Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Jose Fernandez-Alcon

Scheduling Aircrafts

Student Thesis SA-2004-22
Summer Term 2004

Tutor: Dr. Alexander Hall

Supervisor:
Prof. Dr. Thomas Erlebach

2.7.2004

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Aircraft Scheduling

Fernández Alcón, José

August 10, 2004

Contents

List of Tables

List of Figures

1 Introduction

The name *Aircraft Scheduling* covers the interests of an airline of finding an assignment of the company's aircrafts to the flights that the company offers. This assignment should be done taking into account many constrains and trying to minimize the costs.

The main goal of this project is to develop a system for implementing and testing the heuristics of tail assignment and also a solution checker for the assignments.

1.1 An easy scenario

Imagine that we have only one aircraft in our airline, and we can only fly to one destination, (that means that we have two airports: *Home*, and *London*). Scheduling this scenario will not be difficult, our airplane has to fly from *Home* to *London*, and go back. The main problem here is to decide the time slots in which the airplane is going to fly. We can do this with paper and pencil.

1.2 Now a little bit more complicated

Let's introduce Paris as a new destination, this means that our model has three airports (*Home, London* and *Paris*). You may think that adding a new airport is not going to add that much complexity to the problem, but now we have a lot more possible combinations:

- Home → Paris → London → Home
- Home → London → Paris → Home
- Home → London → Home → Paris → Home
- Home → London → Home → London → Paris → Home
- Home → ...

If we add a second aircraft we have many more options, but also many more constrains. Trying to schedule this by hand may take very long.

1.3 What the problem is in reality

A real-world problem might contain all the possible destinations of a big airline, and also a large fleet of airplanes. The task to schedule this without computer aid will be quite hard. Imagine that we got a valid schedule for our airline, well done!. But what if one of our airplanes can not take off. It will not only miss that flight but also the other flights that it has to perform. If we do not have software to help us with the complex problems arising, we very likely will not be able to react properly, leading to large costs.

2 Some useful definitions

Some new terms needed to understand Aircraft scheduling.

Leg: A flight from one airport at another at a certain time.

Tail: An specific aircraft performing an certain leg.

Cluster: A set of consecutive legs.

Cycles: One take off and one landing.

Fleet: All the aircrafts owned by an airline.

Sub-fleet: Aircrafts of the same type belonging to the same fleet.

Aircraft version: Number of seats of each class that an specific aircraft has.

3 What Aircraft Scheduling is

Aircraft Scheduling is the name used to refer to the optimization problem which airlines have to deal with. It is to decide where, when, and which airplane has to fly. It can be divide into subfields:

- Long Term planning of schedule: Decide about the origin, destination, date and time of take off and landing.
- Fleet assignment: Assign sub-fleets to legs.
- Tail assignment: Assign specific aircrafts to the individual legs.
- Schedule recovery: How to respond to an anomaly in the schedule having the smallest costs possible.
- Crew scheduling: Assign crews to legs.

4 Description of the subproblems we are dealing with

We have data from an airline that we need to process in order to load a model that we can use to perform the scheduling operations. Once we get a scheduling solution we have to save it into a file, check that it is a valid solution, and also, compute the cost of this solution.

This is part of a larger project at TIK institute. The goals of the present part are loading the data model, create an input/output interface, and develop a solution checker. Meanwhile that the main goals of the other part[1] are the implementation of the optimizer and developing an objective function[2]

The data that we use in our software is a set of comma separated files, which means that we have a long string of alpha-numerical data. Each substring between two commas is a datum. If the datum that we need to retrieve is not a string, first it is necessary to convert it to the data type that we need.

ac_version.csv Data related to the version of the aircraft.

aircraft.csv Aircraft relevant data such the owner, the subtype, and standard version.

airport.csv Airport information necessary for the optimization.

check_rules_due.csv Data about when a given aircraft has to be checked.

leg.csv All the information necessary to describe a leg.

leg_pax.csv Number of passengers of each class flying in each leg.

minimum_groundtime.csv Minimum time that an aircraft needs to remain on the ground.

