Assessed Individual Coursework 2 — Flight Itinerary

1 Overview

Your task is to use a graph to represent airline data, and to support searching. You should carefully test all of the code you write, generating new data files as necessary, and include the result of your tests in the report.
The coursework aims to reinforce your understanding of module material, specifically the following learning objectives:

- Gain an understanding of a range of graph classes and their use to represent realistic data.
- Gain further experience in object-oriented software engineering with a non-trivial class hierarchy: specifically selecting an appropriate class; reusing existing classes; extending existing classes.
- Using generic code: reusing generic graph classes, and parameterising a class with different types.
- You will also gain general software engineering experience, specifically downloading and using Open Source software, using a general method for a specific purpose, and issues with reusing existing code.
- Gain further experience with Java programming.

Information and guidelines

- This coursework should be done **individually**.
- You should tackle the coursework progressively each part in turn over the weeks to spread the load. See below for a suggested timing. We recommend that you get feedback your progress for each part during the lab sessions following the indicative timing.
- You should submit your report and all source code of your programs on Vision by Thursday 23rd of November, 2017. You will be asked to take part in peer-testing after submission.
- This coursework is worth 20% of this module (50% of the total coursework mark).

Indicative timing

Part A: Week 9

Familiarise with JGraphT.
Create graphs from vertices and edges.
Search for shortest paths.

Part B: Week 10

Add information to the graph's edges.
Other path searches.
Extend to provided routes data.

Part C: Week 11

Itinerary duration.
Implements methods for interface.

2 Coursework Parts

Preliminary Part: Installation and get familiar with JGraphT

You will need to download a personal copy of the Open Source JGraphT graph library. Having a single central copy of the library for all students on the course would save space, but it is a valuable experience for you to use an Open Source code repository.

• JGraphT package download

- Download the Open Source JGraphT graph library by following the instructions on: http://jgrapht.org/
  The following instructions are for `jgrapht-1.0.1` on a Unix machine using bash.
- Decompress and extract the tarball (this will create a 33M `jgrapht-1.0.1` directory), e.g.

```
$ tar zxvf jgrapht-1.0.1.tar.gz
```

- Delete the tarball to save 5.2M (6.9M for the zip file), e.g.

```
$ rm jgrapht-1.0.1.tar.gz
```

• Setup for command line compilation and execution

- Add JGraphT core JAR file to your class path by adding the following commands at the end of your `.profile` file in your home directory.
  In the following command you should replace `yourlogin` with *your login*, and `yourpath` with the *directory path* to where you have installed JGraphT. The first line adds JGraphT core JAR to your class path, the second adds JGraphT demo files your class path.

```
export CLASSPATH=/u1/cs2/yourlogin/yourpath/jgrapht-1.0.1/lib/↵
   jgrapht-core-1.0.1.jar:$CLASSPATH
export CLASSPATH=/u1/cs2/yourlogin/yourpath/jgrapht-1.0.1/↵
   source/jgrapht-demo/src/main/java/org/jgrapht/demo:↵
   $CLASSPATH
```

  For Jane Doe whose login is `jd42` and who is working on the assignment in the directory `F28DA/CW2`, the commands are:

```
export CLASSPATH=/u1/cs2/jd42/F28DA/CW2/jgrapht-1.0.1/lib/↵
   jgrapht-core-1.0.1.jar:$CLASSPATH
export CLASSPATH=/u1/cs2/jd42/F28DA/CW2/jgrapht-1.0.1/source/↵
   jgrapht-demo/src/main/java/org/jgrapht/demo:$CLASSPATH
```

  Note that when you copy-paste these commands, you will need to edit the text lines your obtain to remove spaces and some line breaks.
- Execute your new `.profile`, e.g.

```
% source ~/.profile
```

• Setup for Eclipse integration

To use and integrate JGraphT in Eclipse, you need to Configure the Java Build Path of your project. You will need to apply the following changes in Libraries:

- Add the external archive `jgrapht-core-1.0.1.jar`
- Once the archive is added, you can attach its sources by deploying its menu and edit Source attachment to point to the external directory location
  `/u1/cs2/yourlogin/yourpath/jgrapht-1.0.1/source/jgrapht-core/src`
  This will make the sources of JGraphT directly available within Eclipse for documentation and debugging purposes.
- Similarly, you can add the documentation by editing Javadoc location path to be `/u1/cs2/yourlogin/yourpath/javadoc/`.
  This will make the documentation of JGraphT directly available within Eclipse.

