|  |  |
| --- | --- |
| TALLINN UNIVERSITY OF TECHNOLOGY | |
| School of Information Technologies | |
|  | |
|  | |
| Jevgeni Fenko 200810IADB | |
| Homework 6 | |
| Flight Scheduling Problem (C-13.23, Data Structures and Algorithms in Java. Goodrich, Tamassia) | |
| Supervisor: | Jaanus Pöial |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jevgeni Fenko

19.04.2021

Table of contents

[Author’s declaration of originality 2](#_Toc69778186)

[Table of contents 3](#_Toc69778187)

[1 Task Description 4](#_Toc69778188)

[1.1 Task specification 4](#_Toc69778189)

[2 Description of proposed solution 6](#_Toc69778190)

[2.1 Classes Graph, Vertex, Arc 6](#_Toc69778191)

[2.2 Class Dijkstra – the algorithm 7](#_Toc69778192)

[3 . User Manual 11](#_Toc69778193)

[4 Testing 13](#_Toc69778194)

[4.1 Test 1 14](#_Toc69778195)

[4.2 Test 2 15](#_Toc69778196)

[4.3 Test 3 16](#_Toc69778197)

[4.4 Test 4 17](#_Toc69778198)

[4.5 Test 5 17](#_Toc69778199)

[References 23](#_Toc69778200)

[Appendix 1 – Program Code 24](#_Toc69778201)

# Task Description

Suppose you are given a timetable, which consists of:

A set A of n airports, and for each airport in A, a minimum connecting time c(a).

A set F of m flights, and the following, for each flight f in F:

Origin airport a1(f) in A

Destination airport a2(f) in A

Departure time t1(f)

Arrival time t2(f)

Describe an efficient algorithm for the flight scheduling problem. In this problem we are given airports a and b, and a time t, and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed.

## Task specification

Originally, we must count, if the minimum connecting time at intermediate airport is observed, however, the task has nothing about waiting time at airports and if we need to count them or not.

In common, it looks like usual path finding task with additional parameter c(a) – minimum connecting time. Beside this, the task is abstract and can be improved with implementation of finding not only shortest by time flight, but also shortest by time overall, including waiting hours at transit airports. Also, in real life, not all airports have connections between them or even bi-directional connections, and airports may be in different time zones, which makes additional difficulty in time calculation.

For more explanatory view we will use the following graphical observation of airports and connections. As if we would like to depart from Tallin on 1.06. as early as possible and arrive to Alicante as early as possible. The same example data is used in **class** **GraphTask** method **run**.

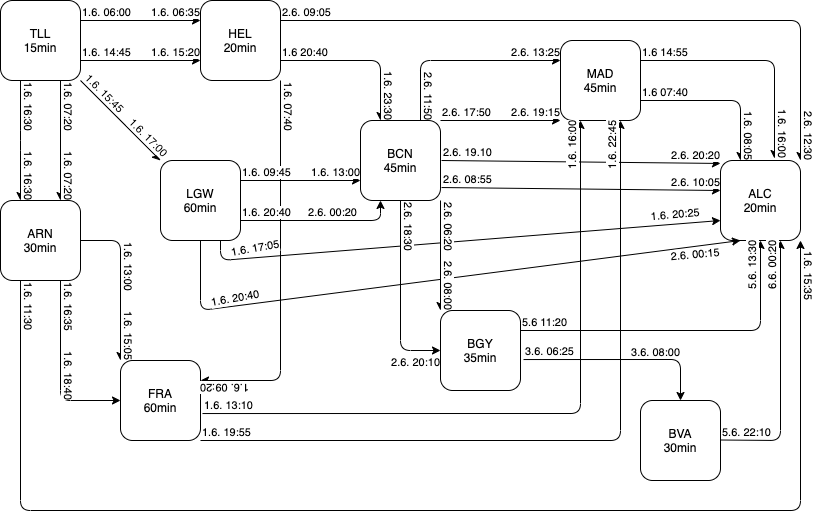


Figure 1. Network of flights to Alicante from Tallinn, with minimum connection times and departures, arrivals.

As you can see, all airports have different number of connections, different minimum connection time and, according to departure and arrival, different time zones.

# Description of proposed solution

For solution of the scheduling problem, we can use weighted directed graph and Dijkstra algorithm. However, in our situation, the algorithm for finding path will be slightly changed for indication of required flights. The Arcs of our Graph will have additional weight for indication of wating time in transit airport. We also can additionally sort the graph and remove flights with impossible connections, before finding the shortest flight from departure to destination.

## Classes Graph, Vertex, Arc

So, at first, we must convert data regarding flights into Graph.  
Our **class Graph** consists of List of Vertices (List of Airports) and List of Arcs (List of Flights between Airports). As in real life, it is not necessary, that all vertices have arcs, such approach used in method for generation of random Graphs for this task.

**static class** Graph {  
  
 **private** List<Vertex> **airports**;  
 **private** List<Arc> **flights**;

As you can see, the class Graph is very simple, we do not require additional properties for it.

The **class Vertex** (airports) has only one additional property, beside **String id**, is **Integer minConTime**, as we have minimum connection time in every airport, which we must observe, while using connections in transit airports.

**static class** Vertex {  
  
 **final private** String **id**;  
 **final private** Integer **minConTime**;

*/\*\* Constructor \*/*Vertex(String id, Integer minConTime) {  
 **this**.**id** = id;  
 **this**.**minConTime** = minConTime;  
}

The most complicated by properties class in the solution is the **class Arc** (flights), as it must have arrival and departure times, airports, to which the flight belongs, calculated flight time and, I used the class to hold the waiting time in transit airport as additional weight of vertex. The constructor also has two additional parameters to input:  
**Integer depTimeZone** – Time Zone for departure airport, **Integer arrTimeZone** – Time Zone for arrival airport.

**static class** Arc {  
  
 **final private** String **id**;  
 **final private** Vertex **departure**;  
 **final private** Vertex **arrival**;  
 **final private** Date **depTime**;  
 **final private** Date **arrTime**;  
 **private final** Integer **flightTime**;  
 **private** Integer **waitingTime** = 0;

*/\*\* Constructor \*/*Arc(String id, Vertex departure, Vertex arrival, String depTime, Integer depTimeZone, String arrTime, Integer arrTimeZone) {  
 DateFormat strToDate = **new** SimpleDateFormat(**"dd.MM.yyyy hh:mm"**);  
 **this**.**id** = id;  
 **this**.**departure** = departure;  
 **this**.**arrival** = arrival;  
 **try** {  
 **this**.**depTime** = strToDate.parse(depTime);  
 } **catch** (ParseException e) {  
 **throw new** RuntimeException(**"Wrong Date String Format, must be : dd.MM.yyyy hh:mm"** + depTime);  
 }  
 **try** {  
 **this**.**arrTime** = strToDate.parse(arrTime);  
 } **catch** (ParseException e) {  
 **throw new** RuntimeException(**"Wrong Date String Format, must be : dd.MM.yyyy hh:mm"** + arrTime);  
 }  
  
 **this**.**flightTime** = Math.*abs*((**int**) (**this**.**arrTime**.getTime() - **this**.**depTime**.getTime()) / 1000) / 60 - (arrTimeZone - depTimeZone) \* 60;  
}

## Class Dijkstra – the algorithm

The whole process of finding the solution after getting the Graph can be divided in 3 parts.

**Sorting.**

We already know that the schedule consists of flights, which cannot have connection between, if the connection time is less than minimum connection time at airport or, if the flight from transit airport departures earlier, than we will arrive to it. So we can purify the list of flights first, to comply to minimum connection time rule and leave only connectable flights. Therefore, I added **method sortGraph** to **class Dijkstra**, which founds all misconnections.

In our example, flight from Tallinn to Gatwick arrives 01.06. at 17:00, this means that flight from Gatwick to Barcelona with departure 01.06. at 09:45 can’t be used and can be removed before starting of algorithm.