Development environment

In the development of this software we use a quite standard environment, plus two proprietary libraries.

- Operating Systems: Linux, Solaris.
- Programming language: C++

[1]This part is the content of the Diplomaarbeit of Peter Keller and Simon Schilling
[2]What an objective function is will be explained later

- Compiler: g++ 3.3
- Property Libraries
    1. LEDA 4.4
    2. QT 3.3 [3]

The LEDA library made our programming task much easier, because it includes a kind of upgraded version of some of the C++ standard classes that can be handled easier, and additionally many routines for handling graphs.

5 Loading the model

5.1 Explanation in plain text

5.1.1 Reading the files

The data that we have from *Lufthansa* is given in comma separated files. By reading each file line by line we create an **array of strings** for each line, dividing them using the comma as a separator. We then save these arrays into an *array of arrays of strings* creating a two-dimensional array.

Example file content:

```
Dulles,Washington,0000,2359
Barajas,Madrid,0000,2359
...
```

Resulting two input array:

```
<<Dulles,Washington,0000,2359>,<Barajas,Madrid,0000,2359>,<...>,...>
```

Figure 1: Example: Creating a 2D array from a file.

In a general description of how we load the data into the model, we can say that there are two phases:

1. Getting the data. This can be done in two different ways. In the first one, we give two parameters to a 2D-table, and retrieve the data that

[3]In this project we used the free version of QT, available at www.troltech.com

Example:

If we want to create the object Dulles, which is in the table *Airports*, we will do something like:

```
Name:=Airports[1][1];
Location:=Airports[1][2];
OpeningTime:=Airports[1][3];
...
```

Figure 2: Loading objects

we want. In the second one, we use a routine called *loadRow* to parse the data directly from the strings.

2. Linking the references. Many of the member variables of the classes are references to objects of other classes, or even to arrays or lists of objects. In this phase we have to create these links.

5.2 The classes

All the classes that we use in the model describe objects which do not have any special functions. They have just a function get() for every member variable to retrieve its value, and sometimes a set(), to change the value of the variable.

5.2.1 AInstance

One object of class AInstance is a whole model. Loading one object of class AInstance you have access to all the objects belonging to the model.

Its member variables are:

	Type	Description
airports	array of AAirport	All the airports used in the model
versions	array of AVersion	The aircraft versions of the model
aircrafts	array of AAircraft	The aircrafts to be scheduled
subFleets	array of ASubFleet	The subFleets
clusters	array of ACluster	The clusters to be flown

Example:

Here we have a line of an example file describing a leg:

```
Dulles, 170220051245, Madrid, 170220051922, ...
```

This line says that the fight departs from Dulles airport on the February the 17th 2005 at 12:45, and arrives at Madrid the same day at 19:22

In the first phase the following will occur:
The function loadRow is called. It reads the line and create an object of class ALeg:

```
pDepartureAp= not set
departureTime= 17th, February, 2005, 12:45
pArrivalAp= not set
arrivalTime= 17th, February, 2005, 19:22
...
```

In this first phase we have also created all the objects of class AAirport. We will focus on two of them:

Object Dulles

```
Name= Dulles
Location= Washington
OpeningTime= 00:00
...
```

Object Barajas

```
Name=Barajas
Location= Madrid
OpeningTime= 00:00
...
```

After the second phase the object ALeg will look like:

```
pDepartureAp= => Object Dulles
departureTime= 17th, February, 2005, 12:45
pArrivalAp= => Object Barajas
arrivalTime= 17th, February, 2005, 19:22
```

Figure 3: The two phases of the load

5.2.2 AAircraft

This class defines the objects that represent the airplanes.