• **JGraphT demonstration programs**

Yo can copy the demonstration programs provided in the JGraphT source directory to your source directory. Go to the JGraphT source directory, e.g.

```
% cd ~/yourpath/jgrapht-1.0.1
% cd source/jgrapht-demo/src/main/java
% cp org/jgrapht/demo/HelloJGraphT.java ~/yoursrcpath
```

To compile, you will need to change the package declaration in the demonstration file to suit your setting. Compile and execute the 1st demo program (or equivalent under Eclipse):

```
% javac HelloJGraphT.java
% java HelloJGraphT
```

Execute other demo programs, e.g. `PerformanceDemo` - takes several minutes!

• **View Javadoc**

You will need to use the JGraphT Javadoc to locate appropriate classes and methods. Browse
http://jgrapht.org/javadoc/

• **Editing & compiling graph programs**

Rename and edit `HelloJGraphT.java` (`org/jgrapht/demo/HelloJGraphT.java`), by adding a new edge to one of the graphs. Compile and run your revised program Congratulations, you are ready to start writing graph programs.

• **Viewing example graph programs**

Often the easiest way to write a program is to reengineer, i.e. copy and modify, a similar program. More example programs are available in the test directory

```
% cd ~/yourpath/jgrapht-1.0.1
% cd source/jgrapht-core/src/test/java
```

$\Longrightarrow$ **Start of week 9** You should complete this set up at the start of week 9.

Part A: Representing direct flights and least cost connections **Week 9**

Write a program to represent the following direct flights with associated costs as a graph. For the purpose of this exercise assume that flights operate in both directions with the same cost, e.g. Edinburgh ↔ Heathrow denotes a pair of flights, one from Edinburgh to Heathrow, and another from Heathrow to Edinburgh.
Hint: Flights are directed, i.e. from one airport to another, and weighted by the ticket cost, hence use the JGraphT `SimpleDirectedWeightedGraph` class. You should display the contents of the graph (and may omit the weights).

Flight	Cost
Edinburgh ↔ Heathrow	£80
Heathrow ↔ Dubai	£130
Heathrow ↔ Sydney	£570
Dubai ↔ Kuala Lumpur	£170
Dubai ↔ Edinburgh	£190
Kuala Lumpur ↔ Sydney	£150

Extend your program to search the flights graph to find the least cost route between two cities consisting of one or more direct flights.
Hint: use methods from the `DijkstraShortestPath` class to find the route. A possible interface for your program might be one where you suggest a start and an end city and the cost of the entire route is added up and printed.

```
The following airports are used:
       Edinburgh
       Heathrow
       ...
 Please enter the start airport
    Edinburgh
 Please enter the destination airport
    Kuala Lumpur
 Shortest (i.e. cheapest) path:
 1. Edinburgh -> Dubai
 2. Dubai -> Kuala Lumpur
 Cost of shortest (i.e. cheapest) path = £360
```

Java hint: You can redefine the `.toString{}` method in your classes to customise printing of information.

⟹ **Week 9** Name the main class of your program `FlightItinerary`. In this class, create a static method `partA()` which contains your program for this part. The `main` method of your program should call `partA()`. You should aim to complete this part by the end of week 9.

Part B: Additional flight information, use provided flight route dataset **Week 10**

Start a static method `partB` in your main class `FlightItinerary` operating on a flight graph that will now include the following information about each flight. The flight number, e.g. `BA345`; the departure time; the arrival time; the flight duration; and the ticket price, e.g. `100`. All times should be recorded in 24 hour `hhmm` format, e.g. `1830`. Individual flight durations are under 24h.
At this stage you could decide to first extend your graph from Part A or to start building your graph from the provided flight routes dataset
Use the additional flight information to print itineraries for least cost journeys in a format similar to the following example. The key aspects are:

1. A sequence of connecting flights (with least cost)
2. A total cost for the route

An example itinerary for Part B (and Part C) might resemble:

```
Itinerary for Edinburgh to Toronto
Leg Leave         At   On    Arrive        At
1   Edinburgh     0530 FZ345 Dubai         1100
2   Dubai         1230 EK657 Kuala Lumpur  1730
3   Kuala Lumpur  1800 QF652 Sydney        2130
Total Journey Cost    = £510
Total Time in the Air = 840
```

Java hint: You should use `String.format` to align the information you are printing.
For the purpose of printing such itinerary, you should implement and use some of the methods of the `IFlightItinerary` and `IItinerary` interfaces (see below).
Build your graph of flights using the provided flight route dataset and its reader (`FlightsReader`). The dataset is composed of a list of airlines (indexed by a two character character code), a list of airports (indexed by a two or three character code), and a list of flight routes (indexed by a flight code). The initial list of airlines, airports and routes is from the Open Flights https://openflights.org/ open source project. In addition to these initial lists the following information were automatically and randomly generated: the flight numbers, departure and arrival times, cost.

$\Longrightarrow$ **Week 10** In the main class (`FlightItinerary`) of your program, create a static method `partB()` which contains your program for this part. The `main` method of your program should call `partB()`. Prepare JUnit test cases of your implementation (based on the `IFlightItinerary` and `IItinerary` interfaces). You should aim to complete this part by the end of week 10.