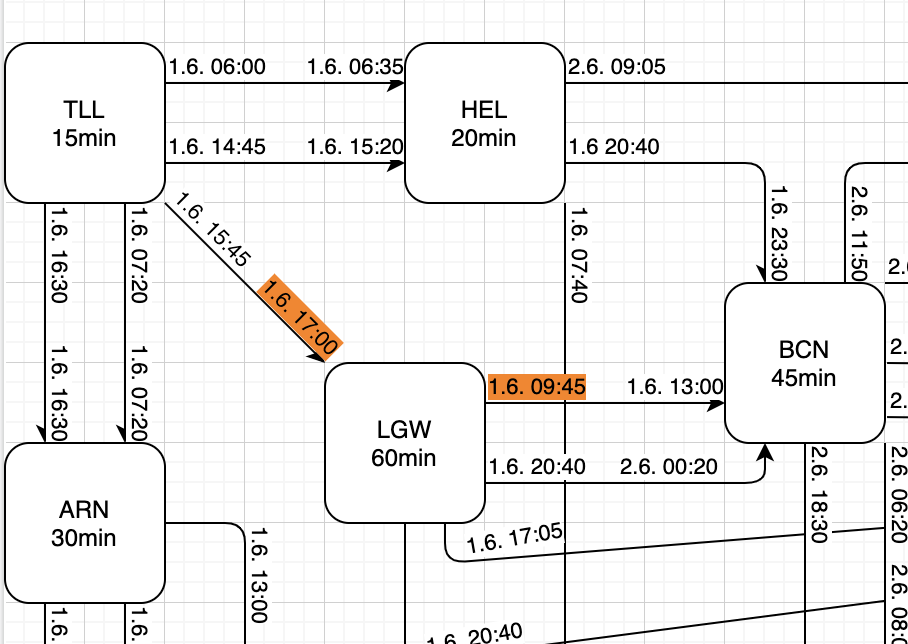


Figure 2. Misconnection at transit airport London Gatwick (LGW)

**Finding shortest path by Dijkstra.**

At the time, we run **method run** of **class Dijkstra**, after the sorting, the algorithm puts the first departure airport (our starting point) to unsettled nodes (unsettledAirports) and starts to find weight for every node, as we need not path, consisting of airports, but flights, in additional to every node, the Dijkstra puts used flight into separate **Map usedFlights**.

Taking literally, the work of algorithm may be described as follows:  
The starting airport is TLL, I will put it into **settledAirports**, as no connections directed to it must be checked.  
I need to check if there are any connected airports with TLL and calculate weights for every flight (right, most important for us is flight time and flights, not airports) and every node.

I put checked flights (green arrows on picture) in **usedFlights** with node (neigbour) as key, and neighbours in **unsettledAirports** and cumulative time to **time Map**.  
Now I need to check every neighbour and will start from HEL. Oh, there is a connection time between first flight from TLL to HEL and flight from HEL to ALC, I must add it to flight from HEL to ALC as waiting time in HEL and add flight (blue arrow on picture) to used Flights. The printed-out information will look like:  
*Waiting Time at HEL: 1590 min*

*HEL-ALC: Flight Duration: 575 min, Dep: Wed Jun 02 09:05:00 Arr: Wed Jun 02 00:30:00*

And so on, and so on… till the last connection between airports.

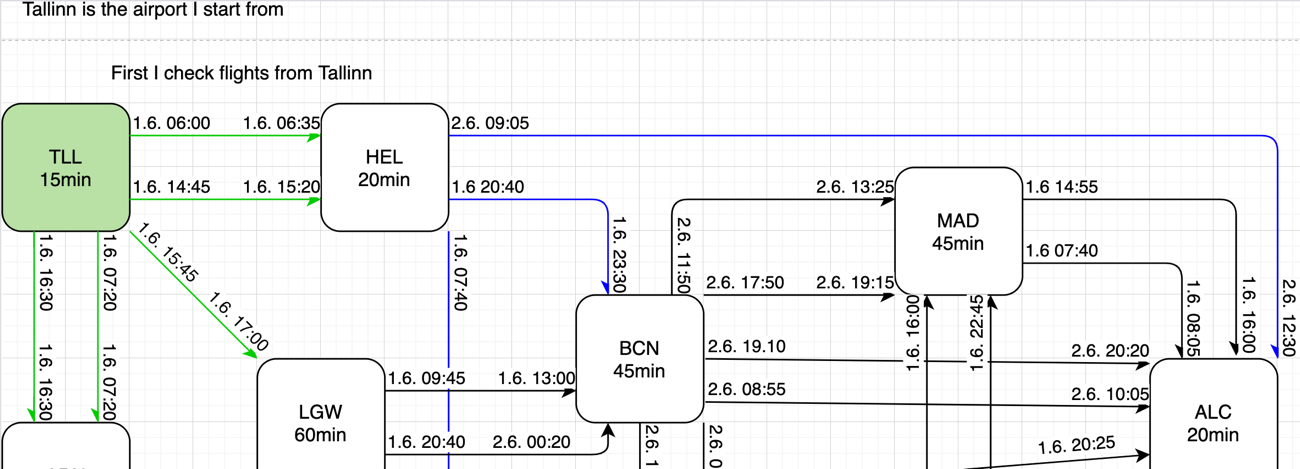


Figure 3. Dijkstra checks every flight from first airport TLL to neighbours and form neighbours to their neighbours.

**Reverting answer to user.**

After all manipulations with airports and flights, the Dijkstra class collects data for every node with cumulative time in **time Map**, connections between nodes with minimum weights, in **previousAirports Map**, and flights as objects with arrival airports as keys in **usedFlights Map**.

Taking this data, we can plot route and indicate flights by **method getRoute** of **class Dijkstra**. The method takes last arrival airport (finish) as parameter. If the airport has link to previous one in **previousAirports**, the method finds the flight, corresponding to this airport in **usedFlights**, and adds weight of flight to **totalTime** parameter of **Dijkstra class**, then takes linked airport and does the same stuff and so on till the first departure airport. As result, method returns List of best suitable flights. In print-out mode the result looks like:  
*TLL-ARN: Flight Duration: 60 min, Dep: 01.06.2021 07:20 Arr: 01.06.2021 07:20*

*Waiting Time at ARN: 250 min*

*ARN-ALC: Flight Duration: 245 min, Dep: 01.06.2021 11:30 Arr: 01.06.2021 15:35*

*Total time: 0 days, 9 hours and 15 minutes.*

You can check it on our example scheme:

**Изображение выглядит как текст, снимок экрана, черный, продажа

Автоматически созданное описание**

Figure 4. Returned best flight schedule on example graphics.

# . User Manual

To use the solution, we must add flight schedule as class Graph object.

Currently it is possible to add all airports and flights manually only.

1. Create list of airports (Vertices) in method run of class GraphTask and add required airports.

i.e.

List<Vertex> testAirports = **new** ArrayList<>();  
testAirports.add(**new** Vertex(**"TLL"**, 15)); *//0*testAirports.add(**new** Vertex(**"HEL"**, 20)); *//1*testAirports.add(**new** Vertex(**"ARN"**, 30)); *//2*testAirports.add(**new** Vertex(**"LGW"**, 60)); *//3*

1. Create list of flights (Arcs) in method run of class GraphTask and add required flights.

i.e.

List<Arc> testFlights = **new** ArrayList<>();  
testFlights.add(**new** Arc(**"TLL-HEL"**, testAirports.get(0), testAirports.get(1), **"01.06.2021 06:00"**, 2, **"01.06.2021 06:35"**, 2));  
testFlights.add(**new** Arc(**"TLL-HEL"**, testAirports.get(0), testAirports.get(1), **"01.06.2021 14:45"**, 2, **"01.06.2021 15:20"**, 2));  
testFlights.add(**new** Arc(**"TLL-LGW"**, testAirports.get(0), testAirports.get(3), **"01.06.2021 15:55"**, 2, **"01.06.2021 17:00"**, 0));  
testFlights.add(**new** Arc(**"TLL-ARN"**, testAirports.get(0), testAirports.get(2), **"01.06.2021 07:20"**, 2, **"01.06.2021 07:20"**, 1));  
testFlights.add(**new** Arc(**"TLL-ARN"**, testAirports.get(0), testAirports.get(2), **"01.06.2021 16:30"**, 2, **"01.06.2021 16:30"**, 1));

3) Create object of class Graph with already created Lists as parameters.

i.e.

Graph testGraph1 = **new** Graph(testAirports, testFlights);

1. Generate object of class Dijkstra from object of class Graph.

i.e.

Dijkstra dijkstra = **new** Dijkstra(testGraph1);

1. Run Dijkstra algorithm with run function of class Dijkstra taking index of first departure airport (start point) as parameter

i.e.

dijkstra.run(testGraph1.**airports**.get(0));

1. Get List of Arcs (flights) as result by running function getRoute of class Dijkstra, taking index of last arrival airport (finish point) as parameter. This is the shortest flight schedule.

i.e.

List<Arc> flights = dijkstra.getRoute(testGraph1.**airports**.get(9));

In case if flight not possible, the List<Arc> flights will be null.