Its member variables are:

	Type	**Description**
id	int	Identification number.
pSubFleet	ASubFleet*	Pointer to the subfleet the aircraft belongs to.
pStdVersion	AVersion*	Pointer to the standard version of the aircraft.

5.2.3 AAirport

The objects of class *AAirport* represents airports used in the model.

Its member variables are:

	Type	**Description**
id	int	Identification number.
codeIATA	String	International identification code of the airport used by IATA.
codeICAO	String	same as above but the one used by ICAO.
name	String	Name of the airport.
openFrom	QTime	Opening time.
openTo	QTime	Closing time.
longitude	String	Geographical location.
latitude	String	Geographical location.

5.2.4 ALeg

A leg is a flight from one airport to another at a certain time.

This class has also an special method called *loadRow* whose task is to set the values for all the member variables in the object. None of the variables of this class has a *set* method.

Its member variables are:

	Type	**Description**
id	int	Identification number.
pDepartureAP	AAirport*	Indicates the departure airport.
pArrivalAP	AAirport*	Indicates the arrival airport.
departureTime	QDateTime	Time and date of the departure.
arrivalTime	QDateTime	Time and date of the arrival.
flightTime	int	Duration of the flight.
paxFirst	int	Number of First class passengers.
paxBusi	int	Number of Business class passengers.
paxEcon	int	Number of Economy class passenger.
legState	String	State of the leg.
legType	String	Type of the leg.

5.2.5 ACluster

A cluster is a set of consecutive legs that are going to be flown by the same aircraft.

Its member variables are:

	Type	**Description**
id	int	Identification number.
legs	AArray<ALeg>	All the legs that conform the cluster.
validACPenalties	AArray<double>	For each valid aircraft there is penalty.
validAircraftPs	AArray<AAircraft*>.	List the aircraft that can fly this cluster.
maintenance	bool	The cluster can be a maintenance flight, so it is not a commercial cluster.
pFixedAircraft	AAircraft*	Aircraft assigned to the cluster.
checkCode	String	This variable is only set if maintenance is true.

This class has three extra functions:

loadRow: Sets the values of some variables of the object.

getFirstLeg: Returns a pointer to the first leg of the cluster. This method is for making simplify checking the validity of the solution.

getLastLeg: Returns a pointer to the last leg of the cluster. It is a useful method because of the same reason as above.

5.2.6 AVersion

An aircraft version is the number of seats in first, business, and economy class that the aircraft has.

Its member variables are:

	Type	Description
id	int	Identification number.
seatsFirst	int	Number of First class seats.
seatsBusi	int	Number of Business class seats.
seatsEcon	int	Number of Economy class seats.
name	string	Name of the version.
subType	string	Aircraft subtype this version belongs to.

5.2.7 ASubFleet

A subfleet is all the aircrafts of the same type which belong to the same fleet. A fleet is all the aircrafts owned by an airline.

Its member variables are:

	Type	Description
id	int	Identification number.
owner	String	Aircraft owner.
subType	String	Sub-type of the aircraft.
subtypeName	String	Name of the sub-type.
aircraftPs	AArray<AAircraft*>	All the aircrafts of the sub-fleet.
versionPs	AArray<AVersion*>	Versions of the aircrafts.
checkCodes	AArray<string>	Types of checks to be perform.
flightTimeUntilCheck	AArray<int>	Flight times until next check.
cyclesUntilCheck	AArray<int>	Cyles until next check.

