasibility of the SAP. The first step of the proof was to define a Stand Allocation Feasibility Problem (SAFP) based on the SAP. The SAFP was modelled as a GCP. The parameters of the SAFP model were as follows:

* – an instance of the SAFP, where
  + was the set of stand operations for a flight
  + was the set of all available stands
  + was the set of stands compatible with flight
* – an undirected graph, where
* were vertices on the graph and corresponds to a stand
* were also vertices and corresponds to a stand operation
* was the set of edges on the graph

When solving the SAFP, colours were assigned to vertices. This was done based on the subgraph induced by the stands – the induced subgraph was the clique hence the graph was coloured by or more different colours. The subgraph for the stand operations, induced by the vertices, was an interval graph.

The next step of solving the SAFP was to add edges to each subgraph so that each vertex ( or ) was linked with all the immediate vertices that had a different colour. Vertices with the same colour were never linked. This same logic was then applied to add edges that linked the 2 subgraphs. Figure 2.2(ii) shows an example of what a graph looked like for a single SAFP instance.



*Figure 2.2(ii): 2 sub-graphs connected to each other through the colour in the SAFP (Guépet et al., 2015).*

The edges of each graph, therefore, represented the incompatibilities within the constraints of the SAP. Based on this construction there were no vertices of the same colour, within the subgraphs, that overlapped. The graphs also showed that each operation of a given colour was assigned to a stand of that same colour. This led the study to several conclusions: the first was that the solution to an instance is a feasible solution, if and only if the graphs follow an -colouring system. The second conclusion was that the graph , that links all the graphs of the SAFP, is -colourable if all graphs were -colourable. This proved that the graphs were NP-complete hence proving the NP-completeness of the SAFP.

The study extends further to model the SAP itself. Here an instance of the SAP was defined as , where was an affinity matrix that realised the assigning of operation to stand . The affinity matrix was defined as .

Since the SAFP was in NP, it was concluded that the SAP was also in NP. The study then concludes that has a feasible solution only if has a solution of , hence proving the NP-hardness of the SAP.

## 2.3 Approaches to solving an SAP

### 2.3.1 *Breakout Local Search*

*“Breakout local search for the multi-objective gate allocation problem”* is a study conducted on solving a Gate Allocation Problem (GAP) with 9 objectives, using a Breakout Local Search (BLS) heuristic (Benlic, Burke and Woodward, 2017). Although the study was titled ‘gate allocation', in its context, the term ‘gate' referred to both contact and remote stands. The GAP was formulated with the following parameters:

* – The set of flights.
* – The set of available gates.
* – A set of adjacent gates. These gates cannot be used simultaneously due to shadow restrictions.
* – This is a set of gate subsets that belong to the same gate group.
* – The set of stand operations obtained by splitting the flight turnarounds.
  + The inequality was used for the splitting. Here, and were the expected departure and arrival times of flight , respectively. The flights that fitted this inequality were split into 3 stand operations and the others into 2 operations. The duration of time was dependent on whether the aircraft size was small, medium or large.
* – This meant that stand operation was assigned to the gate .
* – This meant that stand operation was not assigned to the gate .

The objectives addressed in this study were oriented to improve the airport experience for both the passengers and the aircraft. The features that were optimised were separately described as hard and soft constraints. The hard constraints for the BLS were:

1. *Gate Compatibility.* The usability of each gate was restricted by the size of the aircraft.
2. *One gate per stand operation.* This constraint ensures that each stand operation is definitely assigned to exactly one gate.
3. *Overlapping Restriction.* This is an extension from the previous constraint and it suggested that no 2 operations can be assigned to the same gate, at the same time.
4. *Shadow Restriction.* This constraint was to ensure that any 2 gates , from the set of shadowing gates , were not used at the same time.

These hard constraints guaranteed the BLS to produce a feasible solution. On the other hand, the soft constraints were used to define the objective function that was to be optimised by the GAP. The 4 main domains handled by the soft constraints were:

1. *The ‘idle times’ between conflicting flights.* This domain considered the idle time between flights that were in conflict due to being at the same gate; being at gates under shadow restrictions or being at gates within the same gate group. The idle times were quantified by a cost function which was minimised. This increased the robustness of the solution by reducing the chance of conflict.
2. *Flight to Gate Preference.* This category dealt with 4 soft constraints. The objectives were (i) maximise the usage of gate space, (ii) maximise the fulfilment of each airlines gate preferences for flights. (iii)minimise the number of towing operations to terminal gates, and (iv) minimise the total number of passengers passing through remote gates. The penalties of these objectives were quantified by a function for the total number of passengers served by remote gates.
3. *The Towing Operations.* The focus of this category was to minimise the number of towing operations and gate changed.
4. *Passenger Walking Distance.* Another objective was to minimise the total distance that passengers must walk within the airport terminal. This aimed to minimise the probability of passengers missing their connecting flights. The cost function for this objective was dependent on the number of transfer passengers carried by each flight.

BLS is a variation of the classic local search heuristic and was created in the recent past. It follows the general framework of an iterated local search. The basic idea of the BLS is to apply a descent-based local search on the current search space region; then diversify the search after a local optimum is found. The general framework of the BLS is listed below:

1: – *‘GenerateInitialSolution’*

2: – Initialize the number of perturbation moves

3: **while** stopping condition not reached **do**

4: – *‘DescentBasedSearch*’ on

5: – *‘DetermineJumpMagnitude*’ on (, , history)

6: – *‘DeterminePerturbationType*’ on (, history)

7: *‘Perturb*’ through (, , , history)

8: **end while**

The *‘GenerateInitialSolution’* function employed a Memory-based Greedy Construction Heuristic (MGCH). This heuristic used a function to determine if a feasible gate existed for a given operation – if it did was allocated to . If not, a backtracking function was used to allocate . By the end of the MGCH, the initial solution was created.

Since the BLS intensifies the current search region, once the initial solution was found, all the operations that violated the hard constraints, with high penalty values, were reallocated to improve the optimum. In this study, the reallocation procedure used a Neighbourhood Local Search Heuristic.

The next step of the proposed BLS was to use a descent-based search procedure to determine the number of moves suitable for the next perturbation phase of the BLS. To perturb through the current optimum the BLS chose between using a ‘*Directed Perturbation’* or a *‘Critical Element-guided Perturbation’.* The directed perturbation method chooses the operation-gate pair with the highest penalty value, from the current allocation, then reallocates it. The critical element-guided perturbation method deallocates a random operation-gate pair, sorts the feasible gates in decreasing order of penalty value, then reallocate the operation to the most suitable feasible gate.

The BLS model was tested using data from the Manchester Airport. The results from the model provided a final allocation that was significantly better in quality than the initial solution . The BLS procedure particularly improved the levels of separation between flights at the same gate and the number of towing operations.

### 2.3.2 *Metaheuristics*

It is common practice for 2 or more heuristics to be used together to solve an SAP. This section looks at the work of 2 studies conducted on metaheuristics.

#### **2.3.2a – Simulated Annealing Heuristic**

Wipro Technologies published a paper (Gate Assignment Solution (GAM) Using Hybrid Heuristics Algorithm, 2009) where the flight GAP was defined as a job-shop scheduling problem. A hybrid algorithm based on a Simulated Annealing Heuristic, accompanied by Greedy and Tabu Search Heuristics, was used to solve the GAP. Some of the features of the final heuristic model, by Wipro Technologies, were different from other metaheuristic approaches. These key features are listed below.

1. The model was a simple transportation model that took less time to solve an was more appropriate for the real-life problem.
2. The primary objective increased the customer satisfaction while the 2 secondary objectives reduced the number of flights assigned to remote gates – hence increasing the airport efficiency.
3. The model follows a “first come, first assigned” policy that is practised by all airports. Most other literature ignored this policy, but it helped simplify the model.
4. The model accounted for the uncertainty in the flight arrival and departure times so that the need for forecasting data elements, identified by high uncertainty, was not necessary.

The primary objective of the model was to find a feasible flight to gate assignment that minimised the total walking distance for the passengers. The parameters used for formulating the final metaheuristic model were as follows:

* The number of flights for each fleet type
* The number of gates available (categorised by the airline they are leased to and the fleet types)
* Arrival and Departure times of the flights
* Preceding or Next flight of a connecting flight
* Connection time for passengers
* Number of passengers taking each connecting flight
* Boarding times
* The required Buffer times before each arrival and after each departure

As the first step to formulating the model, the Greedy Heuristic and the Tabu Search Heuristic were defined. The Greedy Heuristic was used to find the initial feasible solution that minimised the number of flights assigned to the *Apron* (i.e. the remote gates). All the flights were sorted in ascending order of *Departure Time.* The sorted flights were then assigned to the last available gate (i.e. the gate with the latest departure time. If no gates were available, the flights were assigned to the apron. The steps of the greedy heuristic used are described below.

**The Greedy Heuristic**

1. **Sort** flights by departure time. Let represent the earliest available time (= the departure time of the last flight) of gate . **Set** for all .
2. **For** each flight ,
   1. Find gate where and is maximised
      1. **If** such a gate exists, assign flight to
      2. **Else**, assign flight to the apron

Once the initial feasible flight-gate assignment was found, the Tabu Search Heuristic was used to improve it. The stages of the tabu search heuristic were as follows:

**The Tabu Search Heuristic**

1. = the current assignment

= the current objective value

**Set** and and the iteration,

1. **If** , **Terminate** at solution:

**Else**, **Go** to step 3

1. Establish a neighbourhood search method with uniform probability.

**Then** generate a neighbourhood as

= a single candidate solution

= some evaluation function

**For** each , **Calculate** :

**If** :

**Accept** and **Set**

**Else** :

**Select**

1. = a random number generated between and

**Update** the Tabu memory by setting:

**Set** :

**If** :

**Set** , hence

**Go** to step 2

The 2 algorithms described above were used in combination with a Simulate Annealing Heuristic to create the final model. The final *‘Master Algorithm’* is defined below.

**The Master Algorithm (Simulated Annealing Heuristic)**

1. = initial feasible assignment

= the objective value from the Greedy Heuristic

**Set** and

1. = the Annealing Temperature (a linear function to the input size)

= the starting temperature

= total number of aircraft to be assigned at the airport gates

**Set**

1. = uniform probability of neighbourhood search method

and = constants that decide on acceptance of neighbourhood move

= random number in interval [0, 1].

**If** the neighbourhood move is made:

**Calculate** the distance change:

**Calculate**

**If**  and :

**Set**  and

Record variables that are or

1. = rate by which is reheated

= rate by which is cooled

**If**  or :

**Run Tabu Search Heuristic**

Reheat the temperature:

**Else:**

Decrease the temperate:

1. **If** termination requirement id not met, **Go** to step 3.

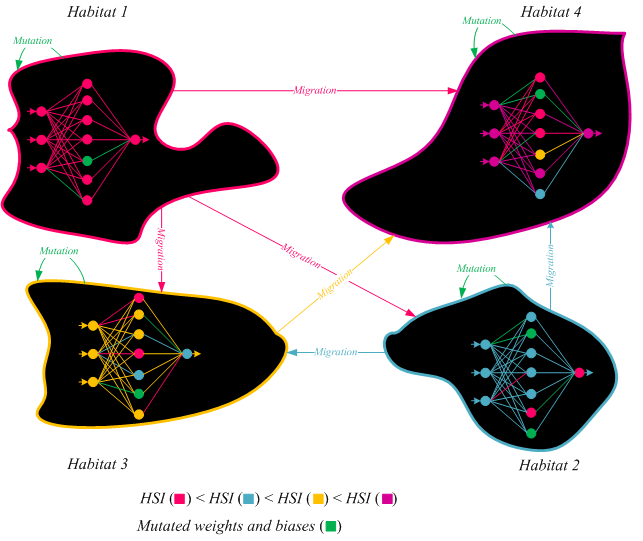
#### **2.3.2b – Bee-Colony and Biogeography-based Optimisation**

Another metaheuristic studied was the combination of a Bee-Colony Optimisation (BCO) Heuristic and a Biogeography-based Optimisation (BBO) Heuristic (Marinelli et al., 2015). Once again, this was a study conducted on a GAP with the 2 main objectives: (1) minimise the total walking distance of the passengers and (2) minimise the number of flights assigned to the apron. The model in this study considered the following constraints:

* Small aircraft could be assigned to large gates, but large aircraft could not be assigned to small gates.
* Every flight must be assigned to one gate, including the apron.
* Ensure that the Arrival and Departure times, at any of the gates, were not overlapping.

BBO is a population based evolutionary algorithm. This heuristic is based on the concept seen in ecosystems where nature tends to improve the stability of different habitats, dynamically. The BBO heuristic starts with a random set of habitats – each habitat has a Habitat Sustainability Index (HSI) and Emigration, Immigration, and Mutation Rates. The features of the BBO heuristic are listed below.

The figure 2.3.2b(i) (Uk.mathworks.com, 2018) depicts the format of the BBO heuristic.



*Figure 2.3.2b(i): The diagram that visualises the BBO heuristic (Uk.mathworks.com, 2018)*

* Habitats with a high HSI have a large number of Species, therefore are thought to have a high emigration rate and low immigration rate.
* If a species migrates to a habitat with a high HSI, they tend to die due to too much competition for resources.
* The maximum immigration rate occurs when there are no species in the habitat. As the number of species in a habitat increase, so does its HSI. When the number of species in a habitat reaches a maximum, the maximum emigration rate occurs.
* Given that: = the candidate solution

= immigration rate

= emigration rate

= the independent variable in the candidate solution

= maximum emigration rate

= maximum immigration rate

= number of individuals in each habitat

The immigration and emigration rates for are calculated by:

and

* Once HSI, and are calculated for each habitat, the habitats are modified based on the values of HSI, and . Some of the habitants are mutated in random habitats.
* The new set of improved habitats are then compared with the objectives and constraints. This is repeated until the nest solution is reached.

BCO is another heuristic that uses a concept based on nature. Social insect colonies are able to create systems that perform highly complex tasks by progressively interacting with each other. The BCO has workers called *‘Artificial Bees’* that cooperate to solve sophisticated, multi-objective optimisation problems. The BCO heuristic starts with all the artificial bees in a *‘Hive’.* They communicate to make a sequence of moves in a way that constructs a feasible partial solution, in increments. During these local moves, the bees make forward steps based on experiences from the past. Each forwards step is followed by a backwards step that brings the bees back to the hive. At the hive the bees partake in a decision-making process – this helps decide the bees next move. BCO is an iterative process that produces a partial solution at each iteration; the steps of this procedure are described below.

**Bee-Colony Optimisation Heuristic**

1. **Initialisation**

= number of bees

= number of iterations

= set of stages =

**Find** any feasible solution and **Set** it as the initial solution.

1. **Set** ; **Repeat** the rest of the stages till :
2. **Set** ; **Repeat** the rest of the stages till :

**Forwards Step:**

* + 1. The bees travel from the hive.
    2. Select B partial solutions from a set of partial solutions at stage

**Backwards Step:**

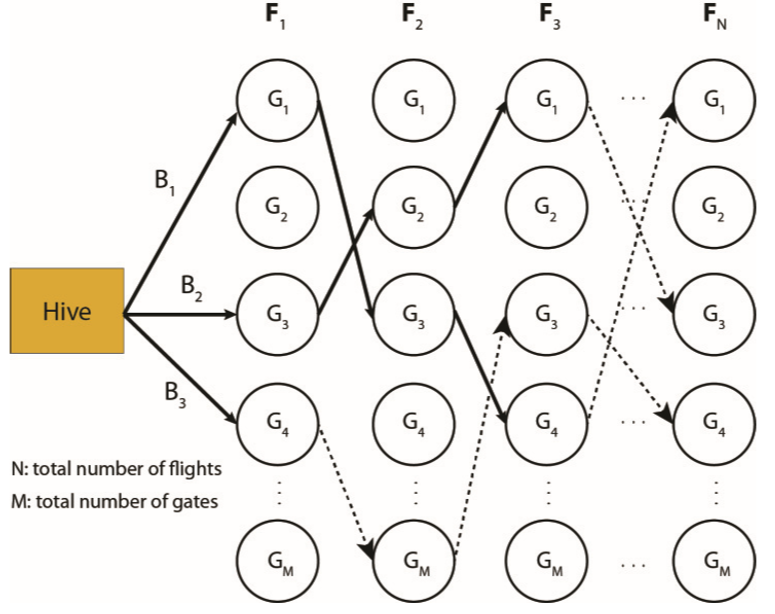
1. Send all the bees back the hive
2. Allow bees to exchange information about the quality of the partial solution. **Decide** between**:**

* **Abandon** the partial solution
* **Expand** the partial solution **without** other bees
* **Expand** the partial solution **with** other bees

1. **Set**
2. **At**  iteration: **If**  = the best solution; **update**
3. **Set**

The general setup of the BCO heuristic is shown in figure 2.3.2b(ii)

.