Depending on user intentions, the list may be used for any other needs. In other case, we just print the result as String.

1. Additionally, to get total time of flight as String, use method totalTimeString of Dijkstra class.

i.e.

System.***out***.println(dijkstra.totalTimeString());

# Testing

Example Graph:

List<Vertex> testAirports = **new** ArrayList<>();  
testAirports.add(**new** Vertex(**"TLL"**, 15)); *//0*testAirports.add(**new** Vertex(**"HEL"**, 20)); *//1*testAirports.add(**new** Vertex(**"ARN"**, 30)); *//2*testAirports.add(**new** Vertex(**"LGW"**, 60)); *//3*testAirports.add(**new** Vertex(**"BCN"**, 45)); *//4*testAirports.add(**new** Vertex(**"FRA"**, 60)); *//5*testAirports.add(**new** Vertex(**"MAD"**, 45)); *//6*testAirports.add(**new** Vertex(**"BGY"**, 35)); *//7*testAirports.add(**new** Vertex(**"BVA"**, 30)); *//8*testAirports.add(**new** Vertex(**"ALC"**, 20)); *//9*List<Arc> testFlights = **new** ArrayList<>();  
testFlights.add(**new** Arc(**"TLL-HEL"**, testAirports.get(0), testAirports.get(1), **"01.06.2021 06:00"**, 2, **"01.06.2021 06:35"**, 2));  
testFlights.add(**new** Arc(**"TLL-HEL"**, testAirports.get(0), testAirports.get(1), **"01.06.2021 14:45"**, 2, **"01.06.2021 15:20"**, 2));  
testFlights.add(**new** Arc(**"TLL-LGW"**, testAirports.get(0), testAirports.get(3), **"01.06.2021 15:55"**, 2, **"01.06.2021 17:00"**, 0));  
testFlights.add(**new** Arc(**"TLL-ARN"**, testAirports.get(0), testAirports.get(2), **"01.06.2021 07:20"**, 2, **"01.06.2021 07:20"**, 1));  
testFlights.add(**new** Arc(**"TLL-ARN"**, testAirports.get(0), testAirports.get(2), **"01.06.2021 16:30"**, 2, **"01.06.2021 16:30"**, 1));  
  
testFlights.add(**new** Arc(**"HEL-ALC"**, testAirports.get(1), testAirports.get(9), **"02.06.2021 09:05"**, 2, **"02.06.2021 12:30"**, 1));  
testFlights.add(**new** Arc(**"HEL-BCN"**, testAirports.get(1), testAirports.get(4), **"01.06.2021 20:40"**, 2, **"01.06.2021 23:30"**, 1));  
testFlights.add(**new** Arc(**"HEL-FRA"**, testAirports.get(1), testAirports.get(5), **"01.06.2021 07:40"**, 2, **"01.06.2021 09:20"**, 1));  
  
testFlights.add(**new** Arc(**"LGW-BCN"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 09:45"**, 0, **"01.06.2021 13:00"**, 1));  
testFlights.add(**new** Arc(**"LGW-BCN"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 21:05"**, 0, **"02.06.2021 00:20"**, 1));  
testFlights.add(**new** Arc(**"LGW-ALC"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 17:05"**, 0, **"01.06.2021 20:25"**, 1));  
testFlights.add(**new** Arc(**"LGW-ALC"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 20:40"**, 0, **"02.06.2021 00:15"**, 1));  
  
testFlights.add(**new** Arc(**"ARN-FRA"**, testAirports.get(2), testAirports.get(5), **"01.06.2021 13:00"**, 1, **"01.06.2021 15:05"**, 1));  
testFlights.add(**new** Arc(**"ARN-FRA"**, testAirports.get(2), testAirports.get(5), **"01.06.2021 16:35"**, 1, **"01.06.2021 18:40"**, 1));  
testFlights.add(**new** Arc(**"ARN-ALC"**, testAirports.get(2), testAirports.get(9), **"01.06.2021 11:30"**, 1, **"01.06.2021 15:35"**, 1));  
  
testFlights.add(**new** Arc(**"BCN-MAD"**, testAirports.get(4), testAirports.get(6), **"02.06.2021 11:50"**, 1, **"02.06.2021 13:25"**, 1));  
testFlights.add(**new** Arc(**"BCN-MAD"**, testAirports.get(4), testAirports.get(6), **"02.06.2021 17:50"**, 1, **"02.06.2021 19:15"**, 1));  
testFlights.add(**new** Arc(**"BCN-BGY"**, testAirports.get(4), testAirports.get(7), **"02.06.2021 06:20"**, 1, **"02.06.2021 08:00"**, 1));  
testFlights.add(**new** Arc(**"BCN-BGY"**, testAirports.get(4), testAirports.get(7), **"02.06.2021 18:30"**, 1, **"02.06.2021 20:10"**, 1));  
testFlights.add(**new** Arc(**"BCN-ALC"**, testAirports.get(4), testAirports.get(9), **"02.06.2021 08:55"**, 1, **"02.06.2021 10:05"**, 1));  
testFlights.add(**new** Arc(**"BCN-ALC"**, testAirports.get(4), testAirports.get(9), **"02.06.2021 19:10"**, 1, **"02.06.2021 20:20"**, 1));  
  
testFlights.add(**new** Arc(**"FRA-MAD"**, testAirports.get(5), testAirports.get(6), **"01.06.2021 13:10"**, 1, **"01.06.2021 16:00"**, 1));  
testFlights.add(**new** Arc(**"FRA-MAD"**, testAirports.get(5), testAirports.get(6), **"01.06.2021 19:55"**, 1, **"01.06.2021 22:45"**, 1));  
  
testFlights.add(**new** Arc(**"MAD-ALC"**, testAirports.get(6), testAirports.get(9), **"02.06.2021 07:40"**, 1, **"02.06.2021 08:50"**, 1));  
testFlights.add(**new** Arc(**"MAD-ALC"**, testAirports.get(6), testAirports.get(9), **"02.06.2021 14:55"**, 1, **"02.06.2021 16:00"**, 1));  
  
testFlights.add(**new** Arc(**"BGY-ALC"**, testAirports.get(7), testAirports.get(9), **"05.06.2021 11:20"**, 1, **"05.06.2021 13:30"**, 1));  
testFlights.add(**new** Arc(**"BGY-BVA"**, testAirports.get(7), testAirports.get(8), **"03.06.2021 06:25"**, 1, **"03.06.2021 08:00"**, 1));  
  
testFlights.add(**new** Arc(**"BVA-ALC"**, testAirports.get(8), testAirports.get(9), **"05.06.2021 22:10"**, 1, **"06.06.2021 00:20"**, 1));  
  
Graph testGraph1 = **new** Graph(testAirports, testFlights);

## Test 1

Finding shortest flight by time from first Vertex to last Vertex in our example.

Input:

Dijkstra dijkstra = **new** Dijkstra(testGraph1);  
dijkstra.run(testGraph1.**airports**.get(0));  
  
List<Arc> flights = dijkstra.getRoute(testGraph1.**airports**.get(9));  
  
**try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
} **catch** (NullPointerException e) {  
 System.***out***.println(**"No connection between "** + testGraph1.**airports**.get(9) + **" and "** + testGraph1.**airports**.get(9) + **"\n"**);  
}

Output:

Изображение выглядит как текст

Автоматически созданное описание

## Test 2

Finding shortest flight by time from first Vertex to last Vertex in our example, if ARN-ALC flight is missing.

Commented out arc:

*testFlights.add(new Arc("ARN-ALC", testAirports.get(2), testAirports.get(9), "01.06.2021 11:30", 1, "01.06.2021 15:35", 1));*

Input:

Dijkstra dijkstra = **new** Dijkstra(testGraph1);  
dijkstra.run(testGraph1.**airports**.get(0));  
  
List<Arc> flights = dijkstra.getRoute(testGraph1.**airports**.get(9));  
  
**try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
} **catch** (NullPointerException e) {  
 System.***out***.println(**"No connection between "** + testGraph1.**airports**.get(9) + **" and "** + testGraph1.**airports**.get(9) + **"\n"**);  
}

Output

Изображение выглядит как текст

Автоматически созданное описание

## Test 3

Finding shortest flight by time from any Vertex to any Vertex in our example, except start and end Vertexes of Graph.