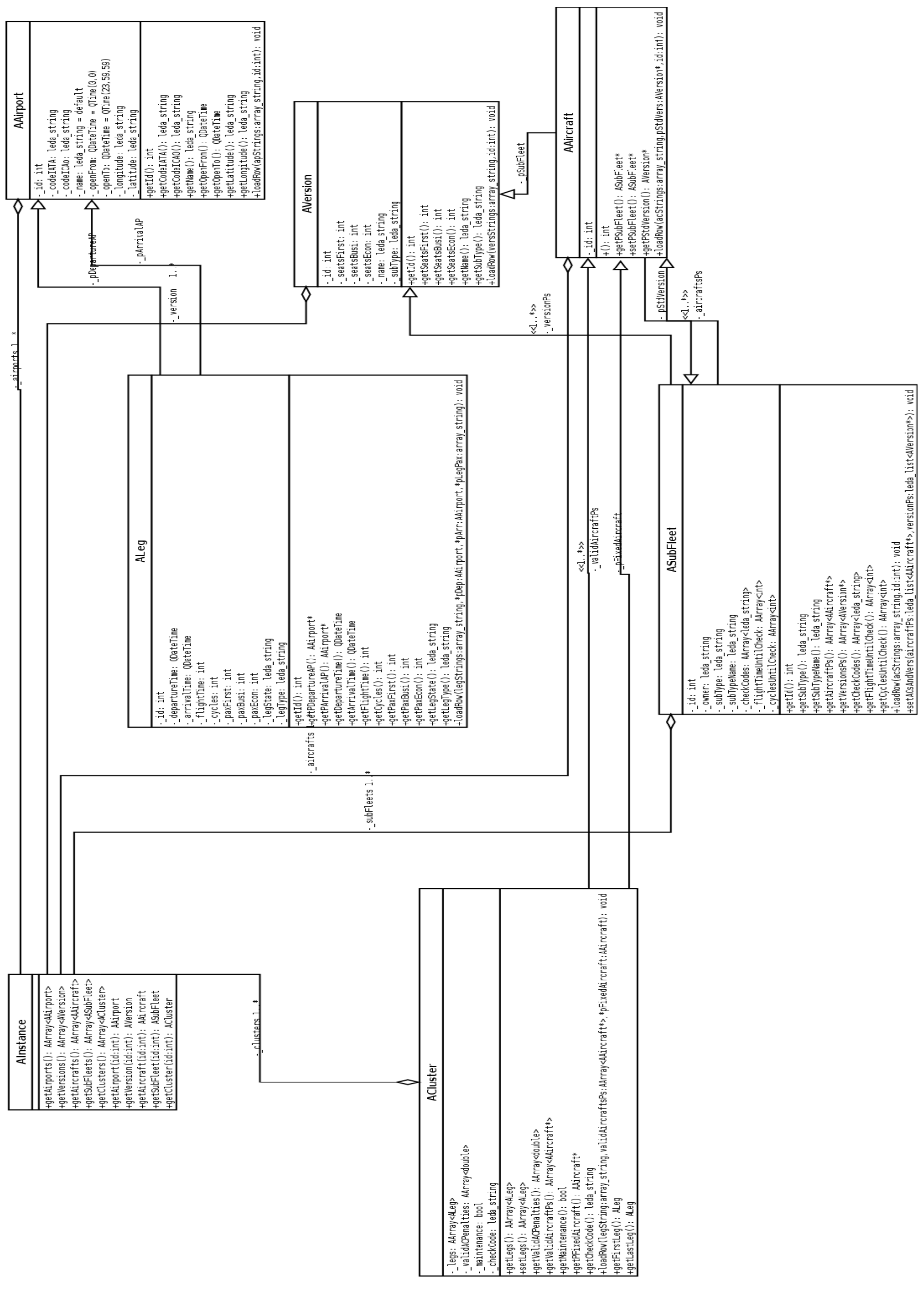
AAirport
- _id: int
- _codeIATA: leda_string
- _codeICAO: leda_string
- _name: leda_string = default
- _openFrom: QDateTime = QTime(0,0)
- _openTo: QDateTime = QTime(23,59,59)
- _longitude: leda_string
- _latitude: leda_string
+getId(): int
+getCodeIATA(): leda_string
+getCodeICAO(): leda_string
+getName(): leda_string
+getOpenFrom(): QDateTime
+getOpenTo(): QDateTime
+getLatitude(): leda_string
+getLongitude(): leda_string
+loadRow(apStrings:array_string,id:int): void
AVersion
- _id: int
- _seatsFirst: int
- _seatsBusi: int
- _seatsEcon: int
- _name: leda_string
- _subType: leda_string
+getId(): int
+getSeatsFirst(): int
+getSeatsBusi(): int
+getSeatsEcon(): int
+getName(): leda_string
+getSubType(): leda_string
+loadRow(versStrings:array_string,id:int): void
AAircraft
- _id: int
+(): int
+getPSubFleet(): ASubFleet*
+setPSubFleet(): ASubFleet*
+getPStdVersion(): AVersion*
+loadRow(acStrings:array_string,pStdVers:AVersion*,id:int): void
ALeg
- _id: int
- _departureTime: QDateTime
- _arrivalTime: QDateTime
- _flightTime: int
- _cycles: int
- _paxFirst: int
- _paxBusi: int
- _paxEcon: int
- _legState: leda_string
- _legType: leda_string
-getId(): int
-getPDepartureAP(): AAirport*
-getPArrivalAP(): AAirport*
-getDepartureTime(): QDateTime
-getArrivalTime(): QDateTime
-getFlightTime(): int
-getCycles(): int
-getPaxFirst(): int
-getPaxBusi(): int
-getPaxEcon(): int
-getLegState(): leda_string
-getLegType(): leda_string
-loadRow(legStrings:array_string,*pDep:AAirport,*pArr:AAirport,*pLegPax:array_string): void
ASubFleet
- _id: int