*Figure 2.3.2b(ii): The diagram that visualises the BCO heuristic (Marinelli et al., 2015).*

The final model used in this study, to solve the GAP, was a combination of these 2 heuristics called Biogeography-based Bee Colony Optimisation (B-BCO). There, the decision space was represented as an artificial network where the B-BCO was used to find an optimal path through the network. The features of the B-BCO heuristic were as follows:

**Biogeography-based Bee Colony Optimisation Heuristic**

* The network is made of a collection of habitats and each habitat consists of different layers.
* These layers signify the set of flights temporarily allocated in the arrangement of a given solution.
* = a flight at iteration

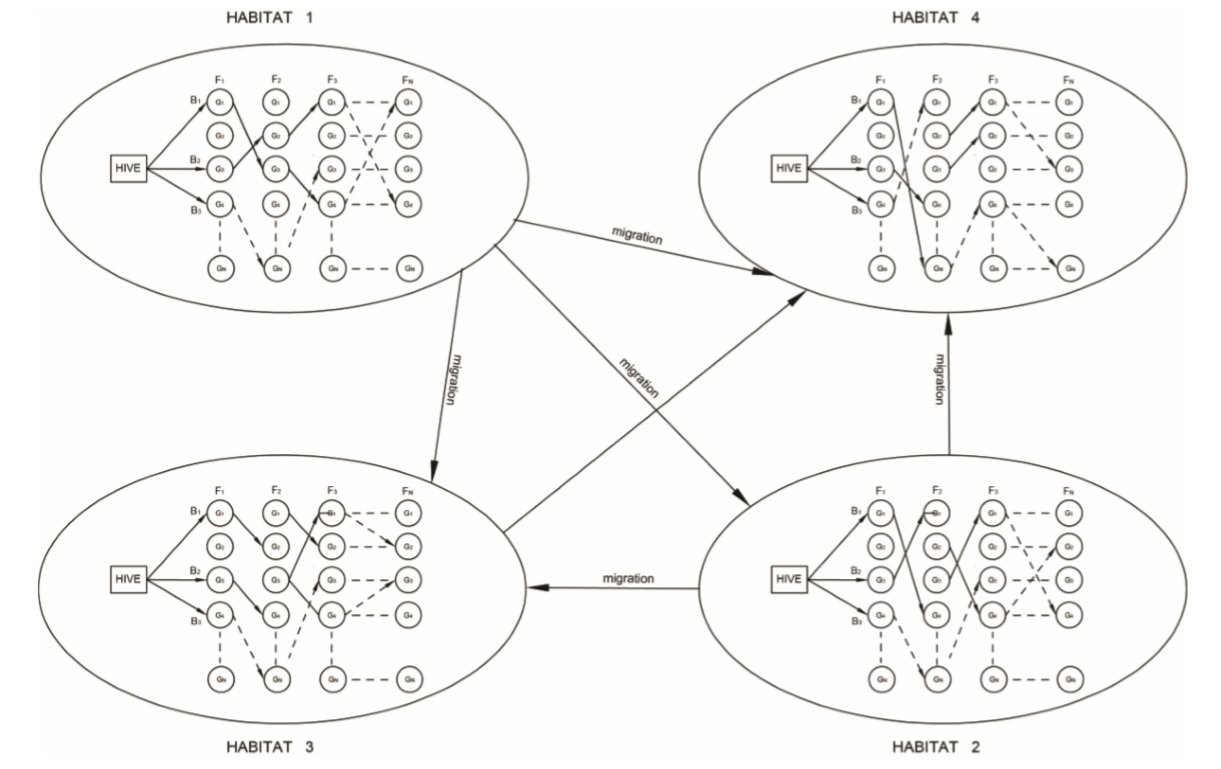
= a gate at layer

= a decision variable for the GAP

Each node on the network represents a link of compatibility between an and an available

* Each path on the network resembles the allocation of a particular to a specific , by a bee in the colony.
* At each iteration, all the solutions found are compared against the objectives and constraints of the GAP and the best solution is saved. This meant that each new iteration as looking for a new set of solutions.

The premise of the B-BCO was to conduct a B-BCO procedure within each habitat in the artificial network. Then once the partial solutions are found, use the BBO heuristic to move the bees from one habitat, along optimal routes, to another habitat. The structure of the B-BCO can be seen in figure 2.3.2b(iii).



*Figure 2.3.2b(iii): The diagram that visualises the B-BCO heuristic. The BCO within the BBO (Marinelli et al., 2015).*

## 2.4 Software for Flight Stand Allocation

Software used for the automatic scheduling of flights and flight operations are a very common sight. There are 100s of such scheduling systems that, not only focuses on scheduling but also provides a platform to manage and track the performance of an airport as an enterprise (Airlinesoftware.net, 2018). At a basic level, applications like *‘FlightNet’* and *‘AircraftLogs’* are web-based hence are easy-to-use and very easily accessible. In the aviation industry, softwares such as *‘CAMP’*, *‘IBS Software’* and *‘CAST’* are used. The industrial software is more expensive and not easily accessible but provide a wider range of services.

### *2.4.1 IBS Software for Airline Operations*

IBS is a company that owns the rights to several software that help manage various aspects of the aviation industry (Ibsplc.com, 2018). These areas include *airline passenger services*, *airline & airport operations*, and *airline cargo & logistics*. IBS also provides for some non-aero related industries like oil & gas energy, cruises and hotels.

IBS has a software called ‘*iAirport’* – an integrated information management platform. It helps airports to enhance their operational efficiency by providing a customised passenger experience. This is achieved by using passenger interactive technologies to efficiently monitor and manage the passenger feedbacks. This software also helps airports improve the quality of their services.

Within iAirport there are 4 applications that provide solutions to a range of different ecosystems within the airport. These 4 applications are:

1. “iAirport Operations”
2. “Passenger Relationship”
3. “Departure Control Systems”
4. “Airport Cargo”

The solving of the SAP is conducted within the “iAirport Operations” suite. This application is used to plan, manage and track flights and the allocation and usage of airport resources. The airport resources are managed in real-time as it allows more flights to operate with shorter turn-around times. The suite also has other features that help airports perform situational analysis and make strategic business decisions: a centralised database, billing management, an accurate projection of flight turnarounds and estimated times, etc.

The IBS’ iAirport software is used by over 170 customers. Some of the key airlines that use this software include British Airways, Emirates and Lufthansa. Heathrow and Gatwick airports use this as their airport management software. Since iAirport provides facilities to manage cargo, companies like American Airlines Cargo are also seen consuming this product.

### *2.4.2 CAST Stand Allocation*

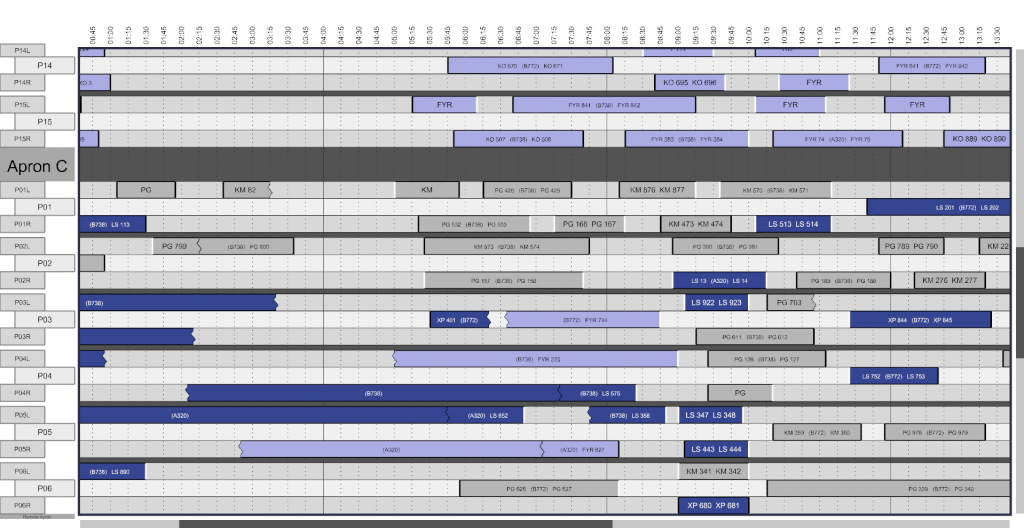
Airport Research Centre GmbH (ARC) is an independent, global company, based in Germany, that provides professional airport and aviation consulting. ARC developed and distributes an airport simulation software called *‘CAST’* (ARC, 2018). Like ARC, another airport consulting firm that employs CAST is the airport planning department of Arup.

CAST is a world leading 3D simulation with systems that simulate, plan and optimise. It is able to solve problems related to pedestrians, vehicles and aircraft traffic. It is also able to produce models of the landside, terminals, airside and the airspace. On top of producing solutions to create an optimised, efficient airport system, CAST allows users to communicate with experts, decision makers and stakeholders. This helps make quick and reliable decisions. CAST was designed to be industry-specific – this was done with the cooperation of Heathrow (London), Fraport, Zurich Airport, Eurocontrol, Charles de Gaulle (Paris), Madrid-Barajas Airport and Dubai Airport.

The 3 modules within the CAST software are: “CAST Simulation”, “CAST Express” and “CAST Allocation”.

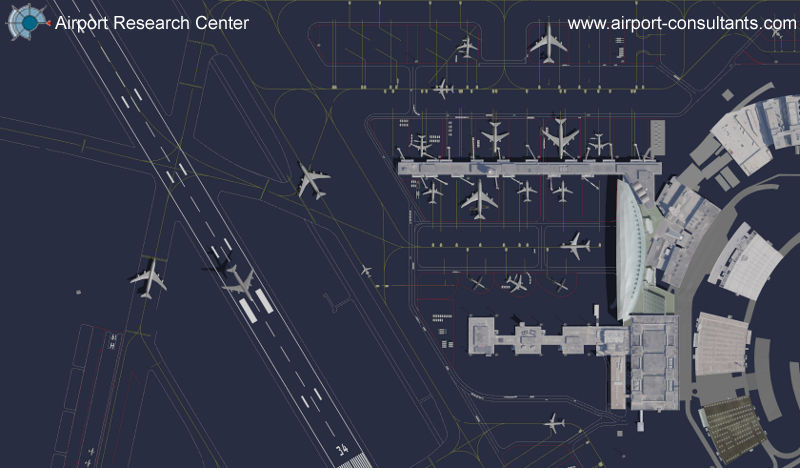
* **CAST Simulation** – simulates aircraft traffic, terminal passenger flow, ground handling traffic and curb-side/parking traffic.
* **CAST Express** – uses a high-speed simulation technology to make real-time predictions on multiple scenarios.
* **CAST Allocation** – software used for the allocation of airport resources.

Unlike most other flight scheduling software, CAST Allocation is dedicated only to solving SAPs and GAPs. This means using it as a stand-alone software is a lot more cost and time efficient. CAST Allocation models the structure of the apron of an airport, then uses manual inputs about flights to automate an allocation schedule to stands and gates. CAST Allocation has other features, like Gantt-charts and animations, that improve the experience of visualising and analysing different aspects of the SAPs and GAPs. The user interface of the software, with these features, is shown below – figure 2.4.2(i) shows the Gantt-chart view and figure 2.4.2(ii) shows the animation of the stand utilisation.



*Figure 2.4.2(i): The CAST user interface with the Gantt-chart view (ARC, 2018).*

*Figure 2.4.2(ii): The CAST user interface with the apron simulation view (ARC, 2018).*



# 3. METHODOLOGY

The following sections are a description of the SAP solved in this project. This chapter introduces the data and constraints used to build the SAP, then outlines and justifies the process of formulating the SAP. Finally, the coding model created to solve the SAP is defined.

## 3.1 Initial Flight Data

Arup is a multinational consultancy company for various engineering industries and built environment. The Airport Planning department at Arup – the sponsors of this project – work on many developments at the Heathrow Airport. Therefore, this project focused on creating an optimised solution for an SAP at Heathrow Airport.

All the initial data used to solve the SAP on this project were of Heathrow Airport, from the year 2017.

### 3.1.1 *Flight Schedule Data*

When solving any SAP, the most important pieces of information required are the arrival and departure times of the flights. The secondary requirements are information about the fleet types, destinations of the flights, etc. All these essential data are included in the flight schedule.

The initial flight schedule dataset obtained had details of flights at every terminal for the whole year of 2017. This was an account of 47,1072 flights, entering and leaving the Heathrow airport.

Table-A1 in Appendix I displays the first 5 and last 5 records of the flight schedule. The list below outlines all the data elements included in it.

* "**Scheduled.Date**" – Date when the flight was scheduled to take place.
* "**Scheduled.Time**" – The scheduled ‘*In-block’* time for arriving flights and the scheduled *‘off-block*’ time for departure flights.
* "**Stand.Date**" – The actual day when the flight arrived or departed at the stand.
* "**Stand.Time**" – The actual time when the flight arrived or departed at the stand.
* "**Actual.Date**" – The actual day when the flight was on the runway.
* "**Actual.Time**" – The actual time when the flight was on the runway.
* "**A.D**" – An indicator if the flight was an arrival or departure. A meant arrival and D meant departure.
* “**Flight.No**" – The flight number.
* "**Airline**" – The airline associated with the flight.
* "**Terminal**" – The terminal at which the flight was occurring.
* "**O.D**" – The origin/destination of the flight denoted by the 3-letter IATA code of the airport.
* "**Aircraft.General**" – The general aircraft type
* "**Aircraft.Specific**" – The specific aircraft type
* "**Seats**" – The number of seats on the plane.
* "**Total.Pax**" – The total number of passengers.
* "**Terminal.Pax**" – The number of terminal passengers.
* "**Direct.Pax**" – The number of direct passengers.
* "**Transfer.Pax**" – The number of passengers connecting through the airport onto another flight.
* "**Transit.Pax**" – The number of passengers who do not leave the plane but connect to another destination.
* "**Load.Factor**" – The percentage ratio of total passengers to seats.
* "**Transfer.Share**" – The percentage of transfer passengers on the flight.
* "**Flight.Type** – An indicator of the status of the flight. 1 is for scheduled flights and 0 for otherwise.
* "**Reg.No**" – The aircraft registration number. This is a unique identifier of the aircraft.
* "**ICAO.Code**" – A letter from *‘A’* to *‘F’* that indicates the size of the aircraft depending on its wingspan.
* "**Actual.Timestamp**” – The *Actual Date* + *Actual Time*
* "**Scheduled.Timestamp**" – The *Scheduled Date* + *Scheduled Time*
* "**Stand.Timestamp**" – The *Stand Date* + *Stand Time*

### 3.1.2 *Airport Details*

It is important to know where a flight is arriving from or departing to. This is because the SAP requires the flights to be differentiated between ‘*Domestic’* and *‘International’* and this affects the flights' compatibility with some stands.

As seen in section 3.1.1, the flight schedule data did not have enough information to identify if a flight was domestic or international. To deal with this issue, a second dataset was added to the initial data. This was a dataset that included details of the airports corresponding to the 11,987 unique IATA codes.

The data elements included in this dataset were as follows:

* The IATA code of the airport
* The ICAO code of the airport
* The complete and shortened names of the airport
* The city and country of the airport
* The longitude and latitude positions of the airport.

A sample of the airport details dataset can be seen in Table-A2 in Appendix I.

### 3.1.3 *Ethical Approval*

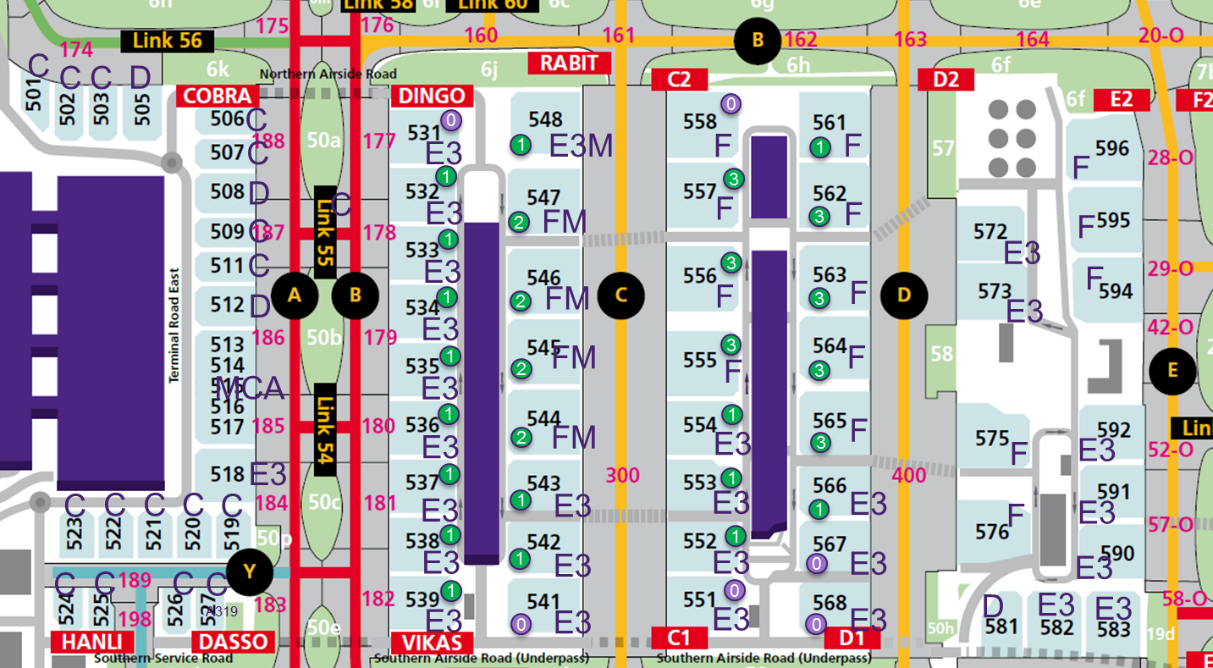
The CSV files of all the data used in the project were provided by Arup. These data are commercially sensitive hence a confidentiality agreement was signed to prevent sharing it. Consideration was given to ethics approval for this research and no application to ERGO was found necessary

## 3.2 Heathrow Airport – Terminal 5

The scope of this project was limited to Terminal 5 at Heathrow Airport. Here, the layout of the Terminal 5 apron is described along with the restrictions and assumptions on airport operations.