Input:

Dijkstra dijkstra = **new** Dijkstra(testGraph1);  
dijkstra.run(testGraph1.**airports**.get(3));  
  
List<Arc> flights = dijkstra.getRoute(testGraph1.**airports**.get(8));  
  
**try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
} **catch** (NullPointerException e) {  
 System.***out***.println(**"No connection between "** + testGraph1.**airports**.get(3) + **" and "** + testGraph1.**airports**.get(8) + **"\n"**);  
}

Output:

Изображение выглядит как текст

Автоматически созданное описание

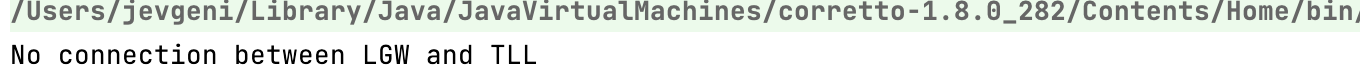
## Test 4

Finding shortest flight by time from any Vertex to any Vertex if no flight connection between them.

Input:

Graph testGraph1 = **new** Graph(testAirports, testFlights);  
  
Dijkstra dijkstra = **new** Dijkstra(testGraph1);  
dijkstra.run(testGraph1.**airports**.get(3));  
  
List<Arc> flights = dijkstra.getRoute(testGraph1.**airports**.get(0));  
  
**try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
} **catch** (NullPointerException e) {  
 System.***out***.println(**"No connection between "** + testGraph1.**airports**.get(3) + **" and "** + testGraph1.**airports**.get(0) + **"\n"**);  
}

Output:



## Test 5

Tests with automatically generated Graphs. As in real life there are no connections to every airport, also the Graphs generated with no connections to part of Vertices. Tests lasts till first successful connection result.  
In case, if no connection, tests, start another run automatically. Besides connections results, tests measure time required for Graph generation and Dijkstra Algorithm run and print out average consumed time.

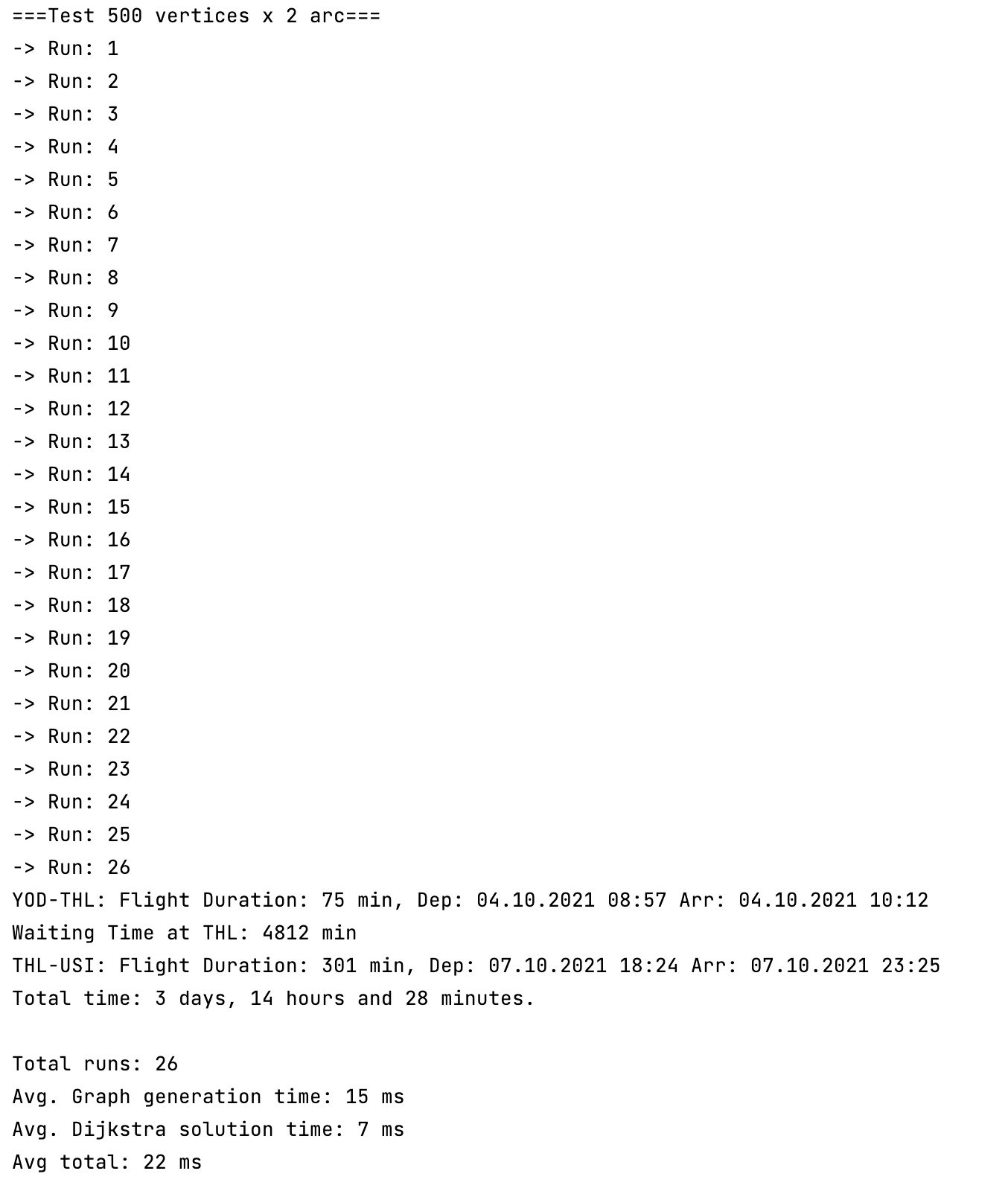
Tests located in GraphTaskTest.java file and use method testRun of GraphTask class.

**public void** testRun(String testName, **int** n, **int** m, **int** start, **int** finish) {  
 List<Arc> flights;  
 **long** graphStart;  
 **long** graphEnd;  
 **long** graphSum = 0;  
 **long** dijkstraStart;  
 **long** dijkstraEnd;  
 **long** dijkstraSum = 0;  
 **int** run = 0;  
  
 System.***out***.println(testName);  
  
 **do** {  
 run += 1;  
 System.***out***.println(**"-> Run: "** + run);  
 graphStart = System.*currentTimeMillis*();  
 Graph testGraph2 = **new** Graph();  
 testGraph2.createRandomGraph(n, m);  
 graphEnd = System.*currentTimeMillis*() - graphStart;  
 graphSum += graphEnd;  
 *//System.out.println("Graph generated in:" + graphEnd + " ms");  
 //System.out.println("Dijkstra started\n");* dijkstraStart = System.*currentTimeMillis*();  
 Dijkstra dijkstra = **new** Dijkstra(testGraph2);  
 dijkstra.run(testGraph2.**airports**.get(start));  
 flights = dijkstra.getRoute(testGraph2.**airports**.get(finish));  
 dijkstraEnd = System.*currentTimeMillis*() - dijkstraStart;  
 dijkstraSum += dijkstraEnd;  
 **try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
 } **catch** (NullPointerException e) {  
 *// System.out.println("No connection between " + testGraph2.airports.get(start) + " and " + testGraph2.airports.get(finish) + "\n");* }  
  
 *//System.out.println("Dijkstra algorithm finished in: " + dijkstraEnd + " ms\n");* } **while** (flights == **null**);  
  
 System.***out***.println(**"Total runs: "** + run);  
 System.***out***.println(**"Avg. Graph generation time: "** + graphSum / run + **" ms"**);  
 System.***out***.println(**"Avg. Dijkstra solution time: "** + dijkstraSum / run + **" ms"**);  
 System.***out***.println(**"Avg total: "** + (graphSum / run + dijkstraSum / run) + **" ms"**);  
}

GraphTaskTest.java

**import static** org.junit.Assert.\*;  
**import** org.junit.Test;  
  
*/\*\* Testklass.  
 \** ***@author*** *jaanus  
 \*/***public class** GraphTaskTest {  
  
 @Test  
 **public void** test1() {   
 GraphTask a = **new** GraphTask();  
 a.testRun(**"===Test 500 vertices x 2 arc==="**, 500, 2, 0, 499);  
 System.***out***.println(**"\n"**);  
 *assertTrue* (**"There are no wrong answers"**, **true**);  
 }  
  
 @Test  
 **public void** test2() {  
 GraphTask a = **new** GraphTask();  
 a.testRun(**"===Test 1000 vertices x 4 arc==="**, 1000, 4, 0, 999);  
 System.***out***.println(**"\n"**);  
 *assertTrue* (**"There are no wrong answers"**, **true**);  
 }  
  