- _owner: leda_string
- _subType: leda_string
- _subTypeName: leda_string
- _checkCodes: AArray<leda_string>
- _flightTimeUntilCheck: AArray<int>
- _cyclesUntilCheck: AArray<int>
+getId(): int
+getSubType(): leda_string
+getSubTypeName(): leda_string
+getAircraftPs(): AArray<AAircraft*>
+getVersionsPs(): AArray<AVersion*>
+getCheckCodes(): AArray<leda_string>
+getFlightTimeUntilCheck(): AArray<int>
+getCyclesUntilCheck(): AArray<int>
+loadRow(acStrings:array_string,id:int): void
+setACsAndVers(aircraftPs:leda_list<AAircraft*>,versionPs:leda_list<AVersion*>): void
AInstance
+getAirports(): AArray<AAirport>
+getVersions(): AArray<AVersion>
+getAircrafts(): AArray<AAircraft>
+getSubFleets(): AArray<ASubFleet>
+getClusters(): AArray<ACluster>
+getAirport(id:int): AAirport
+getVersion(id:int): AVersion
+getAircraft(id:int): AAircraft
+getSubFleet(id:int): ASubFleet
+getCluster(id:int): ACluster
ACluster
- _legs: AArray<ALeg>
- _validACPenalties: AArray<double>
- _maintenance: bool
- _checkCode: leda_string
+getLegs(): AArray<ALeg>
+setLegs(): AArray<ALeg>
+getValidACPenalties(): AArray<double>
+getValidAircraftPs(): AArray<AAircraft*>
+getMaintenance(): bool
+getPFixedAircraft(): AAircraft*
+getCheckCode(): leda_string
+loadRow(legString:array_string,validAircraftsPs:AArray<AAircraft*>,*pFixedAircraft:AAircraft): void
+getFirstLeg(): ALeg
+getLastLeg(): ALeg
- _airports 1..*
- _pDepartureAP
- _pArrivalAP
- _version 1..*
- _pSubFleet
<<1..*>>
- _versionPs
- _pStdVersion
<<1..*>>
- _aircraftsPs
- _aircrafts
- _subFleets 1..*
<<1..*>>
- _validAircraftPs
- _pFixedAircraft
- _clusters 1..*