This information was a simplified version of the “Heathrow Stand Planning Assumptions (2017)” used in industry and was provided by Arup.

### 3.2.1 *Stand Layout*



**T5A**

**T5B0**

**T5C**

*Figure 3.2.1(i) shows a diagram of the layout of the stands at Terminal 5.*

Figure 3.2.1(i) shows a diagram of the layout of the stands at Terminal 5. Terminal 5 apron consisted of 3 sections: T5A, T5B and T5C. Each of these sections had restrictions on the types of aircraft that they could accommodate – these were based on the ICAO code of the aircraft.

The project was limited to only using flights with ICAO codes **C, D, E** and **F** were considered.

**T5A**

* All the stands in this section were code **C** and **D** stands and could accommodate *all aircraft in the code C and D fleets.*
* The stand labelled **MCA** could either be used to park ‘*1 D-code and 2 C-codes’* or ‘*2 E-code’* aircraft.

**T5B & T5C**

* The stands labelled E3 and EM were all code **E** stands and were able to accommodate *all aircraft in the code E fleet.*
* Thecode **F** stands could accommodate *all aircraft in the code F fleet*.
* All of the stands in these sections were large enough to be compatible with **C**-code and **D**-code aircraft.

**Remote Stands**

* The section with **570 – 581**, as seen on figure 3.2.1(i), were remote stands.
* All other stands were excluded from this project.

### 3.2.2 *Operational Assumptions*

Listed below are all the assumptions and restrictions that were applicable to the flights at Terminal 5.

**Domestic and CTA flights**

Flights considered as domestic were those arriving from and departing to, the United Kingdom. CTA flights were flights that were travelling to or from Dublin, Ireland. The service to these flights was limited to specific stands within Terminal 5:

* **Domestic** **arrivals** could only be served by stands **501-503** and **505-507**.
* **Domestic** **departures** were only compatible with the stands **501-503, 505-511, 519, 522** and **523**.
* **CTA arrivals** could only be parked at stand **523**.

**Contact and Remote Stands**

* Since the domestic flights and CTA arrival flights were restricted to be served by the stands mentioned above, an allocation to **any other stand** was *treated as a remote arrival/departure.*
* All other flights were treated as *pier-served flights.*
* However, there were some remote stands within sections T5A, T5B and T5C. **Any** departures or arrivals from these stands were *treated as remote:*

**T5A**: 524 – 527

**T5B**: 531, 541 and 548

**T5C**: 551, 558, 568 and 567

* The stands **572 – 581**, as mentioned before, were remote stands and *all flights occurring at stands were remote*.

**Buffer Times**

The buffer times used were derived from the British Airways (BA) rules in 2017. There were 2 sets of buffer times for the smaller aircraft and the larger aircraft.

* **C-code** and **D-code** aircraft:
  + **25 minutes** between flights
* **E-code** and **F-code** aircraft:
  + **30 minutes** between flights

**Towing Rules and Considerations**

Similar to the buffer times, the rules for towing operations were also dependent on the size of the aircraft. Listed below are the durations for enplaning/deplaning before and after towing the aircraft.

* **C-code** and **D-code** aircraft:
  + **Only overnight** flights – i.e. the aircraft that were parked at the airport through the night – were towed.
  + C-code: **45 minutes** before tow-off | **45 minutes** after tow-back.
  + D-code: **60 minutes** before tow-off | **60 minutes** after tow-back.
* **E-code** and **F-code** aircraft:
  + Flights with turnarounds greater than **8 hours** were towed off.
  + E-code: **90 minutes** before tow-off | **100 minutes** after tow-back.
  + D-code: **100 minutes** before tow-off | **120 minutes** after tow-back.
* The *time taken for towing* was assumed to be **0 minutes.**

### 3.2.3 *Training Data*

The training data used to test the feasibility of the SAP model was the flight schedule from the date: **21st July 2017**.

## 3.3 Defining the Problem

There were 2 primary objectives for the project. The first was a goal for the data manipulation stage – to create a set of useful data for the SAP. The second was the objective for the SAP. These 2 objectives were as follows:

1. *Automate the processing of the flight schedule data to calculate the flight turnaround times using Python.*
2. *Define and implement a heuristic to solve an SAP that maximised the number of passengers passing through pier-served stands.*

The project also aimed to address these questions:

* Was the Python model, for the data manipulation, feasible when dealing with flight data for a year?
* Had all the flights been allocated to a stand?
* Were the allocated stands compatible with the aircraft?
* Had potential delays or early arrivals of flights been accounted for? (Buffer times)

## 3.4 Data Manipulation

Once all the initial data were received, it was important to clean the data and process them into a format that was useful for the SAP.

### 3.4.1 *Data Cleaning*

Data cleaning is the process of detecting any incomplete, inaccurate or irrelevant data, in a dataset, and removing or correcting them.

For this project, only the **scheduled flight times** were considered. This meant that the actual times when the flights were at the stand or runway were ignored. In addition to that, the initial flight schedule dataset had data elements with irrelevant or repeated information. Therefore, the data elements removed from the **flight schedule dataset** were:

* The actual times and dates when the flight was at the stand

(*'Stand.Date', 'Stand.Time', 'Stand.Timestamp’*)

* The actual times and dates when the flight was at the runway (*'Actual.Date', 'Actual.Time', 'Actual.Timestamp'*)
* The indicator of the flights’ status (*'Flight.Type'*) due to irrelevance.
* The repeated information about the scheduled flight times (*'Scheduled.Date', 'Scheduled.Time'*)

Once these data were cleaned, the information of when the flight arrivals and departures occurred was only given by the ***'Scheduled.Timestamp'****.*

As introduced in section 3.1.2, the **airport details dataset** included various pieces of information about the airports, however, not all of this information was needed to help solve the SAP. The important data elements extracted from this dataset were:

* The IATA code of the airport (*‘AIRPORT\_IATA\_CODE'*)
* The city and country of the airport

(*'AIRPORT\_CITY', 'AIRPORT\_COUNTRY’*)

Lastly, it was important the training data – the schedule for 21st July 2017 – was extracted from the dataset.

This produced the final versions of the data: (1) a daily flight schedule for the 21st and (2) the suitable airport details. These were then further processed to create input data for the SAP.

### 3.4.2 *Calculating Flight Turnarounds*

The turnaround of a flight refers to the total time an aircraft spends at the airport, between its arrival and departure. The equation for calculating the turnaround time of a flight is:

**Eq1:**

The final flight schedule dataset was a table that listed the arrival and departure flight details for each aircraft in separate rows. This dataset was used to create a “**Turnarounds**” table. Unlike the flight schedule, the “Turnarounds” table displayed the information of the arrival and departure flights, of each aircraft, in the same row.

A key feature of this table was that it accounted for the towing operations that needed to take place during the 21st of July. When the flight turnaround times were calculated, those with very large turnarounds were split. This splitting process is further described in section 2.1. The turnarounds table allowed the 3 towing operations (enplaning, waiting and deplaning) to be viewed separately. It had 2 empty columns that were used to record the arrival and departure stands for each aircraft. It also included information about the city and country of the arrivals/departures.

The turnarounds table is the most important dataset for airport planners. This is because this table holds all the data required for solving SAPs, or other airport procedures, in one place. This was also the case here as one of the main objectives of the project was to automate the creation of the flight turnarounds dataset.

### 3.4.3 *Modelling the Data Manipulation*

Cleaning the initial data and creating the turnarounds dataset were carried out through an automated model. This model was built using the programming language **Python**. The decision to use Python was made following the request from Arup. The Python module **“Pandas”** and **“NumPy”** were used to conduct the mathematical processing.

#### **3.4.3a – Model for Data Cleaning**

There were 2 functions in the Python model that dealt with the cleaning of the initial data.

The first was **“Load\_File(***Schedule\_file, Airport\_file***)”**. This function took 2 inputs – the file names of the flight schedule and airport details datasets – and read these files onto 2 *‘dataframes’* in Python. This was also the function that carried out the removing of irrelevant data and extracted the useful ones. It was followed by the execution of the **“Select\_Data()”** function. This was to select the training data.

The Python scripts of these 2 functions are shown in Section-A1 and Section-A2 of Appendix II.

#### **3.4.3b – Model for Flight Turnarounds**

To create the “Turnarounds” dataframe, the first step was to generate 2 datasets from the final flight schedule dataframe – one for arrival flights and the other for departures. The registration numbers were used to uniquely identify the aircraft. They were then used to find and match up details of the arrival and departure flights, for each aircraft. These data were recorded in the “Turnarounds” dataframe. The functions **“Sort\_cols()”** and **“Create\_Turns()”**, in the Python model, were responsible for this and are displayed in Section-B1 of Appendix II.

Next, the **“Turnarounds()”** function was executed in the model. This added 7 new columns to the turnarounds dataframe to include the origin city/country, destination city/country, turnaround times and the arrival/departure stands for each aircraft. The columns for the stands were left empty. The turnaround times were calculated using Eq1 and data from within the “Turnarounds” dataframe. To add the origin and destination data, the IATA codes of the arrival and departure airports were used as the unique identifier. These were matched with the IATA codes in the airport details dataframe and the details of the cities and countries were mapped onto the “Turnarounds” dataframe.

**“Turnarounds()”** function also dealt with formatting the data into a more readable initial table. Therefore, the Python script in Section-B2 in Appendix II, shows the snippet of the function where new data are calculated and added.

**Additional Data Cleaning**

Since the data of the day, 21st July, the arrival/departure times of the overnight flights were blank. The was because these occurred during the previous or next day.

The **“Splitting()”** function performed additional data cleaning to replace these missing data with the time ‘*12am*'. This indicated that, at 12 am of 21st July, the overnight flights departing in the morning were at the Terminal 5 apron. It also indicated that the overnight flights that arrived on the 21st stayed on the apron till 12 am of 22nd July.

The main purpose of the **“Splitting()”** function was to conduct the splitting process on the flight turnarounds. To do this the initial turnarounds dataframe was separated into 5 parts:

1. Aircraft that arrived during the day/night and stayed overnight
2. Aircraft that parked overnight and departed in the morning
3. Aircraft in fleet F where
4. Aircraft in fleet E where
5. Aircraft that didn’t require towing.

Once the final data cleaning was completed, 2 duplicates of each row in the first 4 dataframes were created so that each aircraft had 3 rows that separately represented the 3 towing operations. The arrival and departure times on these rows were then recalculated using the enplaning and deplaning times stated in part 3.2.2. The waiting operations were labelled with an **“R”** on their arrival stand column – this was to ensure that these were definitely allocated to remote stands.

The **“Splitting()”** function uses a combination of ‘*for-loops’* and *‘if-statements’* to perform the splitting process. A sample of code for one set of overnight flights and for fleet F flights are displayed in Section-B3 of Appendix II. The structure of the code was similar for feet E data. The last step was to execute the **“final\_Turnarounds()”** function which merged the 5 parts to create the final “Turnarounds” table. The Python script for this is in Section-B4 of Appendix II and a part of the “Turnarounds” table is shown in Table-A3 in Appendix I.

## 3.5 Formulating the SAP

Discussed below is the process of identifying the variables of the SAP and deriving the heuristic to solve it.

### 3.5.1 *Identifying Variables*

The variables used in the formulation of the SAP were based on the 2nd primary objective stated in section 3.3 – maximise the number of passengers passing through pier-served stands. There was also an aim to ensure that flights were allocated to compatible stands only.

Since the objectives were strongly focused on the size/type and compatibility of the stands, the first step was to use the concept partitioning to create sets of feasible stands for aircraft in each fleet type. The partitioning of the stands was conducted using the restrictions and assumptions described in section 3.2. The stand layout was already segmented (T5A, T5B and T5C) and the operational assumptions suggested that majority of the domestic flights were to be allocated to T5A while the international flights to T5B and T5C. As for the MCA stand in T5A, it was assumed to always hold ‘*1 D-code (513) and 2 C-codes (515, 517)*’ aircraft.

The derived sets of stands for each fleet type are listed below.

***Remote Stands***

= {524, 525, 526, 527}

= {581}

= {531, 541, 548, 551, 567, 568, 572, 573}

= {558, 575, 576}

***Pier-served Stands***

= {501, 502, 503, 506, 507, 509, 511, 519, 522}

= {523}

= {515, 517, 520, 521}

= {505, 508}

= {512, 513, 505, 508}

= {518, 532, 533, 534, 535, 536, 537, 538, 539, 542, 543, 552, 553,

554, 566}

= {555, 556, 557, 561, 562, 563, 564, 565, 544, 545, 546, 547}

The final list of variables used in the SAP was:

= Set of Flights in a day

= Sets of compatible Stands

= Set of Buffer times for fleets

= List of Domestic Flights in a day

= List of International Flights in a day

= List of flights for remote stands

= List of flights for pier-served stands.

= Set of flight Arrival times

= Set of flight Departure times

= Set of Total Passengers in each flight

### 3.5.2 *The Stand Allocation Heuristic*

The approach of the *Stand Allocation* heuristic was a combination of concepts from the Greedy heuristic and the BLS heuristic.

It allocated flights following the “first come, first assigned” policy hence the flights with the earliest arrival times were chosen greedily to be allocated first. When allocating flights, if more than one compatible pier-served stand was available, the heuristic chose to allocate to the stand that had served the highest number of passengers.

Once the *Stand Allocation* heuristic produced an initial schedule, it was checked to see if the flights in each stand adhered to the buffer times restriction. If any overlapping existed, these flights were removed from the schedule and reallocated to other feasible stands. This technique was adopted from the BLS heuristic; as discussed in chapter 2.

The procedure of the *Stand Allocation* heuristic segregated the flight data by fleet type (F, E, D, C) and made allocations one fleet at a time. This was because the compatibility of stands depended strongly on the size of the aircraft. The *Stand Allocation* heuristic was applied to flights in decreasing order of aircraft size - the largest aircraft (F) first and the smallest (C) last. This decision was made because stands that could facilitate larger aircraft were big enough to hold smaller aircraft but not vice versa. Therefore, when the heuristic was making reallocations, flights of smaller aircraft that didn’t have room in their own stands could be parked at larger stands.

In industry, airport planners create the stand allocation schedules on a daily basis due to the high probability of changes in the flight schedule. For this reason, the *Stand Allocation* heuristic was also designed to work with daily flight data.

The detailed steps of the *Stand Allocation* heuristic, for assigning flights in each fleet type, are described below.

The procedure for the flights of F-code aircraft.

From , filter the flights of F-code aircraft.

**For F-code aircraft:**

* **Get** the list of flights for remote stands
* **Get** the list of flight for pier-served stands

**Allocate** the 1st flight in to 1st stand in

**For** the rest of flights in :

* **Sort** the stands in in descending order of the total passengers served
* **For** with the largest , **If** the is **true:**

( sees if is free for 30minutes before the arrival of flight . i.e. )

* **Allocate**  to
* **Else: go to** next
* Repeat until a feasible is found and is allocated
* If no feasible is found, leave unallocated

**Then**

**Allocate** the 1st flight in to 1st stand in

**For** the rest of the flights in :

* **Sort** the stands in in descending order of
* **For**  with the largest , **If** the is **true:**
* **Allocate**  to
* **Else: go to** next
* Repeat until a feasible is found and is allocated
* If no feasible is found, leave unallocated

The procedure for the flights of E-code aircraft.

From , filter the flights of E-code aircraft.

**For E-code aircraft:**

* **Get** the lists and

**Allocate** the 1st flight in to 1st stand in

**For** the rest of flights in :

* **Sort** the stands in in descending order of the total passengers served
* **For** with the largest , **If** the is **true:**

( sees if is free for 30minutes before the arrival of flight . i.e. )

* **Allocate**  to
* **Else: go to** next
* Repeat until a feasible is found and is allocated
* **If** no feasible is found, leave **unallocated**

**Detect** unallocated in :

* **Reallocate** to stands in

**Then**

**Allocate** the 1st flight in to 1st stand in

**For** the rest of flights in :

* **Sort** the stands in in descending order of
* **For**  with the largest , **If** the is **true:**
* **Allocate**  to
* **Else: go to** next
* Repeat until a feasible is found and 𝑒 is allocated
* **If** no feasible is found, leave **unallocated**

**Detect** unallocated in :

* **Reallocate** to:
  + Stands in
  + Stands in
  + Stands in

The procedure for the flights of D-code aircraft.

From , filter the flights of D-code aircraft.