 @Test  
 **public void** test3() {  
 GraphTask a = **new** GraphTask();  
 a.testRun(**"===Test 1500 vertices x 6 arc==="**, 1500, 6, 0, 1499);  
 System.***out***.println(**"\n"**);  
 *assertTrue* (**"There are no wrong answers"**, **true**);  
 }  
  
 @Test  
 **public void** test4() {  
 GraphTask a = **new** GraphTask();  
 a.testRun(**"===Test 2000 vertices x 8 arc==="**, 2000, 8, 0, 1999);  
 System.***out***.println(**"\n"**);  
 *assertTrue* (**"There are no wrong answers"**, **true**);  
 }  
  
 @Test  
 **public void** test5() {  
 GraphTask a = **new** GraphTask();  
 a.testRun(**"===Test 2500 vertices x 10 arc==="**, 2500, 10, 0, 2499);  
 System.***out***.println(**"\n"**);  
 *assertTrue* (**"There are no wrong answers"**, **true**);  
 }  
}

Output:



<…>

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как стол

Автоматически созданное описание

<…>

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

References

1. “Algoritmid ja Andmestruktuurid Lectures” Jaanus Pöial

<https://enos.itcollege.ee/~japoia/algoritmid/graafid.html>

1. “Data Structures and Algorithms in Java. 4th Edition” M. Goodrich, R. Tamassia
2. “Dijkstra’s shortest path algorithm in Java – Tutorial”

https://vogella.com/tutorials/JavaAlgorithmsDijkstra/article.html

Appendix 1 – Program Code

**import** java.util.\*;  
**import** java.text.DateFormat;  
**import** java.text.ParseException;  
**import** java.text.SimpleDateFormat;  
  
  
*/\*\* Container class to different classes, that makes the whole  
 \* set of classes one class formally.  
 \*/***public class** GraphTask {  
  
 */\*\*  
 \* Main method.  
 \*/* **public static void** main(String[] args) {  
 GraphTask a = **new** GraphTask();  
 a.run();  
 }  
  
 */\*\*  
 \* Actual main method to run examples and everything.  
 \*/* **public void** run() {  
 List<Vertex> testAirports = **new** ArrayList<>();  
 testAirports.add(**new** Vertex(**"TLL"**, 15)); *//0* testAirports.add(**new** Vertex(**"HEL"**, 20)); *//1* testAirports.add(**new** Vertex(**"ARN"**, 30)); *//2* testAirports.add(**new** Vertex(**"LGW"**, 60)); *//3* testAirports.add(**new** Vertex(**"BCN"**, 45)); *//4* testAirports.add(**new** Vertex(**"FRA"**, 60)); *//5* testAirports.add(**new** Vertex(**"MAD"**, 45)); *//6* testAirports.add(**new** Vertex(**"BGY"**, 35)); *//7* testAirports.add(**new** Vertex(**"BVA"**, 30)); *//8* testAirports.add(**new** Vertex(**"ALC"**, 20)); *//9* List<Arc> testFlights = **new** ArrayList<>();  
 testFlights.add(**new** Arc(**"TLL-HEL"**, testAirports.get(0), testAirports.get(1), **"01.06.2021 06:00"**, 2, **"01.06.2021 06:35"**, 2));  
 testFlights.add(**new** Arc(**"TLL-HEL"**, testAirports.get(0), testAirports.get(1), **"01.06.2021 14:45"**, 2, **"01.06.2021 15:20"**, 2));  
 testFlights.add(**new** Arc(**"TLL-LGW"**, testAirports.get(0), testAirports.get(3), **"01.06.2021 15:55"**, 2, **"01.06.2021 17:00"**, 0));  
 testFlights.add(**new** Arc(**"TLL-ARN"**, testAirports.get(0), testAirports.get(2), **"01.06.2021 07:20"**, 2, **"01.06.2021 07:20"**, 1));  
 testFlights.add(**new** Arc(**"TLL-ARN"**, testAirports.get(0), testAirports.get(2), **"01.06.2021 16:30"**, 2, **"01.06.2021 16:30"**, 1));  
  
 testFlights.add(**new** Arc(**"HEL-ALC"**, testAirports.get(1), testAirports.get(9), **"02.06.2021 09:05"**, 2, **"02.06.2021 12:30"**, 1));  
 testFlights.add(**new** Arc(**"HEL-BCN"**, testAirports.get(1), testAirports.get(4), **"01.06.2021 20:40"**, 2, **"01.06.2021 23:30"**, 1));  
 testFlights.add(**new** Arc(**"HEL-FRA"**, testAirports.get(1), testAirports.get(5), **"01.06.2021 07:40"**, 2, **"01.06.2021 09:20"**, 1));  
  
 testFlights.add(**new** Arc(**"LGW-BCN"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 09:45"**, 0, **"01.06.2021 13:00"**, 1));  
 testFlights.add(**new** Arc(**"LGW-BCN"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 21:05"**, 0, **"02.06.2021 00:20"**, 1));  
 testFlights.add(**new** Arc(**"LGW-ALC"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 17:05"**, 0, **"01.06.2021 20:25"**, 1));  
 testFlights.add(**new** Arc(**"LGW-ALC"**, testAirports.get(3), testAirports.get(4), **"01.06.2021 20:40"**, 0, **"02.06.2021 00:15"**, 1));  
  
 testFlights.add(**new** Arc(**"ARN-FRA"**, testAirports.get(2), testAirports.get(5), **"01.06.2021 13:00"**, 1, **"01.06.2021 15:05"**, 1));  
 testFlights.add(**new** Arc(**"ARN-FRA"**, testAirports.get(2), testAirports.get(5), **"01.06.2021 16:35"**, 1, **"01.06.2021 18:40"**, 1));  
 testFlights.add(**new** Arc(**"ARN-ALC"**, testAirports.get(2), testAirports.get(9), **"01.06.2021 11:30"**, 1, **"01.06.2021 15:35"**, 1));  
  
 testFlights.add(**new** Arc(**"BCN-MAD"**, testAirports.get(4), testAirports.get(6), **"02.06.2021 11:50"**, 1, **"02.06.2021 13:25"**, 1));  
 testFlights.add(**new** Arc(**"BCN-MAD"**, testAirports.get(4), testAirports.get(6), **"02.06.2021 17:50"**, 1, **"02.06.2021 19:15"**, 1));  
 testFlights.add(**new** Arc(**"BCN-BGY"**, testAirports.get(4), testAirports.get(7), **"02.06.2021 06:20"**, 1, **"02.06.2021 08:00"**, 1));  
 testFlights.add(**new** Arc(**"BCN-BGY"**, testAirports.get(4), testAirports.get(7), **"02.06.2021 18:30"**, 1, **"02.06.2021 20:10"**, 1));  
 testFlights.add(**new** Arc(**"BCN-ALC"**, testAirports.get(4), testAirports.get(9), **"02.06.2021 08:55"**, 1, **"02.06.2021 10:05"**, 1));  
 testFlights.add(**new** Arc(**"BCN-ALC"**, testAirports.get(4), testAirports.get(9), **"02.06.2021 19:10"**, 1, **"02.06.2021 20:20"**, 1));  
  
 testFlights.add(**new** Arc(**"FRA-MAD"**, testAirports.get(5), testAirports.get(6), **"01.06.2021 13:10"**, 1, **"01.06.2021 16:00"**, 1));  
 testFlights.add(**new** Arc(**"FRA-MAD"**, testAirports.get(5), testAirports.get(6), **"01.06.2021 19:55"**, 1, **"01.06.2021 22:45"**, 1));  
  
 testFlights.add(**new** Arc(**"MAD-ALC"**, testAirports.get(6), testAirports.get(9), **"02.06.2021 07:40"**, 1, **"02.06.2021 08:50"**, 1));  
 testFlights.add(**new** Arc(**"MAD-ALC"**, testAirports.get(6), testAirports.get(9), **"02.06.2021 14:55"**, 1, **"02.06.2021 16:00"**, 1));  
  
 testFlights.add(**new** Arc(**"BGY-ALC"**, testAirports.get(7), testAirports.get(9), **"05.06.2021 11:20"**, 1, **"05.06.2021 13:30"**, 1));  
 testFlights.add(**new** Arc(**"BGY-BVA"**, testAirports.get(7), testAirports.get(8), **"03.06.2021 06:25"**, 1, **"03.06.2021 08:00"**, 1));  
  
 testFlights.add(**new** Arc(**"BVA-ALC"**, testAirports.get(8), testAirports.get(9), **"05.06.2021 22:10"**, 1, **"06.06.2021 00:20"**, 1));  
  