Figure 4: UML Diagram of the model

5.3 loadLHS

This is the class that actually loads all the data into the model. It reads the files, and parses the tables into two-dimensional arrays as described before.

5.3.1 numStrings

This function returns how many times a substring appears inside of an string.

input: The big string that we are analyzing. It is of type string.

selStr: The substring whose repetition we are calculating. It is also of type string.

A small example:

input=francekkkkfrance
selStr=france

Then this function will return 2.

Figure 5: Example: Function numStrings

5.3.2 extractComa

Extract substrings using the comma as a separator. It is necessary to separate all the different data contained in a line of a comma separated file.

commaLine: The comma-separated string which is going to be divided into substrings.

tokensInArray: The output of the function. It is an array of the substrings of the comma separated line.

5.3.3 createDataArray

The task of this method it to create a list with all the lines contained in a file. Each line is a different list entry.

inputFileStr: It is the input file we are dividing into lines. It is of type String

lines: It is the output list.

5.3.4 createTable

CreateTable uses the functions described above to transform a file into the two-dimensional array explained in the first paragraphs.

inputFileStr: It is a string that tells where the file is.

arr: This is the 2D array that this function returns.

6 Saving the solutions

Once we get a solution we need to store it and read it. We decided to use the XML to store the solution, because, nowadays, it is the standard for data storage.

6.1 Explanation in plain text

The solution of the optimization is given in an array of integers, in which the index number represents the identifier of a cluster, and the integer stored the id. of the aircraft which is going to flight the cluster. The clusters in this array have the same order as in AInstace.

6.2 The class ASolution

The main function of an object of this class is to write and to read the solutions into/from an XML file

6.2.1 writeXML

Writing the solution into an XML file, is quite simple, first, we create a *QDomDocument* object, then we set the root of the document to <*TAILS*> and, executing a loop, for each of the integers of the solution array we create the children tags by calling the function *createXmlTail*. Once we have finished with the array we save the *QDomDocument* object into a file.

This function has only one argument. It is a string which tells the function the file name of the file to be written. If no path is given, it will write the document in the same directory as the executable is.

6.2.2 createXmlTail

This method has been created is to perform the task of producing a XML document departing from the array of integers in which the solution is stored.

This function appends the children tags <*TAIL*> to the root tag. It has three arguments:

doc: It is the *QDomDocument* object we are working with.

root: It is the *QDomElement* root tag inside *doc*. It is the element to which we want to append the children.

value: The value of the tag that we are going to append.

6.2.3 readXML

Reading the solution from an XML file works the other way around. First, we read the file, and then we transfer its contents to a *QDomDocument* object, afterwards, we find the root <*TAILS*> in this object, and then, by calling the function readXMLDoc we get an array with the values of the children <*TAIL*>.

6.2.4 readXMLDoc

Once the root tag is parsed as an argument to this function, it executes a loop that reads all the children of the root and saves them into an array of integers.

root: It is a *QDomDocument* object. The function has to read all the children of this object.

We need to parse to this function the file to be read.

An small solution could look like this:

```
solution[1]=3
solution[2]=6
solution[3]=34
solution[4]=9
solution[5]=12
```

Then our XML file will be like this:

```
<TAILS>
<TAIL>3</TAIL>
<TAIL>6</TAIL>
<TAIL>34</TAIL>
<TAIL>9</TAIL>
<TAIL>12</TAIL>
</TAILS>
```

Figure 6: Example: Solution to XML file

7 Checking the solutions

Although we are really good programmers, we need to check that the obtained solution is valid. We have to verify:

- An aircraft has to depart from the last airport in which it has landed. *If an airplane has landed in London the next flight of this airplane has to take off from London*
- The departure time of an aircraft from an airport must be later that its arrival time to this airport. *A solution can not be valid if an aircraft which is supposed to take off from Athens at 08:30 is only arriving there at 09:00.*
- An aircraft can not be assigned to more that one leg at he same time. *Airplanes don't have the gift to be in different places at the same time*