**For D-code aircraft:**

* **Get** the lists and

1. **Allocate** the 1st flight in to 1st stand in

**For** the rest of flights in :

* **Sort** the stands in in descending order of the total passengers served
* **For** with the largest , **If** the is **true:**

( sees if is free for 30minutes before the arrival of flight . i.e. )

* **Allocate**  to
* **Else: go to** next
* Repeat until a feasible is found and is allocated
* **If** no feasible is found, leave **unallocated**

1. **Detect** unallocated and **remove** infeasibly allocated in :

* **Reallocate** those to:
  + Stands in
  + Stands in

**Then**

1. **For**  in

**Follow** the allocation process in step 1 to:

* **Allocate** domestic to in
* **Allocate** international to in

1. **Detect** unallocated and **remove** infeasibly allocated in :

* **Reallocate** those to:
  + Stands in
  + Stands in
  + Stands in

The procedure for the flights of C-code aircraft.

From , filter the flights of C-code aircraft.

**For C-code aircraft:**

* **Get** the lists and

1. **Allocate** the 1st flight in to 1st stand in

**For** the rest of flights in :

* **Sort** the stands in in descending order of the total passengers served
* **For** with the largest , **If** the is **true:**

( sees if is free for 30minutes before the arrival of flight . i.e. )

* **Allocate**  to
* **Else: go to** next
* Repeat until a feasible is found and is allocated
* **If** no feasible is found, leave **unallocated**

1. **Detect** unallocated and **remove** infeasibly allocated in :

* **Reallocate** those to:
  + Stand in all pier-served C-code stands
  + Stand in
  + Stand in

**Then**

1. **For**  in

**Follow** the allocation process in step 1 to:

* **Allocate** CTA arrival to in
* **Allocate** domestic to in
* **Allocate** international c to in

1. **Detect** unallocated and **remove** infeasibly allocated in :

* **Reallocate** those to:
  + Stand in
  + Stand in
  + Stand in
  + Stand in

To get the stand allocation schedule for the flights of a whole day, the *Stand Allocation* heuristic performed all the fleet-specific procedures. For the monthly schedule was achieved by creating the daily schedule, on loop, for every day of the month.

The details of these procedures were as follows:

**The Daily Stand Allocation Heuristic**

**For** the daily flight turnarounds:

1. Perform the allocation procedure for **F-code** aircraft
2. Perform the allocation procedure for **E-code** aircraft
3. Perform the allocation procedure for **D-code** aircraft
4. Perform the allocation procedure for **C-code** aircraft

**The Monthly Stand Allocation Heuristic**

**For each day** of the monthly flight schedule:

* **Create** the daily flight turnarounds
* **Perform** the Daily Stand Allocation heuristic

### 3.5.3 *Modelling the Stand Allocation Heuristic*

The model of the Stand Allocation Heuristic was created using Python. The programming language, Python, was used, again, following the request made by Arup. The created model had 5 main classes – 4 classes that had functions to allocate flights from each fleet type and 1 class that had functions to create the final daily or monthly stand allocation schedules.

**Class 1: F-Code Flights**

The flights using F-code aircraft were allocated to stands by the function **Schedule\_F.** This function ran these 3 sub-functions:

1. **F\_fl** – Responsible for extracting flight turnarounds data for F-code aircraft and defining the sets of stands compatible with the code F fleet.
2. **Remote\_F** – All the overnight parking was allocated to the remote F-stands, first. This ensured that the pier-served stands were free for the daily stand operations.
3. **Pier\_F** – Assigned the rest of the arrival/departure stand operations to the pier-served F-stands. Any of the operations that didn’t fit the schedule, were assigned to remote F-stands.

The details of the annotated Python scripts, for these functions, are given in Section-C1 of Appendix II.

**Class 2: E-Code Flights**

**Schedule\_E** was the function for allocating all the E-code aircraft and it ran the following sub-functions:

1. **E\_fl** – Responsible for extracting flight turnarounds data for E-code aircraft and defining the sets of stands compatible with the code E fleet.
2. **allo\_checker** – This function checked if there were any infeasible allocations being made, and if there were, those allocations were removed from the schedule. This function was used for the allocation of both E- and D- code flights.
3. **Remote\_E** – All the overnight parking was allocated to the remote E-stands to that the pier-served E-stands were free for the daily stand operations.
4. **Remote\_Eleft** – Checks for overnight parking that had not been allocated to the remote stands and assigns them to a pier-served E- or F-code stand.
5. **Pier\_F** – Assigned the rest of the arrival/departure stand operations to the pier-served E-stands.
6. **E\_to\_FPier** & **E\_to\_ERemote** & **E\_to\_FRemote** – Uses the allo\_checker function to find infeasible allocations at the pier-served E-stands and any stand operations that were not assigned any stand. Then reassigns them to any feasible E- or F-code stands.

The annotated Python scripts for fleet E allocation functions are given in Section-C2 of Appendix II.

**Class 3: D-Code Flights**

**Schedule\_D** assigned all the D-code aircraft to feasible D-, E- and F-code stands. This was achieved by the following sub-functions:

1. **D\_fl** – Responsible for extracting flight turnarounds data for D-code aircraft and defining the sets of stands compatible with the code D fleet.
2. **Remote\_D** – All the overnight parking was allocated to the remote D-stands.
3. **RemoteD\_to\_FPier** & **RemoteD\_to\_EPier** – Checks for overnight parking that had not been allocated to the remote stands and assigns them to a pier-served E- and F-code stand.
4. **UKArr\_D** – The stand operations arriving from a domestic origin were assigned to only stands accommodating the parking of domestic arrival flights.
5. **UKDep\_D** – The stand operations departing to domestic destinations were assigned to the stands that were able to accommodate the domestic departures.
6. **Int\_D** – Assigned all the international flights to any of the pier-served D-code stands. (except those restricting international arrivals)
7. **PierD\_to\_EPier** & **PierD\_to\_FRemote** – Uses the allo\_checker to find infeasible or incomplete allocations made to any pier-served D-stand. They were removed and reassigned to any available, feasible E- and F- code stands.

Section-C3 of Appendix II shows all the annotated Python scripts for these functions.

**Class 4: C-Code Flights**

**Schedule\_C** was the main function thatassigned all the C-code aircraft to feasible C-, D-, E- and F-code stands. This was achieved by the following sub-functions:

1. **C\_fl** – Extracted the flight turnarounds data for C-code aircraft and defined the sets of stands compatible with the code C fleet.
2. **C\_allo\_checker** – Like the ‘allo\_checker’ function, this checks for any infeasible or incomplete (failed) allocations that were made to any of the C-code stands. A checking function was made specifically for the code C fleet because it had the greatest number of flights and was compatible with stands of all sizes.
3. **Remote\_C** – All the overnight parking was allocated to both remote and pier-served C-code stands.
4. **RemoteC\_to\_FPier** & **RemoteC\_to\_EPier** – Identifies all the overnight parking that had not been allocated, or incorrectly allocated, to C-code stands. Then assigns them to a pier-served E- and F-code stand.
5. **CTA\_C** – The flights arriving from Dublin were restricted to only one pier-served stand. This function allocated the CTA arrivals to that stand with priority before non-CTA arrival/departure stand operations.
6. **UKArr\_C** – The stand operations arriving from UK origins were assigned to only C-code stands accommodating the parking of domestic arrival flights. The ‘C\_allo\_checker’ function was then used on the allocations made. The infeasible/incomplete allocations were then allocated to both remote and pier-served F-, E- and D-code stands – all of these were considered remote.
7. **UKDep\_C** – Had the same functionality as ‘UKArr\_C’, except it was applied to the stand operations that were departing to a domestic destination.
8. **Int\_C** – Assigned all the international flights to any of the pier-served C-code stands. (except those restricting international arrivals). The ‘C\_allo\_checker’ function was applied to the allocations and the detected infeasible/incomplete allocations were reassigned to both remote and pier-served F-, E- and D-code stands but they were all

Section-C3 of Appendix II shows all the annotated Python scripts for these functions.

**Class 5: The Daily and Monthly Allocation Schedules**

There were 2 main functions that there defined in this class. The **Daily\_Sched** and **Monthly\_Sched** produced the daily and monthly allocation schedules for the flights, respectively.

The ‘Daily\_Sched’ function took the day of the date and a string input. It then ran the ‘Schedule\_F’, ‘Schedule\_E’, ‘Schedule\_D’ and ‘Schedule\_C’ functions, from the other classes, in this order. The largest aircraft were assigned first and smallest last. The completed flight turnarounds table, with the stand allocations, was then saved as a .CSV file.

The ‘Monthly\_Sched’ function took the numerical value of the month as an input and used it to extract the flight schedule data for that month. Then, the function looped over the dataset, to create a turnarounds table and apply the ‘Daily\_Sched’ function to every day of the month. The allocation schedules for each day of the month were then saved as .CSV files in a folder, labelled by the month.

The Python scripts for these 2 functions are shown in Section-D1 and Section-D2 in Appendix II.

The general code that is used for the allocations of each fleet type was the same. The parts that varied were the fleet specific constraints like the buffer times and the feasible sets of stands. Also, the specifics of which stands can accept reassigned flights – i.e. only small aircraft can be reassigned to larger stands, not vice versa. Described below are some of the key parts of the Python model that perform the allocation heuristic.

* Calculate the total number of passengers served at each used stand and sort the stands from largest to smallest total:

*#calculate the sum of all number of passengers served#*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True)

*# create set of stands sorted from largest to smallest no.of passengers*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending=True).reset\_index()

* The greedy select the stand starting with the one that had largest number of passengers served:

for (j, rows) in df1.iterrows(): *#for each stand is set df1*

stand = df1['Stand\_Arrive'][j]

* Ensure that the buffer times are added:

*#for C- and D- code aircraft*

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>=pd.to\_timedelta('25 minutes')

*#for E- and F- code aircraft*

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

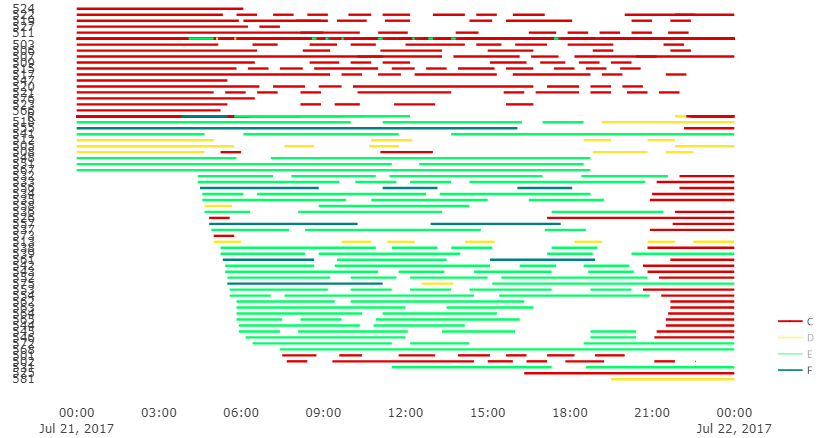
- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>=pd.to\_timedelta('30 minutes')

# 4. RESULTS

This chapter displays and explains the results obtained from applying the Stand Allocation Heuristic to the training data. Due to the commercial sensitivity of the flight data, the detailed numerical result tables have not been displayed.

## 4.1 Stand Allocation Gantt Chart



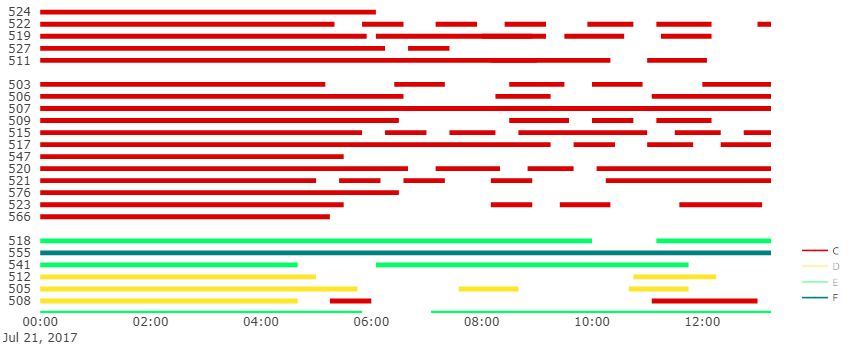
*Figure 4.1(i): The Stand Allocation Gantt chart for all the flights on 21st July 2017*

The Stand Allocations Schedule for 21st July 2017

Figure 4.1(i), above, shows the Gantt chart for all the scheduled flights for the whole day of 21st July. This was generated through Python and was colour coded based on the fleet type. C-code flights were red, D-code flights were yellow, E-code flights were green and the F-codes were blue.

The stand allocation heuristic used a partitioning concept to choose the feasible stands for the different flights. The impact of this could be seen from the Gantt chart had blocks of colour at different time periods.

The through the Gantt chart it was also seen that the number of E-code and C-code flights are significantly higher than others.



Stands

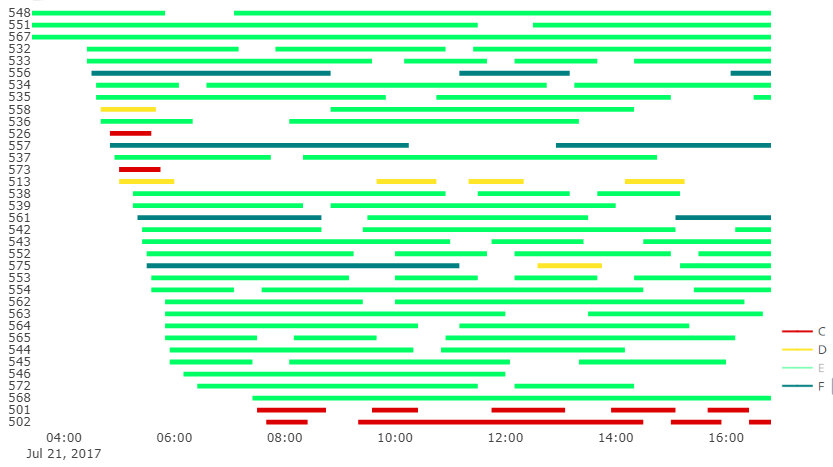
The Stand Allocations Schedule for 21st July 2017

*Figure 4.1(ii): Section of C-code & D-code flight Stand Allocation Gantt chart (12 am to 12 pm on 21st July 2017)*

Stands

The Stand Allocations Schedule for 21st July 2017

*Figure 4.1(iii): Section of E-code & F-code flight Stand Allocation Gantt chart (4 am to 4 pm on 21st July 2017)*



The Gantt chart had several empty, "unallocated", spaces. A closer look at these is shown in figures 4.1(ii) and 4.1(iii). The C and D-code flights showed to be allocated without any overlaps, but these stands were only able to hold small aircraft. And the empty spaces from figure 4.1(iii) were at remote stands where all flights types were treated as remote (e.g. stands 573 and 526).



Table 4.1(A) is a table of the summary of the flight data for 21st July. The number of flights referred to the journeys from origin to destination that occurred, and the number of stand operations was the turnarounds after the splitting process. The total number of passengers was the sum of all the passenger enplaning and deplaning, from all arrival and departure flights.

The number of stand operations for C-code flights was the highest at **363** of the 500 operations**,** followed by E-code flights at **99.** D and F-code flights had the least stand operations, at **27** and **11**, respectively.

## 4.2 Breakdown of Allocations for each Fleet type

The restrictions on the compatibility of remote and pier-served stands depended on the fleet type as well as the origin/destination. Therefore, the results were presented for each fleet type separately.

The number of flights and passengers that were not allocated to a stand were analysed.

According to Table 4.1(A), there were a total of 375 **arrival/departure stand operations** and the results from the flights that arrived and departed on the same day showed that **205** of those operations were not allocated. All the aircraft fleets had failed allocations but the majority of them were of C-code aircraft – this was expected since more than half of the occurred flights were from C-code aircraft. In the 205 failed allocations, there were **40,565 passengers.**

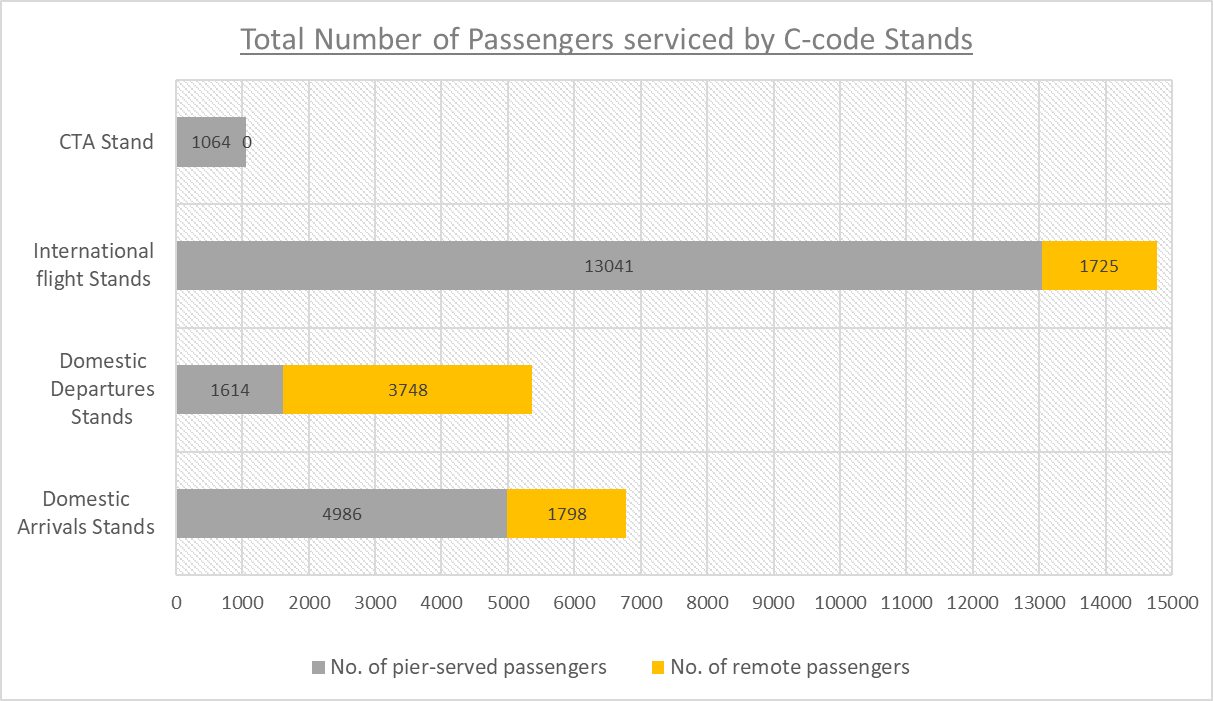
For the flights that were parked overnight before a departure or after an arrival, the results showed that only **33** of the 125 **overnight parking operations** were not allocated to stands. 30 of these were for C-code aircraft and, in the case of E and F-code flights, all overnight parking operations were allocated. The overnight parking operations did not have any passengers – no people are being served overnight. These results – for both arrival/departure and overnight parking operations have been summarised in Table 4.2(A).