 Graph testGraph1 = **new** Graph(testAirports, testFlights);  
  
 Dijkstra dijkstra = **new** Dijkstra(testGraph1);  
 dijkstra.run(testGraph1.**airports**.get(3));  
  
 List<Arc> flights = dijkstra.getRoute(testGraph1.**airports**.get(0));  
  
 **try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
 } **catch** (NullPointerException e) {  
 System.***out***.println(**"No connection between "** + testGraph1.**airports**.get(3) + **" and "** + testGraph1.**airports**.get(0) + **"\n"**);  
 }  
  
  
  
 }  
  
 */\*\*  
 \* Method for automatic test generation and times calculation.  
 \* Prints out:  
 \* result of each run  
 \* times for graph generation and dijkstra work  
 \*  
 \** ***@param testName*** *- name of test: String  
 \** ***@param n*** *- amount of airports (vertices): int  
 \** ***@param m*** *- amount of flights (arcs): int  
 \** ***@param start*** *- airport of departure in list of vertices: int  
 \** ***@param finish*** *- airport of arrival in list of vertices: int  
 \*/* **public void** testRun(String testName, **int** n, **int** m, **int** start, **int** finish) {  
 List<Arc> flights;  
 **long** graphStart;  
 **long** graphEnd;  
 **long** graphSum = 0;  
 **long** dijkstraStart;  
 **long** dijkstraEnd;  
 **long** dijkstraSum = 0;  
 **int** run = 0;  
  
 System.***out***.println(testName);  
  
 **do** {  
 run += 1;  
 System.***out***.println(**"-> Run: "** + run);  
 graphStart = System.*currentTimeMillis*();  
 Graph testGraph2 = **new** Graph();  
 testGraph2.createRandomGraph(n, m);  
 graphEnd = System.*currentTimeMillis*() - graphStart;  
 graphSum += graphEnd;  
 *//System.out.println("Graph generated in:" + graphEnd + " ms");  
 //System.out.println("Dijkstra started\n");* dijkstraStart = System.*currentTimeMillis*();  
 Dijkstra dijkstra = **new** Dijkstra(testGraph2);  
 dijkstra.run(testGraph2.**airports**.get(start));  
 flights = dijkstra.getRoute(testGraph2.**airports**.get(finish));  
 dijkstraEnd = System.*currentTimeMillis*() - dijkstraStart;  
 dijkstraSum += dijkstraEnd;  
 **try** {  
 **for** (Arc flight : flights) {  
 System.***out***.println(flight);  
 }  
 System.***out***.println(dijkstra.totalTimeString() + **"\n"**);  
 } **catch** (NullPointerException e) {  
 *// System.out.println("No connection between " + testGraph2.airports.get(start) + " and " + testGraph2.airports.get(finish) + "\n");* }  
  
 *//System.out.println("Dijkstra algorithm finished in: " + dijkstraEnd + " ms\n");* } **while** (flights == **null**);  
  
 System.***out***.println(**"Total runs: "** + run);  
 System.***out***.println(**"Avg. Graph generation time: "** + graphSum / run + **" ms"**);  
 System.***out***.println(**"Avg. Dijkstra solution time: "** + dijkstraSum / run + **" ms"**);  
 System.***out***.println(**"Avg total: "** + (graphSum / run + dijkstraSum / run) + **" ms"**);  
 }  
  
 */\*\*  
 \* Suppose you are given a timetable, which consists of:  
 \* A set A of n airports, and for each airport a in A, a minimum connecting time c(a).  
 \* A set F of m flights, and the following, for each f in F:  
 \* Origin airport a1(f) in A  
 \* Destination airport a2 in A  
 \* Departure time t1(f)  
 \* Arrival time t2(f)  
 \*  
 \* Describe an efficient algorithm for the flight scheduling problem. In this problem,  
 \* we are given airports a and b, and a time t, and we wish to compute a sequence of flights  
 \* that allows one to arrive at the earliest possible time in b when departing from a at or after  
 \* time t. Minimum connecting times at intermediate airports should be observed.  
 \* What is the running time of your algorithm as a function n and m?  
 \*/  
  
  
 /\*\*  
 \* Class Vertex implements airport as Vertex of Graph.  
 \* Class has the following attributes:  
 \*  
 \* String id: IMO ID of Airport (3 characters)  
 \* Integer minConTime: minimum connecting time at airport - c(a)  
 \*/* **static class** Vertex {  
  
 **final private** String **id**;  
 **final private** Integer **minConTime**;  
  
 */\*\* Constructor \*/* Vertex(String id, Integer minConTime) {  
 **this**.**id** = id;  
 **this**.**minConTime** = minConTime;  
 }  
  
 */\*\* Method returns Vertex as String \*/* @Override  
 **public** String toString() { **return id**; }  
  
 */\*\* Method returns id of Vertex as String \*/* **public** String getId() { **return id**; }  
  
 */\*\* Method returns minimum connection time of Vertex as String \*/* **public** Integer getMinConTime() {  
 **return minConTime**;  
 }  
 }  
  
  
 */\*\*  
 \* Arc represents one arrow in the graph. Two-directional edges are  
 \* represented by two Arc objects (for both directions).  
 \*  
 \* In this solution class Arc represents flight between airports as edge in Graph.  
 \*  
 \* Class Arc has the following attributes:  
 \* String id : name of flight connection.  
 \* Vertex departure: Airport of departure, represented through class Vertex  
 \* Vertex arrival: Airport of arrival, represented through class Vertex  
 \* Date depTime: Time of departure in format dd.MM.yyyy hh:mm  
 \* Date arrTime: Time of arrival in format dd.MM.yyyy hh:mm  
 \* Integer flightTime: Flight time in minutes  
 \* Integer waitingTime: (default = 0) Waiting time at Airport of Departure in minutes  
 \*/* **static class** Arc {  
  
 **final private** String **id**;  
 **final private** Vertex **departure**;  
 **final private** Vertex **arrival**;  
 **final private** Date **depTime**;  
 **final private** Date **arrTime**;  
 **private final** Integer **flightTime**;  
 **private** Integer **waitingTime** = 0;  
  
 */\*\* Constructor \*/* Arc(String id, Vertex departure, Vertex arrival, String depTime, Integer depTimeZone, String arrTime, Integer arrTimeZone) {  
 DateFormat strToDate = **new** SimpleDateFormat(**"dd.MM.yyyy HH:mm"**);  
 **this**.**id** = id;  
 **this**.**departure** = departure;  
 **this**.**arrival** = arrival;  
 **try** {  
 **this**.**depTime** = strToDate.parse(depTime);  
 } **catch** (ParseException e) {  
 **throw new** RuntimeException(**"Wrong Date String Format, must be : dd.MM.yyyy hh:mm"** + depTime);  
 }  
 **try** {  
 **this**.**arrTime** = strToDate.parse(arrTime);  
 } **catch** (ParseException e) {  
 **throw new** RuntimeException(**"Wrong Date String Format, must be : dd.MM.yyyy hh:mm"** + arrTime);  
 }  
  
 **this**.**flightTime** = Math.*abs*((**int**) (**this**.**arrTime**.getTime() - **this**.**depTime**.getTime()) / 1000) / 60 - (arrTimeZone - depTimeZone) \* 60;  
 }  
  
 */\*\* Method returns flight connection as String.  
 \* If Arc Object has waitingTime > 0, method returns waiting time as separate line.  
 \*/* @Override  
 **public** String toString() {  
 DateFormat dateToStr = **new** SimpleDateFormat(**"dd.MM.yyyy HH:mm"**);  
 String string;  
 **if** (**this**.**waitingTime** != 0) {  
 string = **"Waiting Time at "** + **departure** + **": "** + **waitingTime** + **" min\n"** +  
 **id** + **": Flight Duration: "** + **flightTime** + **" min, Dep: "** + dateToStr.format(**depTime**) + **" Arr: "** + dateToStr.format(**arrTime**);  
 } **else** {  
 string = **id** + **": Flight Duration: "** + **flightTime** + **" min, Dep: "** + dateToStr.format(**depTime**) + **" Arr: "** + dateToStr.format(**arrTime**);  
 }  
 **return** string;  
 }  
  
 */\*\* Method returns airport of departure as Vertex \*/* **public** Vertex getDeparture() {  
 **return departure**;  
 }  
  