7.1 How do we check the solutions

To be able to check our schedule, we first transform the solution from an array of integers to an array of arrays of integers. In this new array the

Example:

```
solution[1]=4
solution[2]=3
solution[3]=4
solution[4]=2
solution[5]=2
solution[6]=2
solution[7]=1
solution[8]=1
solution[9]=4
.
.
.
solution[87]=23
solution[88]=4
solution[89]=24
solution[90]=23
```

That means:

Aircraft	Which clusters is it flying
1	7,8
2	4,5,6
3	2
4	1,3,9,88
23	87,90
24	89

Figure 7: Example: What a Solution means

Example:

```
aircraft[1]=<7,8>
aircraft[2]=<4,5,6>
aircraft[3]=<2>
aircraft[4]=<1,3,9,88>
aircraft[23]=<87,90>
aircraft[24]=<89>
```

Figure 8: Example: Checking the solution

index is the identifier of the aircraft, and the array of integers are the id's of the cluster that the aircraft is flying.

In this new array we have the route which was assigned to an aircraft. We know that the legs within a cluster are OK, so, the only thing that we need to test are the connections between clusters. So what we really need to check is:

- The departure airport of the first leg of the next cluster must be the same as the arrival airport of the last leg of the previous cluster.
- The departure time of the first leg of the next cluster must later than the arrival time of the last leg of the previous cluster.

If these two conditions are fulfilled then the schedule is feasible.

7.2 The objective function

The parameters that we are trying to optimize[4] in this project are the costs[5] of the schedule. To calculate the cost of each scheduling solution is it necessary to compute the *Objective function.*

To calculate the objective function, we have assigned penalties to actions that we consider relevant. Then to compute the final value of this function, for each of these actions we calculate how many times this action has been taken and multiply it times its penalty. Once we have done this, we add all these results.

In our objective function costs are allocated in terms of:

[4]The optimization is performed by the software that Simon Schilling and Peter Keller are developing for their Diplomaarbeit.

[5]As these costs are not easy to assess, we have defined an easy to compute, but still realistic, objective function.

- Passengers not served
- Empty seats
- Others

Costs descriptions:

Passengers not served	Value of ω_i
First	12
Business	8
Economy	5

Table 1: Costs of not serving passengers

Empty seats	Value of ω_i
First	3
Business	2
Economy	1

Table 2: Costs of having empty seats

Objective function:

$$cost = \sum_i a_i \omega_i$$

7.3 The classes

7.3.1 Solchk

This module checks the solution and calculates its cost.

For checking the solution, *solchk* loads an object of class AInstance, then transforms the solution array into the array with the routes of each airplane, as explained above. Later we call the function *CheckAllRoutes* which obtains the array of routes as arguments.

Miscellaneous	Value of ω_i
Version change	50
Cycles violation	75
Fly time violation	5

Table 3: Costs of miscellaneous

Example:

Solution A: *In this solution we have only two First class passenger which are not served, so the total cost will be:*

$$Total\_cost = 2 \times 12 = 24$$

Solution B: *Here we have three empty seats in Business class, but we also have to change one aircraft version.*

$$Total\_cost = 3 \times 2 + 1 \times 50 = 56$$

Note: Here an optimization algorithm would choose *Solution A* because it is cheaper.

Figure 9: Example: Computing the objective function

In order to calculate the cost, *solchk* calls the function *costs Schedule*, which returns an integer with the value of the total cost of the schedule. How this function works will be explained later.

The functions whose name begins with *check-* are used for checking the feasibility of the solution, and the functions whose name begins with *costs-* are used for computing the costs of the solution.

7.3.2 CheckAllRoutes

This function returns a boolean value, *true* if the schedule is feasible, and *false* otherwise. What we do is to check if the routes of each aircraft are OK by calling the function *CheckAircraftRoute*.