The flight results showed that the **rate of allocation** of the heuristic was **52.4%.** The results also indicated that **61% of passengers**, passing through Terminal 5 in a day, were in flights served by a Terminal 5 stand.



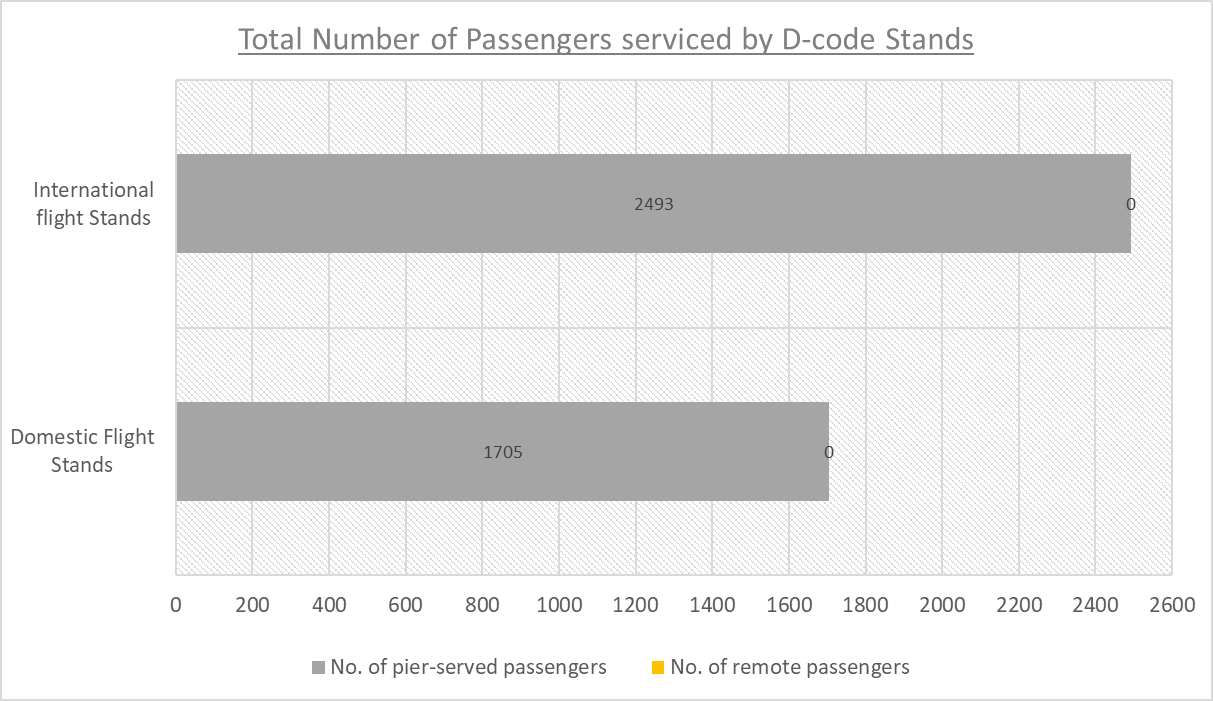
There was a total of **63,538** passengers in the flights that were allocated to Terminal 5 stands. The C-code stands served a total of **27,976** passengers; D-code stands served **4198** passengers; E-code stands served **18,276** passengers and the F-code stands served **13,088** passengers.

The pier-served C-code stands were restricted by 4 conditions: Domestic Arrivals, Domestic Departures, International flights and CTA Arrivals. Any flights that were allocated to these stands that didn’t adhere to the conditions were remote (e.g. if an international arrival was serviced by a domestic arrival stand – it was remote). This resulted in **20,705 pier-served passengers** and **7271 remote passengers** at the C-code stands. A breakdown of these numbers at each set of stands are shown in Figure 4.2(i). Only overnight parking operations were allocated to the remote C-code stands.



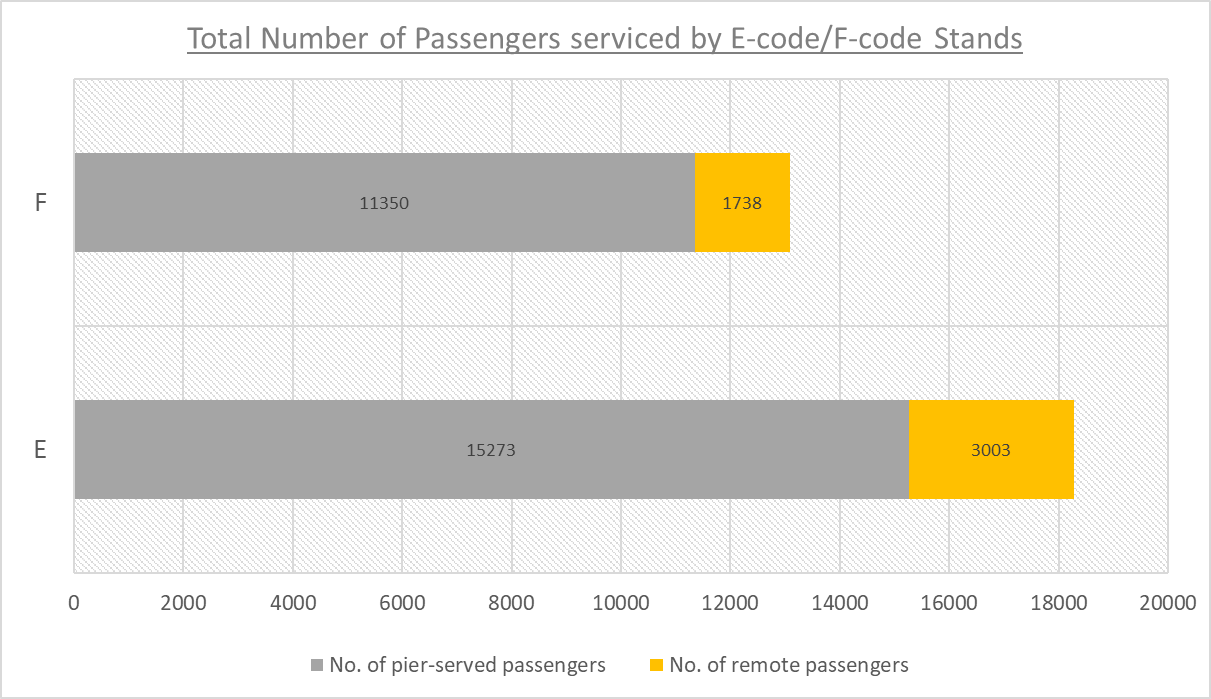
*Figure 4.2(i): A breakdown of no. of passengers serviced by each set of C-code stands*

The pier-served D-code stands were restricted between international and domestic flights. The results showed that **all** the passengers were **pier-served.** These results have been displayed on the bar chart in Figure 4.2(ii). All the remote D-code stands were only allocated with overnight parking.



*Figure 4.2(ii): A breakdown of no. of passengers serviced by each set of D-code stands*

For the E-code and F-code stands, all the pier-served stands were ones that accepted international flights. The number of **pier-served passengers** at the E-code stands was **15,273** and at F-code stands was **11,350**. The number of **remote passengers** at the E-code stands was **3003** and at F-code stands was **1738**. This included the passengers of flights allocated to the remote E and F-code stands and of any domestic flights allocated to the pier-served stands. These results have been summarised in Figure 4.2(iii).



*Figure 4.2(iii): A breakdown of no. of passengers serviced by E-code and F-code stands*

The stand allocation produced resulted in a **Pier-Service Level** (the proportion of pier-served passengers) **of 81.1%.**

# 5. DISCUSSION

The main finding from the results was that the Pier-Service Level was 81.1%. this was the most important result because the objective of the SAP was to maximise the number of pier-served passengers. Another key finding from the results was that the Stand Allocation Heuristic has a rate of allocation of 52.4%. When the heuristic was applied to data from the whole month of July, the rate of allocation was consistently between 50%-55% every day.

The Pier-Service Level is a percentage measure for the number of passengers at pier-served stands and the one achieved in this project was very high. This was as expected since the heuristic had features (like the partitioning of the stands) that actively prevented allocating flights to stands that would provide them with a remote service. The heuristic was also set up to greedily choose to allocate flights to stands that had served the most passengers. These allowed the number of pier-served passengers increase. However, the stand allocation produced by Arup using CAST software had a Pier-Service Level of 89%. This meant that the optimised Pier-Service Level produced by the heuristic was 8.9% lower than the required level.

The rate of allocation of 52.4% meant more than half of the daily flights were allocated to stands at Terminal 5. Those unallocated flights were assumed to be redirected to other terminals. An optimal solution would have an allocation rate of 100% hence this mid-level result was a lot lower than the expected rate. Using the CAST software, Arup had achieved a rate of allocation of 100%.

There were several factors that affected the quality of the results of this project and caused the differences between the CAST results.

SAPs are very complexed problems by nature, therefore, the SAP solved in this project was a simplified version of the reality. In order to reduce the complexity of the SAP, the stand layout defined in chapter 3 excluded some stands that actually exist in the Terminal 5 apron. The stand layout also ignored the BA bases and aircraft hangers from the problem – the BA bases are remote stands (that requires bus access) that are often used to park overnight flights. This meant the heuristic was allocating overnight parking operations to pier-served stands, hence occupying the spaces that could have, otherwise, been used for early morning or late-night flights. However, the SAP solved using the CAST software was an exact depiction of reality.

Another simplification of the SAP was that it only had one objective – to maximise the number of pier-served passengers. This meant that some features of the Stand Allocation Heuristic had a positive effect on the objective but caused the solution to be lacking in other areas. One such feature was the partitioning of the stands. As mentioned earlier, this actively discouraged the allocation of flights to remote stands resulting in the high pier-service level. However, it also meant that even if a stand was available, a flight was not allocated to it if the stand provided a remote service to the flight. The resulting effect of this was that the objective was maximised, but the number of failed allocations was also increased, hence reducing the rate of allocation.

The SAP used the buffer times 25 minutes and 30 minutes and the solution produced by the heuristic was successful at preventing any overlaps at the stands. Arup’s CAST solution used the minimal buffer times of 10 minutes and 15 minutes. This was another factor that affected the difference in allocation rates between the heuristic and the CAST solution.

It should also be noted that the stand allocations achieved through this project were from a heuristic. Heuristics approaches produce solutions that are good but never optimal. The CAST software, like other stand allocation software, takes an input of data and assumptions but the procedure used to derive the solution is a black box. This was the reason that Arup, and a lot of other airport planning companies, apportion funds for research on other methods of solving SAPs. The Stand Allocation Heuristic produced in this project was the initial step towards this goal.

# 6. CONCLUSION

This project introduced the airport system as a combination of airport structures and airport operations. The airport operation that was focused on was Stand Allocation. The SAP in this project was defined as an optimisation problem with the objective to maximise the number of pier-served passengers.

The first primary objective of this project was to automate the process of creating the input for the SAP – the Flight Turnarounds table including the turnaround splits. This was achieved by modelling the flight turnarounds on Python, during the data manipulation stage of the project.

The second primary objective of the project was to define and implement a heuristic to solve the SAP. A Stand Allocation Heuristic was derived using concepts from the Greedy heuristic and the BLS heuristic. This heuristic was modelled using Python and implemented on the turnarounds data. The solution from the heuristic resulted in a high Pier-Service Level of 81.1% which indicated that the objective of the SAP was achieved.

The allocation schedule produced by the heuristic did not have any flights assigned to incompatible stands (i.e. larger aircraft in smaller stands) and none of the stands had flights (including buffer times) that were overlapping. The Python models designed were successfully applied on flight data for the whole year. These attributes of the solution and the Python models meant that majority of the aims of the project were achieved. The only aim that was not fulfilled by the project was that the heuristic did not successfully allocate every single flight to a Terminal 5 stand.

The solution of the heuristic was compared with the solution produced by the CAST software. The results showed discrepancies between these solutions. It was acknowledged that this was caused by the simplification of the SAP for this project and because the procedure used by CAST to solve the SAP was unknown.

The Turnarounds table produced was accurate and was applicable to yearly data. Therefore, it was concluded that the Python model for the data manipulation was successful. Despite the failed allocations, the heuristic solution had a very high Pier-Service Level. Therefore, it was concluded that the Stand Allocation Heuristic was successful at focusing on increasing the number of pier-served passengers. However, further work would be required to produce a heuristic that also succeeded in reducing the number of failed allocations.

# 7. RECOMMENDATIONS

The analysis of the results concluded that the focus of any future studies needs to be on increasing the rate of allocation of the Stand Allocation Heuristic.

To achieve this, the formulated SAP and allocation heuristic would need to be modified before being reapplied on the data.

The SAP could be defined to be multi-objective: to maximise, both, the number of flights assigned to a stand and the total number of pier-served passengers. New variables could be added to the SAP to include the BA bases that were excluded from this study.

For the heuristic solution to be better compared to the CAST solution, the heuristic could be reapplied, on the turnarounds, with the minimal buffer time. However, making these modifications would make the SAP, hence the allocation heuristic, a lot more complex.

In the Python model of the stand allocation heuristic, the code for identifying and reassigning the infeasible/incomplete allocations was too long and not very efficient. Therefore, in order to improve the overall processing time of the model, this piece of code could be refined further.

# REFERENCES

The Geography of Transport Systems. (2018). Air Transport. [online] Available at: https://transportgeography.org/?page\_id=1765 [Accessed 01 Sep. 2018].

Schmitt, D. and Gollnick, V. (2016). *Air transport system*. Springer-Verlag Wien, p.Chapter 2 Historical Development of Air Transport.

The Economic Times. (2018). Air transport changed significantly in last 50 years. [online] Available at: https://economictimes.indiatimes.com/industry/transportation/airlines-/-aviation/air-transport-changed-significantly-in-last-50-years/articleshow/24816429.cms [Accessed 01 Sep. 2018].

Central Region Airports Division (2018). n 03/01/2018 AIP Sponsor Guide - 500. [ebook] FAA, p.500 - Airport Planning. Available at: https://www.faa.gov/airports/central/aip/sponsor\_guide/media/0500.pdf [Accessed 20 Sep. 2018].

Arup.com. (2018). [online] Available at: https://www.arup.com/expertise/services/infrastructure/airport-planning [Accessed 20 Sep. 2018].

Janić, M. (2013). Airport Analysis, Planning and Design. Hauppauge: Nova Science Publishers, Inc.

Kazda, A. and Caves, R. (2015). Airport design and operation. Bingley: Emerald.

Skybrary.aero. (2018). ICAO Aerodrome Reference Code - SKYbrary Aviation Safety. [online] Available at: https://www.skybrary.aero/index.php/ICAO\_Aerodrome\_Reference\_Code [Accessed 18 Sep. 2018].

[9] National Academies of Sciences, Engineering, and Medicine (2013). Apron planning and design guidebook. Washington, D.C: The National Academies Press.

Guépet, J., Acuna-Agost, R., Briant, O. and Gayon, J. (2015). Exact and heuristic approaches to the airport stand allocation problem. European Journal of Operational Research, 246(2), pp.597-608.

Gate Assignment Solution (GAM) Using Hybrid Heuristics Algorithm. (2009). Airline Innovation Centre of Excellence Team,Wipro Technologies, pp.1-9.

Justesen, T., Larsen, J. and Dohn, A. (2014). Allocation of Ground Handling Resources at Copenhagen Airport. Department of Management Engineering, Technical University of Denmark.

Eurocontrol. (2018). Delays – three questions and many answers. [online] Available at: https://www.eurocontrol.int/news/delays-three-questions-and-many-answers [Accessed 3 Sep. 2018].