 */\*\* Method returns airport of arrival as Vertex \*/* **public** Vertex getArrival() {  
 **return arrival**;  
 }  
  
 */\*\* Method returns departure time as Date\*/* **public** Date getDepTime() {  
 **return depTime**;  
 }  
  
 */\*\* Method returns arrival time as Date\*/* **public** Date getArrTime() {  
 **return arrTime**;  
 }  
  
 */\*\* Method returns sum of waiting time and time in flight as Integer\*/* **public** Integer getFlightTime() {  
 **return flightTime** + **waitingTime**;  
 }  
  
 */\*\*  
 \* Method sets Waiting time  
 \** ***@param waitingTime*** *: Integer - waiting time at airport of departure  
 \*/* **public void** setWaitingTime(**int** waitingTime) {  
 **this**.**waitingTime** = waitingTime;  
 }  
 }  
  
 */\*\*  
 \* class Graph used for Graph generation from Vertexes (airports) and Arcs (flights)  
 \* class Graph has following attributes:  
 \* List<Vertex> airports - list of airports implemented as objects of class Vertex  
 \* List<Arc> flights - list of flight connections between airports implemented as objects of class Arc  
 \*/* **static class** Graph {  
  
 **private** List<Vertex> **airports**;  
 **private** List<Arc> **flights**;  
  
 */\*\* Constructor \*/* Graph(List<Vertex> airports, List<Arc> flights) {  
 **this**.**airports** = airports;  
 **this**.**flights** = flights;  
 }  
  
 */\*\* Alternative constructor \*/* Graph() {  
 **this**(**null**, **null**);  
 }  
  
 */\*\* Method returns list of airports in format of List of Vertexes \*/* **public** List<Vertex> getAirports() {  
 **return airports**;  
 }  
  
 */\*\* Method returns list of flights in format of List of Flights \*/* **public** List<Arc> getFlights() {  
 **return flights**;  
 }  
  
 */\*\* Method returns Graph as table of airports and departure flights of each airport. \*/* @Override  
 **public** String toString() {  
 String nl = System.*getProperty*(**"line.separator"**);  
 StringBuilder sb = **new** StringBuilder(nl);  
 **for** (Vertex airport : **airports**) {  
 sb.append(**"Airport: "**).append(airport.toString());  
 sb.append(**" -->"**);  
 sb.append(nl);  
 **for** (Arc flight : **flights**) {  
 **if** (airport == flight.**departure**) {  
 sb.append(**"Outgoing Flight: "**).append(flight);  
 sb.append(nl);  
 }  
 }  
 sb.append(nl);  
 }  
 **return** sb.toString();  
 }  
  
 */\*\* Method for creation of random Vertexes. Used for creation of random Graph \*/* **public** Vertex createRandomVertex() {  
 **int** leftLimit = 97;  
 **int** rightLimit = 122;  
 **int** idLength = 3;  
  
 String id = **new** Random().ints(leftLimit, rightLimit + 1).limit(idLength).collect(StringBuilder::**new**,  
 StringBuilder::appendCodePoint, StringBuilder::append).toString().toUpperCase();  
  
 Integer minConTime = **new** Random().nextInt(61);  
 **return new** Vertex(id, minConTime);  
 }  
  
 */\*\* Method for creation of random Arcs. Used for creation of random Graph \*/* **public** Arc createRandomArc(Vertex departure, Vertex arrival) {  
 String id = departure.getId() + **"-"** + arrival.getId();  
 **int** day = **new** Random().nextInt(10);  
 **int** depHour = **new** Random().nextInt(23);  
 **int** depMin = **new** Random().nextInt(60);  
 **int** arrHour;  
 **do** {  
 arrHour = **new** Random().nextInt(25);  
 } **while** (arrHour <= depHour);  
 **int** arrMin = **new** Random().nextInt(60);  
  
 **return new** Arc(id, departure, arrival, day + **".10.2021 "** + depHour + **":"** + depMin, 0, day + **".10.2021 "** + arrHour + **":"** + arrMin, 0);  
 }  
  
 */\*\* Method for creation of random Graphs. Used for creation of random Graph.  
 \* Due to implementation of filters must be rewritten \*/* **public void** createRandomGraph(**int** n, **int** m) {  
 **if** (n < 2) {  
 **throw** (**new** RuntimeException(**"The amount "** + n + **" of airports is not enough for flights generation. Rule: 1 < n < 2501"**));  
 } **else if** (n > 2500) {  
 **throw new** RuntimeException(**"The search of flight with amount "** + n + **" of airports will take much time. Rule: 1 < n < 2501"**);  
 }  
  
 **if** (m < 1) {  
 **throw** (**new** RuntimeException(**"The amount "** + m + **" of flights from each airport. Rule: 0 < m < 11"**));  
 } **else if** (m > 10) {  
 **throw new** RuntimeException(**"The search of flight with amount "** + m + **" of flights from each airport will take much time. Rule: 0 < m < 11"**);  
 }  
 List<Vertex> airports = **new** ArrayList<>();  
 List<Arc> flights = **new** ArrayList<>();  
 List<Arc> removeFlights = **new** ArrayList<>();  
 Vertex firstAirport = createRandomVertex();  
 airports.add(firstAirport);  
 **while** (airports.size() != n) {  
 **for** (Vertex airport : airports) {  
 Vertex nextAirport = createRandomVertex();  
 **if** (!airport.getId().equals(nextAirport.getId())) {  
 airports.add(nextAirport);  
 **break**;  
 }  
 }  
 }  
 **this**.**airports** = airports;  
  
 **while** (flights.size() != n \* m) {  
 **for** (**int** i = 0; i < airports.size() - 1; i++) {  
 **int** minArrInMins = Integer.***MAX\_VALUE***;  
 **for** (Arc flight : flights) {  
 **if** (flight.getArrival() == airports.get(i)) {  
 **if** ((**int**) flight.getArrTime().getTime() / 1000 / 60 < minArrInMins) {  
 minArrInMins = (**int**) flight.getArrTime().getTime() / 1000 / 60;  
 }  
 }  
 }  
 **for** (Arc flight : flights) {  
 **if** (flight.getDeparture() == airports.get(i)) {  
 **if** ((**int**) flight.getArrTime().getTime() / 1000 / 60 < minArrInMins + airports.get(i).getMinConTime()) {  
 removeFlights.add(flight);  
 }  
 }  
 flights.removeAll(removeFlights);  
 }  
 }  
 **for** (Vertex airport : airports) {  
 Vertex arrival;  
 **for** (**int** i = 0; i < m; i++) {  
 **do** {  
 arrival = airports.get(**new** Random().nextInt(airports.size()));  
 } **while** (airport.getId().equals(arrival.getId()));  
 flights.add(createRandomArc(airport, arrival));  
 }  
 }  
 }  
 **this**.**flights** = flights;  
 }  
 }  
  
 */\*\*  
 \* Class Dijkstra - class of Dijkstra algorithm used for search of fastest flight schedule.  
 \* Class Dijkstra has the following attributes:  
 \* List<Vertex> airports - list of airports from Class Graph object, implemented via Vertex class.  
 \* List<Arc> flights - list of flights from Class Graph object, implemented via Arc class.  
 \* Set<Vertex> settledAirports - set of airports, which shortest time already found.  
 \* Set<Vertex> unsettledAirports - set of airports, which shortest time is not found yet.  
 \* Map<Vertex, Vertex> previousAirports - map of indices, each vertex contains the index of previous vertex in a  
 \* path through the graph.  
 \* Map<Vertex, Integer> time - map of indices, each vertex contains the time used to reach this vertex.  
 \* Map<Vertex, Arc> usedFlights - map created for more convenient check of used flights in the path.  
 \* Vertex firstAirport - (default = null) first airport in path. The airport from which the journey starts.  
 \* Integer totalTime - default = 0) counter of Total Time used for journey.  
 \*/* **static class** Dijkstra {  
 **final private** List<Vertex> **airports**;  
 **private final** List<Arc> **flights**;  
  
 **private** Set<Vertex> **settledAirports**;  
 **private** Set<Vertex> **unsettledAirports**;  
  
 **private** Map<Vertex, Vertex> **previousAirports**;  
 **private** Map<Vertex, Integer> **time**;  
 **private** Map<Vertex, Arc> **usedFlights**;  
  