This function has two arguments:

- **aircraftRoute:** Is the array of array of integers which represents the routes of the aircrafts.
- **clusters:** An array with all objects of class ACluster. It is necessary for getting the clusters, because in aircraftRoute we only have the

identifiers of the clusters, not the real object.

7.3.3 CheckAircraftRoute

This function returns a boolean value. If the route is OK then it returns *true*, and false otherwise. We look for incompatibilities between to consecutive clusters. To do so we first order the array clustersId calling the function *orderClusters*. After that, we iteratively get an id from clustersIds and the its respective cluster. Once we have two consecutive clusters we check the two conditions explained in the beginning of this section. If both conditions are accomplished it returns true.

To check the whole schedule this function is called once per aircraft inside the function *CheckAllRoutes*.

The two arguments of this function are:

- **clustersIds:** It is an array of integers with the identifiers of the cluster that this aircraft is going to fly
- **clusters:** An array with all the objects of class ACluster.

7.3.4 OrderClusters

The array clustersId might not be given in the scheduled order. If that is the case, the solution checker will return that the schedule is unfeasible when it actually is. So, in order to prevent that, this function orders the array considering the departing time of the first leg of the cluster.To do this first, we get the ids of the clusters and store them into a priority queue. The parameter used to compare the priority is the departing time, so the last cluster will have the highest priority. Then we iteratively take the cluster with the lowest priority and save it into the new clustersId array.

- **arrayToOrder:** The array of integers representing clusters that we have to order.
- **clusters:** All the objects of class ACluster are stored in this array.

7.3.5 DateTime2int

The departing time of an object of class ALeg is of type *QDateTime*. The priority queue can not compare two QDateTime objects, so it can not assign priorities based on this data type. The function *dateTime2int* transforms a *QDateTime* into an *int* so that the priority queue can compare the two dates and assign a priority.

- **date:** The *QDateTime* data that is going to be converted into an integer.

7.3.6 CostsPerLeg

The function costsPerLeg calculates the costs of one tail. That means the costs of having an special aircraft flying one specific leg.

This function calculates the costs derived from the difference between the number of first class passengers of the leg and the number of first class seats in the aircraft, then we do the same with Business and Economy. The final value that this function gives back, is the addition of the three values computed before.

This function has two arguments:

- **leg:** It is an object of class ALeg. It is the leg whose cost we are calculating.
- **aircraft:** It is an object of class AAircraft. It represents the aircraft which is flying the leg above.

7.3.7 CostsPerCluster

The purpose of this function is to calculate the costs of assigning an aircraft to a cluster. We do this by calling the function *costsPerLeg* for each of the legs of the cluster, parsing the same aircraft as an argument all the times. It returns an integer with the cost.

Arguments:

- **cluster:** Object of class ACluster. It is the cluster whose cost we are computing
- **acs:** Object of class AAircraft. The aircraft that is flying the cluster.

7.3.8 CostsSchedule

CostsSchedule computes the objective function of the whole schedule. For each aircraft, it calls the function *CostsPerCluster* for each of the clusters which were assigned to the aircraft. It returns an integer with the cost of the whole schedule.

This function has three arguments:

- **assignment:** It if type *array_clusters_id* that we have defined in this class. This *typedef* is an array of arrays of integers. Assignment tells us which clusters are going to be flown by which aircraft.
- **acs:** An array with all the aircrafts of the model. We need it because in an assignment we only have the id of the aircraft, but not the real object.
- **cluster:** An array containing the clusters of the model. It is necessary because in assignment we have the id's of the cluster, and we need the objects.

8 Conclusions

This *Semmesterarbeit* is part of a larger project whose target is to create an optimal schedule for an airline. Our main goals have been to create the I/O system of the whole system. That means:

- To create and load a model using real data from an airline
- To store and load the solutions given by the optimizer.
- Analyze the feasibility and the cost of a solution.

We programed the software using C++ and some proprietary libraries.