Chun, A., Chan, S., Tsang, F. and Yeung, D. (2000). Stand-Allocation System (SAS) - A Constraint-Based System Developed with Software Components. AI Magazine, (Volume 21 Number 4), pp.1-12.

Benlic, U., Burke, E. and Woodward, J. (2017). Breakout local search for the multi-objective gate allocation problem. Computers & Operations Research, 78, pp.80-93.

Uk.mathworks.com. (2018). Biogeography-Based Optimizer (BBO) for training Multi-Layer Perceptron (MLP) - File Exchange - MATLAB Central. [online] Available at: https://uk.mathworks.com/matlabcentral/fileexchange/45804-biogeography-based-optimizer-bbo-for-training-multi-layer-perceptron-mlp?focused=9301906&tab=function [Accessed 02 Sep. 2018].

Marinelli, M., Palmisano, G., Dell’Orco, M. and Ottomanelli, M. (2015). Fusion of Two Metaheuristic Approaches to Solve the Flight Gate Assignment Problem. Transportation Research Procedia, 10, pp.920-930.

Airlinesoftware.net. (2018). Planning & scheduling. [online] Available at: https://www.airlinesoftware.net/products/15/planning--scheduling [Accessed 11 Sep. 2018].

Ibsplc.com. (2018). Airport Operations - IBS Software. [online] Available at: https://www.ibsplc.com/products/airport-operations [Accessed 11 Sep. 2018].

ARC. (2018). CAST - ARC. [online] Available at: https://airport-consultants.com/cast-simulation/ [Accessed 11 Sep. 2018].

# APPENDIX I : Initial Data

**Table-A1** – A sample of the initial Flight Schedule data

The complete dataset had 47,1072 entries.

The table below shows all the columns of the table, with the first 5 and last 5 data entries.



**Table-A2** – A sample of the initial Airport Details data

This table shows 15, randomly selected, entries of the airport details dataset.



**Table-A3** – A sample of the final Flight Turnarounds data

5 records from the start, middle and end of the Turnarounds table are shown below.

The empty fields represent overnight flights.



# APPENDIX II : Python Scripts

**(*Data Manipulation*)**

**Section-A1** – The function for loading the initial data onto Python

**Load\_File**

import pandas as pd

import numpy as np

*# needs input of the file names for flight schedule & airport data*

def **Load\_File**(Schedule\_file, Airport\_file):

global schedule, Airport\_Details

*#Read flight schedule onto dataframe*

schedule = pd.read\_csv('C:/Users/User/OneDrive - University of Southampton /Dissertation/Data\_Manipulation/' + Schedule\_file)

*#Drop irrelevant data*

schedule = schedule.drop(columns = ['Scheduled.Date',\

'Scheduled.Time', 'Stand.Date',

'Stand.Time', Actual.Date', 'Actual.Time','Flight.Type',

'Actual.Timestamp’,‘Stand.Timestamp’])

fields = ['AIRPORT\_IATA\_CODE', 'AIRPORT\_CITY', 'AIRPORT\_COUNTRY']

*#Read airport details*

Airport\_DF = pd.read\_csv('C:/Users/User/OneDrive - University of Southampton/Disseration/Data\_Manipulation/' + Airport\_file)

*#Extract useful information only onto dataframe*

Airport\_Details = pd.DataFrame(columns=fields,

data=Airport\_DF[fields])

**Section-A2** – The function for cleaning the data

**Select\_Data**

def **Select\_Data**(month, date):

global schedule

*#update datatypes for the dates and times*

schedule[['Scheduled.Timestamp']] = schedule[['Scheduled.Timestamp']]\

.apply(pd.to\_datetime, dayfirst = True)

*#get the training data:*

*#from month = 7 (July); day = 21st; Terminal = T5*

schedule = schedule.loc[(schedule\

['Scheduled.Timestamp'].dt.month == month) &

(schedule['Terminal'] == 5) &

(schedule['Scheduled.Timestamp'].dt.day == date)]

**Section-B1** – Functions for the first step of creating the Flight Turnarounds

*These 2 functions matched and joined together arrival and departure flights for each aircraft in the flight schedule.*

**Sort\_cols**

def **Sort\_cols**():

global schedule, cols

*#create a list of all the column names*

cols = schedule.columns.tolist()

*#create a new column “ID” using the registration numbers.*

cols.insert(0, cols.pop(cols.index('Reg.No')))

schedule = schedule.loc[:, cols]

schedule['ID'] = schedule.index

**Create\_Turns**

def **Create\_Turns**():

*#sort schedule by Origin/Destination, Registration & Scheduled Time*

sort\_cols = ['ID','Reg.No', 'Scheduled.Timestamp']

schedule.sort\_values(by=sort\_cols, inplace = True)

*#convert index into column and create new shifted id: “nextID”*

schedule['nextID'] = schedule.groupby(sort\_cols[1])['ID'].\

shift(1)

*#filter arrivals & departures and outer join by registration numbers*

*# add suffix to indicate arrival and departure data*

*#create a dataframe -turns*

global turns

turns = schedule[schedule['A.D'] == 'A'].\

merge(schedule[schedule['A.D'] == 'D'],

how='outer',

left\_on= 'ID',

right\_on='nextID',

suffixes=('\_Arr','\_Dep'))

*#coalesce turnarounds into a better format*

for c in sort\_cols:

turns['sort\_'+ c] = turns[c + '\_Arr']\

.combine\_first(turns[c + '\_Dep'])

turns.sort\_values(by=['sort\_{}'.\

format(c) for c in sort\_cols], inplace = True)

turns = turns[['{}\_Arr'.format(c) for c in cols] +\

['{}\_Dep'.format(c) for c in cols]]

**Section-B2** – Function to create the initial Turnarounds table

**Turnarounds**

def **Turnarounds**():

global Turnarounds,Turnarounds\_loc

*(Section of the code for reformatting the table is omitted)*

*# Add 7 empty columns to the Turnarounds dataframe*

Turnarounds['Stand\_Arrive'] =""

Turnarounds['Origin\_City'] =""

Turnarounds['Origin\_Country'] =""

Turnarounds['Stand\_Depart'] =""

Turnarounds['Dest\_City'] =""

Turnarounds['Dest\_Country'] =""

Turnarounds['Turnover'] = ""

*# create a list of column names in the dataframe*

cols2 = Turnarounds.columns.tolist()

*# reorder columns*

cols2 = cols2[:8] + [cols2[-7]] +[cols2[-6]] +[cols2[-5]]+ cols2[8:18] + [cols2[-4]]+ [cols2[-3]] + [cols2[-2]]+ cols2[18:25]+[cols2[-1]]

Turnarounds = Turnarounds[cols2]

*# fill empty columns with data from Airport Details dataset*

Turnarounds['Origin\_City'] = Turnarounds['Origin']\

.map(Airport\_Details\

.set\_index('AIRPORT\_IATA\_CODE')['AIRPORT\_CITY'])

Turnarounds['Origin\_Country'] = Turnarounds['Origin']\

.map(Airport\_Details\

.set\_index('AIRPORT\_IATA\_CODE')['AIRPORT\_COUNTRY'])

Turnarounds['Dest\_City'] = Turnarounds['Destination']\

.map(Airport\_Details\

.set\_index('AIRPORT\_IATA\_CODE')['AIRPORT\_CITY'])

Turnarounds['Dest\_Country'] = Turnarounds['Destination']\

.map(Airport\_Details\

.set\_index('AIRPORT\_IATA\_CODE')['AIRPORT\_COUNTRY'])

*# Calculate the turnarounds times*

Turnarounds['Turnover'] = pd.to\_timedelta(\

Turnarounds['Scheduled\_Timestamp\_Depart']\

- Turnarounds['Scheduled\_Timestamp\_Arrive'])

*# Sort the Turnarounds in order of first arrival*

Turnarounds = Turnarounds.sort\_values(by=['Scheduled\_Timestamp\_Arrive'],

ascending = True)

**Section-B3** – Function that executed the splitting process

*The script for this function is only a sample. This section shows the process for the overnight flights departing in the morning. The code for other overnight flights followed the same structure.*

**Splitting**

def **Splitting**():

global Turn\_NaTDep, Turn\_NaTArr, Turn\_8hrsE, Turn\_8hrsF, Turn\_unsplit

*# Extract data of overnight flight leaving in the morning*

Turn\_NaTDep = Turnarounds[pd.isnull(\

Turnarounds['Scheduled\_Timestamp\_Depart'])== True]

*# create 2 duplicates of each row*

Turn\_NaTDep = pd.DataFrame(np.repeat(Turn\_NaTDep.values,2,axis=0),

columns=Turnarounds.columns)

*# Calculate and update arrival and departure times for the 3 towing operations*

*# fleet C*

for i in Turn\_NaTDep.index:

TimeStamp = pd.to\_datetime(str((schedule.loc[schedule.index[0],\

'Scheduled.Timestamp']).date() \

+ pd.to\_timedelta('1 day')) + ' 12:00AM')

if Turn\_NaTDep.loc[i, 'ICAO'] == 'C':

if (pd.isnull(Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']))==\

True:

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

+ pd.to\_timedelta('45 minutes')

Turn\_NaTDep.loc[i+1 , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']

Turn\_NaTDep.loc[i+1, 'Scheduled\_Timestamp\_Depart'] = TimeStamp

Turn\_NaTDep.loc[i+1, 'Stand\_Arrive'] = 'R'

*# fleet D*

if Turn\_NaTDep.loc[i, 'ICAO'] == 'D':

if (pd.isnull(Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']))==\

True:

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

+ pd.to\_timedelta('60 minutes')

Turn\_NaTDep.loc[i+1 , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']

Turn\_NaTDep.loc[i+1, 'Scheduled\_Timestamp\_Depart'] = TimeStamp

Turn\_NaTDep.loc[i+1, 'Stand\_Arrive'] = 'R'

*# fleet E*

if Turn\_NaTDep.loc[i, 'ICAO'] == 'E':

if (pd.isnull(Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']))==\

True:

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Arrive']\

+ pd.to\_timedelta('90 minutes')

Turn\_NaTDep.loc[i+1 , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']

Turn\_NaTDep.loc[i+1, 'Scheduled\_Timestamp\_Depart'] = TimeStamp

Turn\_NaTDep.loc[i+1, 'Stand\_Arrive'] = 'R'

*# fleet F*

if Turn\_NaTDep.loc[i, 'ICAO'] == 'F':

if (pd.isnull(Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']))==\

True:

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart'] =

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Arrive']\

+ pd.to\_timedelta('100 minutes')

Turn\_NaTDep.loc[i+1 , 'Scheduled\_Timestamp\_Arrive'] = \

Turn\_NaTDep.loc[i, 'Scheduled\_Timestamp\_Depart']

Turn\_NaTDep.loc[i+1, 'Scheduled\_Timestamp\_Depart'] = TimeStamp

Turn\_NaTDep.loc[i+1, 'Stand\_Arrive'] = 'R'

*This section shows the splitting of fleet F aircraft with turnarounds > 8 hours. The fleet E aircraft with turnarounds > 8hours followed the same structure.*

**Splitting (continued)**

*# Extract data of flights with turnarounds > 8hrs in fleet F*

Turn\_8hrsF = Turnarounds.loc[(Turnarounds['Turnover'] > pd.to\_timedelta('480 minutes')) & (Turnarounds['ICAO'] == 'F')]

*# create duplicates of each row*

Turn\_8hrsF = pd.DataFrame(np.repeat(Turn\_8hrsF.values,3,axis=0),

columns=Turnarounds.columns)

*#calculate and update arrival and departure times of the 3 towing operations*

for i in Turn\_8hrsF.index:

if i == 0:

Turn\_8hrsF.loc[i, 'Scheduled\_Timestamp\_Depart'] = \

Turn\_8hrsF.loc[i, 'Scheduled\_Timestamp\_Arrive']\

+ pd.to\_timedelta('100 minutes')

elif i in Turn\_8hrsF.index[1: len(Turn\_8hrsF)-1]:

if Turn\_8hrsF.loc[i, 'Reg\_No'] == \

Turn\_8hrsF.loc[i-1, 'Reg\_No'] == \

Turn\_8hrsF.loc[i+1, 'Reg\_No']:

Turn\_8hrsF.loc[i , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_8hrsF.loc[i-1, 'Scheduled\_Timestamp\_Depart']

Turn\_8hrsF.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_8hrsF.loc[i, 'Scheduled\_Timestamp\_Depart'] \

- pd.to\_timedelta('120 minutes')

Turn\_8hrsF.loc[i, 'Stand\_Arrive'] = 'R'

continue

elif Turn\_8hrsF.loc[i, 'Reg\_No'] == \

Turn\_8hrsF.loc[i-1, 'Reg\_No']!= \

Turn\_8hrsF.loc[i+1, 'Reg\_No']:

Turn\_8hrsF.loc[i , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_8hrsF.loc[i-1, 'Scheduled\_Timestamp\_Depart']

continue

elif Turn\_8hrsF.loc[i, 'Reg\_No'] == \

Turn\_8hrsF.loc[i+1, 'Reg\_No'] != \

Turn\_8hrsF.loc[i-1, 'Reg\_No']:

Turn\_8hrsF.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_8hrsF.loc[i, 'Scheduled\_Timestamp\_Arrive']\

+ pd.to\_timedelta('100 minutes')

elif i == len(Turn\_8hrsF)-1:

Turn\_8hrsF.loc[i , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_8hrsF.loc[i-1,'Scheduled\_Timestamp\_Depart']

*This section shows the splitting of fleet C aircraft with turnarounds > 11.5 hours. The fleet D aircraft with turnarounds > 11.5 hours followed the same structure.*

**Splitting (continued)**

*# Extract data of flights with turnarounds > 11.5 hrs in fleet C*

Turn\_11hrsC = pd.DataFrame(\

np.repeat(Turn\_11hrsC.values,3,axis=0),\

columns=Turnarounds.columns)

for i in Turn\_11hrsC.index:

if i == 0:

Turn\_11hrsC.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_11hrsC.loc[i, 'Scheduled\_Timestamp\_Arrive']\

+ pd.to\_timedelta('45 minutes')

elif i in Turn\_11hrsC.index[1: len(Turn\_11hrsC)-1]:

if Turn\_11hrsC.loc[i, 'Reg\_No'] == \

Turn\_11hrsC.loc[i-1, 'Reg\_No'] == \

Turn\_11hrsC.loc[i+1, 'Reg\_No']:

Turn\_11hrsC.loc[i , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_11hrsC.loc[i-1, 'Scheduled\_Timestamp\_Depart']

Turn\_11hrsC.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_11hrsC.loc[i, 'Scheduled\_Timestamp\_Depart']\

- pd.to\_timedelta('45 minutes')

Turn\_11hrsC.loc[i, 'Stand\_Arrive'] = 'R'

continue

elif Turn\_11hrsC.loc[i, 'Reg\_No'] ==\

Turn\_11hrsC.loc[i-1, 'Reg\_No']!= \

Turn\_11hrsC.loc[i+1, 'Reg\_No']:

Turn\_11hrsC.loc[i , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_11hrsC.loc[i-1, 'Scheduled\_Timestamp\_Depart']

continue

elif Turn\_11hrsC.loc[i, 'Reg\_No'] == \

Turn\_11hrsC.loc[i+1, 'Reg\_No']!= \

Turn\_11hrsC.loc[i-1, 'Reg\_No']:

Turn\_11hrsC.loc[i, 'Scheduled\_Timestamp\_Depart'] =\

Turn\_11hrsC.loc[i, 'Scheduled\_Timestamp\_Arrive']\

+ pd.to\_timedelta('45 minutes')

elif i == len(Turn\_11hrsC)-1:

Turn\_11hrsC.loc[i , 'Scheduled\_Timestamp\_Arrive'] =\

Turn\_11hrsC.loc[i-1, 'Scheduled\_Timestamp\_Depart']

**Section-B4** – Function to create and save the final Turnarounds table

**Final\_Turnarounds**

def **final\_Turnarounds**():

global Turnarounds

Turnarounds = pd.concat([Turn\_unsplit, Turn\_8hrsE, Turn\_8hrsF,

Turn\_NaTArr, Turn\_NaTDep])

*# Re-calculate Turnaround times*

Turnarounds['Turnover'] = pd.to\_timedelta(\

Turnarounds['Scheduled\_Timestamp\_Depart']\

- Turnarounds['Scheduled\_Timestamp\_Arrive'])

*#Re-sort the Turnarounds according to first arrival*

Turnarounds = Turnarounds.sort\_values(\

by=['Scheduled\_Timestamp\_Arrive'], ascending = True)\

.reset\_index(drop = True)

*#Save the final Turnarounds table as a CSV file*

Turnarounds.to\_csv("Turnarounds\_Final.csv", index = False)

**(*Stand Allocation Heuristic*)**

**Section-C1** – The functions for allocating fleet F flights

**F\_fl:** Filters the fleet F flights, creates stands sets.

import pandas as pd

import numpy as np

def **F\_fl**():

global F, F\_Pier, F\_Remote, ind\_F, ind\_FRemote, ind\_FPier

Fleet = Turnarounds.groupby(['ICAO'])