 **private** Vertex **firstAirport** = **null**;  
  
 **private** Integer **totalTime** = 0;  
  
 */\*\* Constructor  
 \*  
 \** ***@param graph*** *- object of Class Graph  
 \*/* Dijkstra(Graph graph) {  
 **this**.**airports** = **new** ArrayList<>(graph.getAirports());  
 **this**.**flights** = **new** ArrayList<>(graph.getFlights());  
 }  
  
 */\*\* Main method of Dijkstra Class. Used for execution of all needed calculations \*/* **public void** run(Vertex firstAirport) {  
 **this**.**firstAirport** = firstAirport;  
 **settledAirports** = **new** HashSet<>();  
 **unsettledAirports** = **new** HashSet<>();  
 **time** = **new** HashMap<>();  
 **usedFlights** = **new** HashMap<>();  
 **previousAirports** = **new** HashMap<>();  
 sortGraph(**airports**, **flights**);  
 **time**.put(firstAirport, 0); *// first airport added with 0 flight time* **unsettledAirports**.add(firstAirport);  
 **while** (**unsettledAirports**.size() > 0) {  
 Vertex airport = getMin(**unsettledAirports**);  
 **settledAirports**.add(airport);  
 **unsettledAirports**.remove(airport);  
 findMinimalTime(airport);  
 }  
 }  
  
 */\*\* Method sorts List of flights, received from graph object and remove flights with  
 \* impossible connection. In example, when departure time from airport is earlier than first possible arrival  
 \* time for this airport or time required for connection is less than minimum connection time of the  
 \* airport.  
 \** ***@param airports*** *- List of airports form Dijkstra class object.  
 \** ***@param flights*** *- List of flights form Dijkstra class object.  
 \*  
 \* returns noting, but reduce amount of flights in this.flights attribute of Dijkstra class object.  
 \*/* **private void** sortGraph(List<Vertex> airports, List<Arc> flights) {  
 List<Arc> removeFlights = **new** ArrayList<>();  
 **for** (**int** i = 0; i < airports.size() - 1; i++) {  
 **int** minArrInMins = Integer.***MAX\_VALUE***;  
 **for** (Arc flight : flights) {  
 **if** (flight.getArrival() == airports.get(i)) {  
 **if** ((**int**) flight.getArrTime().getTime() / 1000 / 60 < minArrInMins) {  
 minArrInMins = (**int**) flight.getArrTime().getTime() / 1000 / 60;  
 }  
 }  
 }  
 **for** (Arc flight : flights) {  
 **if** (flight.getDeparture() == airports.get(i)) {  
 **if** ((**int**) flight.getArrTime().getTime() / 1000 / 60 < minArrInMins + airports.get(i).getMinConTime()) {  
 removeFlights.add(flight);  
 }  
 }  
 }  
 }  
 **this**.**flights**.removeAll(removeFlights);  
 }  
  
 */\*\* Method finds minimal Time for airport and puts time and arrival Airport to Map time.  
 \* Also method updates Map usedFlights and previousAirports, and adds arrival airports to unsettled.  
 \** ***@param airport*** *- airport as Vertex class object  
 \*/* **private void** findMinimalTime(Vertex airport) {  
  
 List<Vertex> adjacentAirports = getNeighbors(airport);  
  
 **for** (Vertex arrAirport : adjacentAirports) {  
 **if** (getShortestTime(arrAirport) > getShortestTime(airport) + (getFlight(airport, arrAirport)).getFlightTime()) {  
 **time**.put(arrAirport, getShortestTime(airport) + (getFlight(airport, arrAirport)).getFlightTime());  
 **usedFlights**.put(arrAirport, getFlight(airport, arrAirport));  
 **previousAirports**.put(arrAirport, airport);  
 }  
 **unsettledAirports**.add(arrAirport);  
 }  
 }  
  
 */\*\* Method finds flight in list of flights of Dijkstra class object. In case if such flight not found,  
 \* what is impossible in proper code, the method trows Runtime Error  
 \*  
 \** ***@param airport*** *- departure airport in flight  
 \** ***@param arrAirport*** *- arrival airport in flight  
 \** ***@return*** *Arc  
 \*/* **private** Arc getFlight(Vertex airport, Vertex arrAirport) {  
 **for** (Arc flight : **flights**) {  
 **if** (flight.getDeparture().equals(airport) && flight.getArrival().equals(arrAirport)) {  
 **return** flight;  
 }  
 }  
 **throw new** RuntimeException(**"Some error in code."**);  
 }  
  
 */\*\* Method finds airport as Vertex class object with shortest time from Set of airports.  
 \*  
 \** ***@param airports*** *\* @ return Vertex  
 \*/* **private** Vertex getMin(Set<Vertex> airports) {  
 Vertex min = **null**;  
 **for** (Vertex airport : airports) {  
 **if** (min == **null**) {  
 min = airport;  
 } **else** {  
 **if** (getShortestTime(airport) < getShortestTime(min)) {  
 min = airport;  
 }  
 }  
 }  
 **return** min;  
 }  
  
 */\*\* Method checks if there are any departure flights from airport after arrival time + minimum connection time  
 \* and returns neighbors airports if connections with them is available.  
 \** ***@param airport*** *- current airport  
 \** ***@return*** *List*<*Vertex*> *- List of neighbor airports with feasible connection.  
 \*/* **private** List<Vertex> getNeighbors(Vertex airport) {  
 List<Vertex> neighbors = **new** ArrayList<>();  
  
 **int** minWaitingTime = 0;  
 **if** (**usedFlights**.get(airport) != **null**) {  
 minWaitingTime = (**int**) **usedFlights**.get(airport).getArrTime().getTime() / 1000 / 60 + airport.getMinConTime();  
 }  
  
 **for** (Arc flight : **flights**) {  
 **if** (flight.getDeparture().equals(airport) && !isSettled(flight.getArrival())) {  
 **if** (airport == **this**.**firstAirport**) {  
 neighbors.add(flight.getArrival());  
 } **else** {  
 **if** (minWaitingTime <= (**int**) flight.getDepTime().getTime() / 1000 / 60) {  
 flight.setWaitingTime((**int**) flight.getDepTime().getTime() / 1000 / 60 - (**int**) **usedFlights**.get(airport).getArrTime().getTime() / 1000 / 60);  
 neighbors.add(flight.getArrival());  
 }  
 }  
 }  
 }  
 **return** neighbors;  
 }  
  
 */\*\* Method checks if given airport is settled.  
 \*  
 \** ***@param airport*** *- airport for checking  
 \** ***@return*** *boolean - if settled = true, not settled = false  
 \*/* **private boolean** isSettled(Vertex airport) {  
 **return settledAirports**.contains(airport);  
 }  
  
 */\*\* Method returns time if calculated for arrAirport. If not calculated -  
 \* method returns max value for Integer (substitution for Infinity from mathematics Dijkstra algorithm)  
 \** ***@param arrAirport*** *- airport as Vertex class object  
 \** ***@return*** *int - time in minutes if known or MAX\_VALUE.  
 \*/* **private int** getShortestTime(Vertex arrAirport) {  
 **if** (**time**.get(arrAirport) == **null**) {  
 **return** Integer.***MAX\_VALUE***;  
 } **else** {  
 **return time**.get(arrAirport);  
 }  
 }  
  
 */\*\* Method connects path and returns list of flights required for this path.  
 \*  
 \** ***@param arrAirport*** *- airport as Vertex class object  
 \** ***@return*** *LinkedList*<*Arc*> *- list of flights required for this path as Arc class objects.  
 \*/* **public** LinkedList<Arc> getRoute(Vertex arrAirport) {  
 **this**.**totalTime** = 0;  
 LinkedList<Arc> flights = **new** LinkedList<>();  
 Vertex step = arrAirport;  
 **if** (**previousAirports**.get(step) == **null**) {  
 **return null**;  
 }  
 **while** (**previousAirports**.get(step) != **null**) {  
 flights.add(**usedFlights**.get(step));  
 **this**.**totalTime** += **usedFlights**.get(step).getFlightTime();  
 step = **previousAirports**.get(step);  
 }  
 Collections.*reverse*(flights);  
 **return** flights;  
 }  
  
 */\*\* Method converts Dijkstra class object totalTime, used for journey from first airport to last airport,  
 \* to Days, Hours, Minutes and returns it as String  
 \** ***@return*** *String - Total time used for journey as String.  
 \*/* **public** String totalTimeString() {  
 **return "Total time: "** + **this**.**totalTime** / 24 / 60 + **" days, "** + **this**.**totalTime** / 60 % 24 + **" hours and "** + **this**.**totalTime** % 60 + **" minutes."**;  
 }  
 }  
}