Part C: Itinerary durations, least stopover, and meet ups — Week 11

Extend your program to calculate the total time in the air, i.e. the sum of the durations of all flights in the itinerary and the total trip time.
Hint: you will need to write functions to perform arithmetic on 24 hour clock times.
Attempt the following extensions to your program.

1. Extend your program to locate routes with the fewest number of changeovers.

   Hint: use a standard graph traversal algorithm, available in JGraphT.
2. Extend your program to offer the possibility to exclude one or more airports from the itinerary search.
3. Extend your program to offer the possibility to search for meet-up place for two people located at two different airports.

Extra, attempt the following extension to program.

1. Extend your program to offer the possibility to search for least time meet-up place for two people located at two different airports (considering a given starting time).

$\Longrightarrow$ **Week 11** In the main class (`FlightItinerary`) of your program, create a static method `partC()` which contains your program for this part. The `main` method of your program should call `partC()`. Prepare additional JUnit test cases of your implementation (based on the `IFlightItinerary` and `IItinerary` interfaces). You should aim to complete this part by the deadline of the coursework on week 11.

3 Interfaces

IFlightItinerary

`boolean populate(HashSet<String[]> airlines, HashSet<String[]> airports, HashSet<String[]> routes)`

Populates the graph with the airlines, airports and routes information.

`IItinerary leastCost(String to, String from)`

Returns the cheapest flight itinerary from one airport (airport code) to another

`IItinerary leastHop(String to, String from)`

Returns least connections flight itinerary from one airport (airport code) to another

`IItinerary leastCost(String to, String from, List<String> excluding)` Returns the cheapest flight itinerary from one airport (airport code) to another, excluding a list of airport (airport codes)

`IItinerary leastHop(String to, String from, List<String> excluding)`

Returns least connections flight itinerary from one airport (airport code) to another, excluding a list of airport (airport codes)

`String leastCostMeetUp(String at1, String at2)`

Returns the airport code of the best airport for the meet up of two people located in two different airports (airport codes) accordingly to the itineraries costs

`String leastHopMeetUp(String at1, String at2)`

Returns the airport code of the best airport for the meet up of two people located in two different airports (airport codes) accordingly to the number of connections

`String leastTimeMeetUp(String at1, String at2, String startTime)`

Returns the airport code of the best airport for the earliest meet up of two people located in two different airports (airport codes) when departing at a given time

IItinerary

`List<String> getStops()`

Returns the list of airports codes composing the itinerary

`List<String> getFlights()`

Returns the list of flight codes composing the itinerary

`int totalHop()`

Returns the number of connections of the itinerary

`int totalCost()`

Returns the total cost of the itinerary

`int airTime()`

Returns the total time in flight of the itinerary

`int connectingTime()`

Returns the total time in connection of the itinerary

`int totalTime()`

Returns the total travel time of the itinerary

4 Coding Style

Your mark will be based partly on your coding style. Here are some recommendations:

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and whitespaces should be used to improve readability.
- No variable declarations should appear outside methods ("instance variables") unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose value does not have to be remembered until the next method call, should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared private (not protected) to maximize information hiding. Any access to the variables should be done with accessor methods.

5 Submission

Submit on Vision a `.zip` or `.tar.gz` archive containing a brief report, and your Java source code.

Report The report in `.pdf`, `.rtf`, `.odt`, `.doc` or `.docx` format should be brief (max. 5 pages), prepared using the text processor of your choice, and should indicate the implementation and representation choices you made for part B and C, the known limitations of your implementation of each part, a description of test data and testing outcomes, and includes reasons why test data was chosen.

Source code Your code should indicate in comments when necessary the parts A, B and C. Place in a sub directory of the archive the `.java` source files of your program (do **not** include the compiled `.class` files). This includes unit test cases `*Test.java` testing your implementation of the `IFlightItinerary` methods. Do not include the dataset.

Peer-testing You will be required to take part in peer-testing after submission. You will be using your classes and the unit test cases you have prepared. At the end of the peer-testing period you will be asked to submit a short reflective summary on Vision. In some circumstances, a demonstration of your program could be organised, this needs to be approved by the lecturer of the course.

6 Marking Scheme

Your **overall mark** will be computed as follows.

Part A	10 marks
Part B	25 marks
Part C	25 marks
Coding style	10 marks
Report with clear structure, content and appropriate length.	10 marks
Test cases submitted, peer-testing involvement and reflective summary (or demonstration of your program if circumstances require)	20 marks

Your coursework is due to be submitted by Thursday 23rd of November, 2017. If you hand in work late, without extenuating circumstances, 10% of the maximum available mark will be deducted from the mark awarded for each day late.