*#Filter into groups of Fleets (C, D, E and F)*

F = Fleet.get\_group('F')

*# Define Sets of stands and list of remote and pier-served flights #*

F\_Pier = np.array([555, 556, 557, 561, 562, 563, 564, 565, 544, 545,

546, 547])

F\_Remote = np.array([558, 575, 576])

ind\_FRemote = np.array((F.loc[F['Stand\_Arrive']=='R']).index.tolist())

ind\_FPier = np.array((F.loc[F['Stand\_Arrive']!='R']).index.tolist())

**Schedule\_F:** The allocation of all F-code flights

def **Schedule\_F**():

*# Manipulate 'F' aircraft data to use in scheduling aglorithm #*

**F\_fl()**

*# Allocate towed flights to Remote stands first - so they don't block up* *pier-served stands #*

**Remote\_F()**

*# Allocate flights to the Pier-served stands #*

**Pier\_F()**

**Remote\_F:** Allocates flights with long turnarounds to the remote stands

def **Remote\_F**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_FRemote:

if Turnarounds.index[i] == ind\_FRemote[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = F\_Remote[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i, 'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_FRemote:

if Turnarounds.index[i] in ind\_FRemote[1:]:

*# Sort stands by the greatest number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: g.assign(\

col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df[df['Stand\_Arrive'].isin(F\_Remote)]

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',ascending= False).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

*# Perform a Buffer times check #*

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] = Turnarounds.loc[i,'Stand\_Arrive']

break

else:

index = np.where(F\_Remote == stand)[0]

*# Pick stand with next most passengers served #*

stand = np.int(F\_Remote[(index +1) % 3])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

*# Perform buffer times check #*

Arrival\_Check = Turnarounds\

.loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] –\

Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('30 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i,'Stand\_Arrive']

else:

pass

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i,'Stand\_Arrive']

break

**Pier\_F:** Allocates all the flights to the Pier-served stands

def **Pier\_F**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_FPier:

if Turnarounds.index[i] == ind\_FPier[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = F\_Pier[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i, 'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_FPier:

if Turnarounds.index[i] in ind\_FPier[1:]:

*# Sort stands by the greatest number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df[df['Stand\_Arrive'].isin(F\_Pier)]

df[['Stand\_Arrive']] = df[['Stand\_Arrive']].apply(pd.to\_numeric)

*# Pick stand with the greatest number of passengers served #*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',ascending= False).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

*# perform buffer check #*

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] >

Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] >=

pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i,'Stand\_Arrive']

break

else:

index = np.where(F\_Pier == stand)[0]

*# pick stand with next greatest number of passengers served #*

stand = np.int(F\_Pier[(index +1) % 12])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive']> Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],\

'Scheduled\_Timestamp\_Depart']>= pd.to\_timedelta('30 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i,'Stand\_Arrive']

else:

pass

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

break

**Section-C2** – The functions for allocating fleet E flights

**E\_fl:** Filters the fleet E flights, creates stands sets.

import pandas as pd

import numpy as np

def **E\_fl**():

global E, E\_Pier, E\_Remote, ind\_EPier, ind\_ERemote

Fleet = Turnarounds.groupby(['ICAO'])

E = Fleet.get\_group('E')

E\_Pier = np.array([518, 532, 533, 534, 535, 536, 537, 538, 539, 542,

543, 552, 553, 554, 566])

E\_Remote = np.array([531, 541, 548, 551, 567, 568, 572, 573])

ind\_ERemote = np.array((E.loc[E['Stand\_Arrive']=='R']).index.tolist())

ind\_EPier = np.array((E.loc[E['Stand\_Arrive']!='R']).index.tolist())

**Schedule\_E:** The allocation of all E-code flights

def **Schedule\_E**():

*# Manipulate 'E' aircraft data to use in scheduling aglorithm #*

**E\_fl()**

*# Allocate towed flights to Remote stands first - so they don't block up pier-served stands #*

**Remote\_E()**

*# Allcoate unallocated towed flights to Remote 'F' stands #*

**Remote\_Eleft()**

*# Allocate flights to the Pier-served stands #*

**Pier\_E()**

*# Allcoate unallocated flights to pier-served 'F' stands #*

**E\_to\_FPier()**

*# Allcoate unallocated flights to Remote 'E' stands #*

**E\_to\_ERemote()**

*# Allocate unallocated flights to Remote 'F' stands #*

**E\_to\_FRemote()**

**allo\_checker:** Checks feasibility of allocations and removes infeasible ones

def **allo\_checker**(ICAO, Stands):

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(Stands)]

f\_allo = f\_allo[f\_allo['ICAO'] == str(ICAO)]

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).sort\_values\

(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] > f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] - f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = ''

Turnarounds.loc[index1, 'Stand\_Depart'] = ''

**Remote\_E:** Allocates flights with long turnarounds to the remote stands

def **Remote\_E**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_ERemote:

if Turnarounds.index[i] == ind\_ERemote[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = E\_Remote[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,’Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_ERemote:

if Turnarounds.index[i] in ind\_ERemote[1:]:

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending=

False)

df = df[df['Stand\_Arrive'].isin(E\_Remote)]

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] >=

pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(E\_Remote == stand)[0]

stand = np.int(E\_Remote[(index +1) % 8])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Arrive'] – Turnarounds.loc[Turnarounds.index[f],\

'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i, 'Stand\_Arrive']

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

**Remote\_Eleft:** Reallocates infeasible remote flights

def **Remote\_Eleft**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_ERemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values\

(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in F\_Remote:

Arrive =(Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

– pd.to\_timedelta('30 minutes'))

Depart =(Turnarounds.loc[i,'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('30 minutes'))

df1 = df[df['Stand\_Arrive'] == k]

if df1.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

for (j, rows) in df1.iterrows():

Arrival\_Check = (Arrive < df1.loc[j, 'Scheduled\_Timestamp\_Arrive']) ==True and (df1.loc[j, 'Scheduled\_Timestamp\_Arrive']< Depart) == True

Depart\_Check = (Arrive < df1.loc[j, 'Scheduled\_Timestamp\_Depart']) ==True and (df1.loc[j, 'Scheduled\_Timestamp\_Depart']< Depart) == True

if Arrival\_Check == False and Depart\_Check ==

False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

pass

**Pier\_E:** Allocates flights to pier-served stands

def **Pier\_E**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.index[i] == ind\_EPier[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = E\_Pier[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.index[i] in ind\_EPier[1:]:

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df[df['Stand\_Arrive'].isin(E\_Pier)]

df[['Stand\_Arrive']] = df[['Stand\_Arrive']].apply(pd.to\_numeric)

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',\

ascending= False).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check=(Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[index1],\

'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[index1],\

'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(E\_Pier == stand)[0]

stand = np.int(E\_Pier[(index +1) % 15])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[f],\

'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

**E\_to\_FPier:** Reallocates infeasible E flights to other stands

def **E\_to\_FPier**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in F\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty ==\

True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] –

pd.to\_timedelta('30 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] +

pd.to\_timedelta('30 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

*#Check of errors in allocations and correct*

allo\_checker('E', F\_Pier)

*#Reallocate to better gates*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Pier)]

for f in F\_Pier:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] =\

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i,'Stand\_Arrive']

**E\_to\_ERemote:** Reallocates infeasible E flights to other stands

def **E\_to\_ERemote**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in E\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] -\

pd.to\_timedelta('30 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] +\

pd.to\_timedelta('30 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True and\

(Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True and\

(Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

*#Check of errors in allocations and correct*

allo\_checker('E', E\_Remote)

*#Reallocate to better gates*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Remote)]

for f in E\_Remote:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep='first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] =

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

*#Check of errors in allocations and correct*

allo\_checker('E', E\_Remote)

**Section-C3** – The functions for allocating fleet D flights

**E\_to\_FRemote:** Reallocates infeasible E flights to other stands

def **E\_to\_FRemote**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in F\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - pd.to\_timedelta('30 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] + pd.to\_timedelta('30 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

*#Check of errors in allocations and correct*

allo\_checker('E', F\_Remote)

*#Reallocate to better gates*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_EPier:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Remote)]

for f in F\_Remote:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep='first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] =

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

**Schedule\_D:** The allocation of all D-code flights

def **Schedule\_D**():

*# Manipulate 'D' aircraft data to use in scheduling aglorithm #*

**D\_fl()**

*# Allocate towed flights to Remote stands first - so they don't block up pier-served stands #*

**Remote\_D()**

*# Allcoate unallocated towed flights to 'F' and ‘E’ stands #*

**RemoteD\_to\_FPier()**

**RemoteD\_to\_EPier()**

*# Allocate domestic flights to the Pier-served stands #*

**UKArr\_D()**

**UKDep\_D()**

*# Allocate international flights to the Pier-served stands #*

**Int\_D()**

*# Reallcoate unallocated flights to other stands #*

**PierD\_to\_EPier()**

**PierD\_to\_FRemote()**

**D\_fl:** Filters the fleet D flights, creates stands sets.

def **D\_fl**():

global D, D\_UK, D\_Int, D\_Remote, ind\_DUK\_Arr

global ind\_DUK\_Dep,ind\_DRemote, ind\_DInt

Fleet = Turnarounds.groupby(['ICAO'])

*#Filter into groups of Fleets (C, D, E and F)*

D = Fleet.get\_group('D')

DPier = D.loc[(D['Stand\_Arrive'] != 'R')]

DRemote = D.loc[(D['Stand\_Arrive'] == 'R')]

*# Domestic Flights #*

DUK\_Arr = DPier.loc[(DPier['Origin\_Country'] == 'United Kingdom')]

DUK\_Dep = DPier.loc[(DPier['Origin\_Country'] != 'United Kingdom') &\

(DPier['Dest\_Country'] == 'United Kingdom')]

*# International Flights #*

DInt = DPier.loc[(DPier['Origin\_Country'] != 'United Kingdom') &\

(DPier['Dest\_Country'] != 'United Kingdom')]

*#Stands#*

D\_Remote = np.array([581, 505, 508])

D\_UK = np.array([505, 508])

D\_Int = np.array([512, 513, 505, 508])

ind\_DRemote = np.array(DRemote.index.tolist())

ind\_DUK\_Arr = np.array(DUK\_Arr.index.tolist())

ind\_DUK\_Dep = np.array(DUK\_Dep.index.tolist())

ind\_DInt = np.array(DInt.index.tolist())

**Remote\_D:** Allocates flights with long turnarounds to the remote stands

def **Remote\_D**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DRemote:

if Turnarounds.index[i] == ind\_DRemote[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = D\_Remote[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,’Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DRemote:

if Turnarounds.index[i] in ind\_DRemote[1:]:

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending=

False)

df = df[df['Stand\_Arrive'].isin(D\_Remote)]

*# pick stand with largest number of passengers served #*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(D\_Remote == stand)[0]

stand = np.int(D\_Remote[(index +1) % 3])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

**RemoteD\_to\_FPier:** Reallocates infeasible remote D flights to pier-served stands

def **RemoteD\_to\_FPier():**

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in F\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True\

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Pier)]

f\_allo = f\_allo[f\_allo['ICAO'] == 'D']

*#Remove infeasible allocations*

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = 'R'

Turnarounds.loc[index1, 'Stand\_Depart'] = 'R'

**RemoteD\_to\_FPier (continued)**

*#Reallocate the removed flights to pier-served ‘F’ stands*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Pier)]

for f in F\_Pier:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] = int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

**RemoteD\_to\_EPier:** Reallocates infeasible remote D flights to pier-served stands

def **RemoteD\_to\_EPier**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in E\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True\

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

**RemoteD\_to\_EPier (continued)**

*#Remove infeasible allocations*

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Pier)]

f\_allo = f\_allo[f\_allo['ICAO'] == 'D']

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = 'R'

Turnarounds.loc[index1, 'Stand\_Depart'] = 'R'

*#Reallocate the removed flights to pier-served ‘E’ stands*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Pier)]

for f in E\_Pier:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']/

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] = int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

**UKArr\_D:** Allocates the Domestic Arrival, fleet D flights pier-served stands.

def **UKArr\_D**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DUK\_Arr:

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in D\_UK:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True \

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

continue

def **UKDep\_D**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DUK\_Dep:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(D\_UK)]

for f in D\_UK:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive',\

keep = 'first')

Arrival\_Check =(Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Arrive']> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] =\

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

*#Check for infeasible allocations and remove them*

allo\_checker('D', D\_UK)

**UKDep\_D:** Allocates the Domestic Departure, fleet D flights pier-served stands.

def **Int\_D**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DInt:

if Turnarounds.index[i] == ind\_DInt[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = D\_Int[0]

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_DInt:

if Turnarounds.index[i] in ind\_DInt[1:]:

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df[df['Stand\_Arrive'].isin(D\_Int)]

df[['Stand\_Arrive']] = df[['Stand\_Arrive']].apply(pd.to\_numeric)

*#pick the stands with the largest number of passengers served #*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',\

ascending= False).reset\_index()

**Int\_D:** Allocates the International, fleet D flights to pier-served stands.

*#Allocate to stands #*

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(D\_Int == stand)[0]

stand = np.int(D\_Int[(index +1) % 4])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

*#check for infeasible allocations and remove them #*

allo\_checker('D', D\_Int)

**Int\_D (continued)**

def **PierD\_to\_EPier**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.loc[i, 'ICAO'] == 'D':

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Pier)]

for f in E\_Pier:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] =\

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

continue

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Pier)]

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

> f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = ''

Turnarounds.loc[index1, 'Stand\_Depart'] = ''

**PierD\_to\_EPier:** Reallocate flights from pier-served D-code stands to other stands.

def **PierD\_to\_FRemote**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.loc[i, 'ICAO'] == 'D':

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Remote)]

for f in F\_Remote:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] = \

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

for (i, row) in Turnarounds.iterrows():

if Turnarounds.loc[i, 'ICAO'] == 'D':

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Remote)]

for f in F\_Remote:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

< f\_allo1['Scheduled\_Timestamp\_Arrive'])

Buffer = (f\_allo1['Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] = int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

**PierD\_to\_FRemote:** Reallocate flights from pier-served D-code stands to other stands.

**Section-C4** – The functions for allocating fleet C flights

**Schedule\_C:** The allocation of all C-code flights

def **Schedule\_C**():

**C\_fl()**

*# Towed flights*

**Remote\_C()**

*#Reallocate remote flights to E and F stands*

**RemoteC\_to\_EPier()**

**RemoteC\_to\_FPier()**

*# CTA: Flights from Dublin #*

**CTA\_C()**

*# Domestic Flights #*

**UKArr\_C()**

**UKDep\_C()**

*# International flights #*

**Int\_C()**

**C\_fl:** Filters the fleet C flights, creates stands sets.

def **C\_fl**():

global C, CTA, C\_Remote, C\_AllPier, C\_UK\_Arr, C\_UK\_Dep, C\_Int

global ind\_CRemote, ind\_CUK\_Arr, ind\_CUK\_Dep, ind\_CTA, ind\_CInt

Fleet = Turnarounds.groupby(['ICAO'])

*#Filter into groups of Fleets (C, D, E and F)*

C = Fleet.get\_group('C')

CPier = C.loc[(C['Stand\_Arrive'] != 'R')]

CRemote = C.loc[(C['Stand\_Arrive'] == 'R')]

*#CTA#*

C\_CTA = CPier.loc[(CPier['Origin\_City'] == 'Dublin')]

*# Domestic Flights #*

CUK\_Arr = CPier.loc[(CPier['Origin\_Country'] == 'United Kingdom')]

CUK\_Dep = CPier.loc[(CPier['Origin\_Country'] != 'United Kingdom') &\

(CPier['Origin\_City']!= 'Dublin')&(CPier['Dest\_Country']=='United Kingdom')]

*# International Flights #*

CInt = CPier.loc[(CPier['Origin\_Country'] != 'United Kingdom') &\

(CPier['Origin\_City']!= 'Dublin')&(CPier['Dest\_Country']!='United Kingdom')]

*#Stands#*

C\_Remote = np.array([524, 525, 526, 527])

C\_AllPier = np.array([501, 502, 503, 506, 507, 509, 511, 515, 517,

519, 520, 521, 522, 523])

C\_UK\_Arr = np.array([501, 502, 503, 506, 507])

C\_UK\_Dep = np.array([506, 507, 509, 511, 519, 522])

CTA = np.array([523])

C\_Int = np.array([515, 517, 520, 521, 522, 519, 511, 509,507, 506])

ind\_CRemote = np.array(CRemote.index.tolist())

ind\_CUK\_Arr = np.array(CUK\_Arr.index.tolist())

ind\_CUK\_Dep = np.array(CUK\_Dep.index.tolist())

ind\_CInt = np.array(CInt.index.tolist())

ind\_CTA = np.array(C\_CTA.index.tolist())

**C\_allo\_checker:** Checks feasibility of C-code allocations and removes infeasible ones

def **C\_allo\_checker(Stands):**

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(Stands)]

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = ''

Turnarounds.loc[index1, 'Stand\_Depart'] = ''

**Remote\_C:** Allocated the overnight C-code flights to remote stands

def **Remote\_C**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.index[i] == ind\_CRemote[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = C\_Remote[0]

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.index[i] in ind\_CRemote[1:]:

*# sort stands by number of passengers served #*

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df[df['Stand\_Arrive'].isin(C\_Remote)]

*#pick the stands with the largest number of passengers served #*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

*#Allocate to remote C stands#*

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

*(continued…)*

**Remote\_C (Continued)**

else:

index = np.where(C\_Remote == stand)[0]

stand = np.int(C\_Remote[(index +1) % 4])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

*#Allocate to pier-served C stands#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

Turnarounds.loc[i, 'Stand\_Arrive'] = C\_AllPier[0]

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df[df['Stand\_Arrive'].isin(C\_AllPier)]

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

*(continued…)*

**Remote\_C (Continued)**

else:

index = np.where(C\_AllPier == stand)[0]

stand = np.int(C\_AllPier[(index +1) % 14])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

**RemoteC\_to\_EPier:** Reallocate flights from remote C-code stands to other stands

def **RemoteC\_to\_EPier**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in E\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.\

empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] \

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True\

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

*(continued…)*

**RemoteC\_to\_EPier (continued)**

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Pier)]

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

> f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = 'R'

Turnarounds.loc[index1, 'Stand\_Depart'] = 'R'

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(E\_Pier)]

for f in E\_Pier:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', \

keep = 'first')

Arrival\_Check =(Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] =\

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']]

**RemoteC\_to\_FPier:** Reallocate flights from remote C-code stands to other stands

def **RemoteC\_to\_FPier**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in F\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k

, 'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] \

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True \

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True \

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Pier)]

for (f, row) in f\_allo.iterrows():

stand = f\_allo.loc[f, 'Stand\_Arrive']

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == stand]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True).reset\_index()

for (i, row) in f\_allo1.iloc[1:].iterrows():

Arrival\_Check =(f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

> f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart'])

Buffer = (f\_allo1.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- f\_allo1.loc[i-1,'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

continue

else:

index1 = int(f\_allo1.loc[i, 'index'])

Turnarounds.loc[index1, 'Stand\_Arrive'] = 'R'

Turnarounds.loc[index1, 'Stand\_Depart'] = 'R’

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CRemote:

if Turnarounds.loc[i, 'Stand\_Arrive'] == 'R':

f\_allo = Turnarounds[Turnarounds['Stand\_Arrive'].isin(F\_Pier)]

*(continued…)*

**RemoteC\_to\_FPier (Continued)**

for f in F\_Pier:

f\_allo1 = (f\_allo[f\_allo['Stand\_Arrive'] == f]).\

sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False).reset\_index()

f\_allo1 = f\_allo1.drop\_duplicates('Stand\_Arrive', \

keep = 'first')

Arrival\_Check =(Turnarounds.loc[i, \

'Scheduled\_Timestamp\_Arrive'] > f\_allo1['Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- f\_allo1['Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('25 minutes'))

if all(Buffer) == True and all(Arrival\_Check) == True:

index1 = int(f\_allo1[ 'index'])

Turnarounds.loc[i, 'Stand\_Arrive'] = \

int(f\_allo1['Stand\_Arrive'])

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

**CTA\_C:** Allocation of C-code CTA Arrival flights

def **CTA\_C**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CTA:

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= True)

df = df[df['Stand\_Arrive'].isin(CTA)]

*#pick stand with largest number of passengers*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive'] \

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(CTA == stand)[0]

stand = np.int(CTA[(index +1) % 1])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive']> Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart']=Turnarounds.loc[i,'Stand\_Arrive']

*(continued…)*

**CTA\_C (continued)**

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] = Turnarounds.loc[i,'Stand\_Arrive']

break

else:

pass

C\_allo\_checker(CTA)

*# Allocate remaining flights to C\_Int Stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CTA:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= True)

df = df[df['Stand\_Arrive'].isin(C\_Int)]

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,'Stand\_Arrive']

break

else:

index = np.where(C\_Int == stand)[0]

stand = np.int(C\_Int[(index +1) % 10])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

C\_allo\_checker(C\_Int)

**UKArr\_C:** Allocates the Domestic Arrival, fleet C flights pier-served stands.

def **UKArr\_C**():

*#Set stand for first flight#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] == ind\_CUK\_Arr[0]:

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: \

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= True)

df = df[df['Stand\_Arrive'].isin(C\_UK\_Arr)]

*#pick stand with largest number of passengers#*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(C\_UK\_Arr == stand)[0]

stand = np.int(C\_UK\_Arr[(index +1) % 5])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check=Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] =Turnarounds.loc[i,'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart']=\

Turnarounds.loc[i,'Stand\_Arrive']

break

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] == ind\_CUK\_Dep[0]:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

Turnarounds.loc[i, 'Stand\_Arrive'] = C\_UK\_Dep[0]

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

continue

*(continued…)*

**UKArr\_C (continued)**

*#Set stands for rest of the flights#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CUK\_Arr[1:]:

df = Turnarounds[Turnarounds.index.isin(ind\_CUK\_Arr)]

df = df.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df[df['Stand\_Arrive'].isin(C\_UK\_Arr)]

*#pick stand with largest number of passengers#*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(C\_UK\_Arr == stand)[0]

stand = np.int(C\_UK\_Arr[(index +1) % 5])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive']> Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

C\_allo\_checker(C\_UK\_Arr)

*(continued…)*

**UKArr\_C (continued)**

*#Set unallocated flights to E stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CUK\_Arr:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: \

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in E\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Arrive'] - pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Depart'] + pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() ==\

True and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == \

True and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and \

Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(E\_Pier)

*#Set unallocated flights to F stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CUK\_Arr:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.\

sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in F\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.\

empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

*(continued…)*

**UKArr\_C (continued)**

else:

Arrive = (Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Arrive'] - pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i,\

'Scheduled\_Timestamp\_Depart'] + pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all()==\

True and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() ==\

True and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and \

Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(F\_Pier)

**UKDep\_C():** Allocates the Domestic Departure, fleet C flights pier-served stands.

def **UKDep\_C():**

*#Set stand for first flight#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] == ind\_CUK\_Dep[0]:

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: \

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True)

df = df[df['Stand\_Arrive'].isin(C\_UK\_Dep)]

*#pick stand with largest number of passengers#*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(C\_UK\_Dep == stand)[0]

stand = np.int(C\_UK\_Dep[(index +1) % 6])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,'Stand\_Arrive']

else:

continue

*(continued…)*

**UKDep\_C (continued)**

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] == ind\_CUK\_Dep[0]:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

Turnarounds.loc[i, 'Stand\_Arrive'] = C\_UK\_Dep[0]

#Departure Stand

Turnarounds.loc[i,’Stand\_Depart'] = Turnarounds.loc[i,'Stand\_Arrive']

continue

*#Set stands for rest of the flights#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CUK\_Dep[1:]:

df = Turnarounds[Turnarounds.index.isin(ind\_CUK\_Dep)]

df = df.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df[df['Stand\_Arrive'].isin(C\_UK\_Dep)]

*#pick stand with largest number of passengers#*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] =\

Turnarounds.loc[i,'Stand\_Arrive']

break

else:

index = np.where(C\_UK\_Dep == stand)[0]

stand = np.int(C\_UK\_Dep[(index +1) % 6])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

break

C\_allo\_checker(C\_UK\_Dep)

*(continued…)*

**UKDep\_C (continued)**

*#Set unallocated flights to F-Remote Stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CUK\_Dep:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in F\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart']=Turnarounds.loc[i,'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] \

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True \

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True \

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(F\_Remote)

*#Set unallocated flights to D-Remote Stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CUK\_Dep:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in D\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.\

empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] \

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True\

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True \

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(D\_Remote)

**Int\_C:** Allocates the Domestic Departure, fleet C flights pier-served stands.

def **Int\_C():**

*#Set stand for first flight#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] == ind\_CInt[0]:

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= True)

df = df[df['Stand\_Arrive'].isin(C\_Int)]

*#pick stand with largest number of passengers#*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.\

loc[i, 'Scheduled\_Timestamp\_Arrive']> Turnarounds.\

loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] =\

Turnarounds.loc[i,'Stand\_Arrive']

break

else:

index = np.where(C\_Int == stand)[0]

stand = np.int(C\_Int[(index +1) % 10])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.\

loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.\

loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

*(continued…)*

**Int\_C (continued)**

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] == ind\_CInt[0]:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

Turnarounds.loc[i, 'Stand\_Arrive'] = C\_Int[0]

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] =Turnarounds.loc[i,'Stand\_Arrive']

continue

*#Set stands for rest of the flights#*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CInt[1:]:

df = Turnarounds[Turnarounds.index.isin(ind\_CInt)]

df = df.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df[df['Stand\_Arrive'].isin(C\_Int)]

*#pick stand with largest number of passengers#*

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',\

ascending= True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart']

>= pd.to\_timedelta('25 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

index = np.where(C\_Int == stand)[0]

stand = np.int(C\_Int[(index +1) % 10])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']

> Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart'] \

>= pd.to\_timedelta('25 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

C\_allo\_checker(C\_Int)

*(continued…)*

**Int\_C (continued)**

*# Set unallocated flights to pier-served 'F' stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CInt:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in F\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart']=\

Turnarounds.loc[i,'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] \

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True \

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(F\_Pier)

*# Set unallocated flights to remote 'F' stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CInt:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: \

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in F\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

#Departure Stand

Turnarounds.loc[i,'Stand\_Depart']=Turnarounds.loc[i,'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True\

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(F\_Remote)

*(continued…)*

**Section-D1** – The function for creating the daily allocation schedule.

**Int\_C (continued)**

*# Set unallocated flights to remote 'C' stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CInt:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in C\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] = \

Turnarounds.loc[i, 'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True \

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True \

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(C\_Remote)

*# Set unallocated flights to remote 'E' stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CInt:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g: \

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df.reset\_index()

for k in E\_Remote:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty == True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] =Turnarounds.loc[i,'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Depart'] \

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True\

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart'] = \

Turnarounds.loc[i,'Stand\_Arrive']

C\_allo\_checker(E\_Remote)

*(continued…)*

**Int\_C (continued)**

*# Set unallocated flights to pier-served 'E' stands #*

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_CInt:

if Turnarounds.loc[i, 'Stand\_Arrive'] == '':

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',\

ascending= False)

df = df.reset\_index()

for k in E\_Pier:

SchedArrivals = df.loc[df['Stand\_Arrive'] == k, \

'Scheduled\_Timestamp\_Arrive']

SchedDeparts = df.loc[df['Stand\_Arrive'] == k,\

'Scheduled\_Timestamp\_Depart']

if SchedArrivals.empty ==True and SchedDeparts.empty == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i,'Stand\_Depart']=\

Turnarounds.loc[i,'Stand\_Arrive']

break

else:

Arrive = (Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

- pd.to\_timedelta('25 minutes'))

Depart = (Turnarounds.loc[i,'Scheduled\_Timestamp\_Depart']\

+ pd.to\_timedelta('25 minutes'))

Arrival\_Check2 = (SchedArrivals < Arrive).all() == True \

and (Arrive < SchedDeparts).all() == True

Depart\_Check2 = (SchedArrivals < Depart).all() == True\

and (Depart < SchedDeparts).all() == True

if Arrival\_Check2 == False and Depart\_Check2 == False:

Turnarounds.loc[i, 'Stand\_Arrive'] = k

*#Departure Stand*

Turnarounds.loc[i, 'Stand\_Depart'] =\

Turnarounds.loc[i, 'Stand\_Arrive']

C\_allo\_checker(E\_Pier)

**Daily\_Sched**

def **Daily\_Sched**(date):

*#Run the allocation functions for each fleet, starting from F to C*

**Schedule\_F()**

**Schedule\_E()**

**Schedule\_D()**

**Schedule\_C()**

*#Save file*

Turnarounds.to\_csv( 'Allocation\_'+date+'.csv', index=False)

**Section-D2** – The function for creating the monthly allocation schedule.

**Monthly\_Sched**

def **Monthly\_Sched**(month):

global schedule,Schedule

*#update datatypes for the dates and times*

flight\_schedule[['Scheduled.Timestamp']] = \

flight\_schedule[['Scheduled.Timestamp']]\

.apply(pd.to\_datetime, dayfirst = True)

*#get terminal 5 data*

Schedule =flight\_schedule.loc[(flight\_schedule[\

'Scheduled.Timestamp'].dt.month == month) & \

(flight\_schedule['Terminal'] == 5)]

*#schedule for each day of the month*

for i, day in Schedule.groupby(\

Schedule['Scheduled.Timestamp'].dt.date):

schedule = Schedule.loc[\

(Schedule['Scheduled.Timestamp'].dt.date == i)]

**Sort\_cols()**

**Create\_Turns()**

**Turnaround()**

**Splitting()**

**final\_Turnarounds()**

**Daily\_Sched(i)**

*# save to new folder*

Turnarounds.to\_csv(r"C:/Users/User/Documents/Paba/DISS/July/"+ \

"Turnarounds"+str(i)+'.csv', index=False)

**Monthly\_Sched**

def **Monthly\_Sched**(month):

global schedule,Schedule

*#update datatypes for the dates and times*

flight\_schedule[['Scheduled.Timestamp']] = \

flight\_schedule[['Scheduled.Timestamp']]\

.apply(pd.to\_datetime, dayfirst = True)

*#get terminal 5 data*

Schedule =flight\_schedule.loc[(flight\_schedule[\

'Scheduled.Timestamp'].dt.month == month) & \

(flight\_schedule['Terminal'] == 5)]

*#schedule for each day of the month*

for i, day in Schedule.groupby(\

Schedule['Scheduled.Timestamp'].dt.date):

schedule = Schedule.loc[\

(Schedule['Scheduled.Timestamp'].dt.date == i)]

**Sort\_cols()**

**Create\_Turns()**

**Turnaround()**

**Splitting()**

**final\_Turnarounds()**

**Daily\_Sched(i)**

*# save to new folder*

Turnarounds.to\_csv(r"C:/Users/User/Documents/Paba/DISS/July/"+ \

"Turnarounds"+str(i)+'.csv', index=False)

hjghjgjkgkjhjkh

**Remote\_E:** Allocates E flights with long turnarounds to the remote stands

def **Remote\_E**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_ERemote:

if Turnarounds.index[i] == ind\_ERemote[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = E\_Remote[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_ERemote:

if Turnarounds.index[i] in ind\_ERemote[1:]:

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:

g.assign(col1\_sum=g.Total\_Pax\_Depart.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending=

False)

*#sort stand by the largest number of passengers served#*

df = df[df['Stand\_Arrive'].isin(E\_Remote)]

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='Scheduled\_Timestamp\_Depart',ascending=

True).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

*#Perform buffer check#*

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i, 'Stand\_Arrive']

break

else: *# pick next stand#*

index = np.where(E\_Remote == stand)[0]

stand = np.int(E\_Remote[(index +1) % 8])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check = Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i, 'Stand\_Arrive']

#break

else:

continue

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] =

Turnarounds.loc[i, 'Stand\_Arrive']

break

**Pier\_F:** Allocates all the flights to the Pier-served stands

def **Pier\_F**():

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_FPier:

if Turnarounds.index[i] == ind\_FPier[0]:

Turnarounds.loc[i, 'Stand\_Arrive'] = F\_Pier[0]

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i, 'Stand\_Arrive']

continue

for (i, row) in Turnarounds.iterrows():

if Turnarounds.index[i] in ind\_FPier:

if Turnarounds.index[i] in ind\_FPier[1:]:

df = Turnarounds.groupby('Stand\_Arrive').apply(lambda g:\

g.assign(col1\_sum=g.Total\_Pax\_Arrive.sum()))

df.index = df.index.droplevel(level=0)

df = df.sort\_values(by='Scheduled\_Timestamp\_Arrive',ascending= False)

df = df[df['Stand\_Arrive'].isin(F\_Pier)]

df[['Stand\_Arrive']] = df[['Stand\_Arrive']].apply(pd.to\_numeric)

df1 = df.drop\_duplicates('Stand\_Arrive', keep = 'first')

df1 = df1.sort\_values(by='col1\_sum',ascending= False).reset\_index()

for (j, rows) in df1.iterrows():

stand = df1['Stand\_Arrive'][j]

index1 = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =(Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] > Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'])

Buffer = (Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive'] - Turnarounds.loc[Turnarounds.index[index1],'Scheduled\_Timestamp\_Depart'] >= pd.to\_timedelta('30 minutes'))

if Buffer == True and Arrival\_Check == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,\

'Stand\_Arrive']

break

else:

index = np.where(F\_Pier == stand)[0]

stand = np.int(F\_Pier[(index +1) % 12])

if stand in np.array(df1['Stand\_Arrive']):

f = int(df1.loc[df1['Stand\_Arrive'] == stand, 'index'])

Arrival\_Check =Turnarounds.loc[i,'Scheduled\_Timestamp\_Arrive']\

> Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']

Buffer = Turnarounds.loc[i, 'Scheduled\_Timestamp\_Arrive']\

- Turnarounds.loc[Turnarounds.index[f],'Scheduled\_Timestamp\_Depart']\

>= pd.to\_timedelta('30 minutes')

if Arrival\_Check == True and Buffer == True:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

else:

pass

else:

Turnarounds.loc[i, 'Stand\_Arrive'] = stand

#Departure Stand

Turnarounds.loc[i, 'Stand\_Depart'] = Turnarounds.loc[i,

'Stand\_Arrive']

break