Assessed Individual Coursework 2 — Flying Planner

1 Overview

Your task is implement a Flying Planner which uses a graph library to represent airline data, and which supports searching. You should carefully test all of the code you write, generating new test files as necessary, and include illustrations of your Flying Planner user interface in your report.
The coursework aims to reinforce your understanding of course 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 timing guidelines

- This coursework should be done **individually**. See Section 5 for details.
- This coursework is worth 20% of the course (50% of the total coursework mark).
- You should submit your report and all source code of your programs on Vision by 3:30pm on Thursday November 28th, 2019. You will be asked to take part in peer-testing after submission. See Section 6 for details.
- 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 on your progress for each part during the lab sessions following the indicative timing.

Part A: Week 8-9

Familiarise yourself with JGraphT (see Section 2)
Create graphs by hand from vertices and edges
Search for shortest paths

Part B: Week 9-10

Build graph from provided flights data
Add information to the graph's edges
Other path searches

Part C: Week 10-11

Implement all extensions:
Journey duration
Least hops
Directly connected order
Meet-Up search

2 JGraphT

JGraphT is a Java library of graph theory data structures and algorithms

Note that we will be using JGraphT version **1.3.0** which is the latest released version at the time of writing this coursework specifications.

2.1 Preliminary Part: Installation and get familiar with JGraphT

To use the library, you need to have a personal copy of the Open Source JGraphT graph library in your working environment. As we explain below, this could be automatically done with Maven. Find also in this section instructions to manually install in Eclipse, and under Linux.
You can get more information about JGraphT on its public website, and more information about the classes the library provides in its Javadoc documentation available online:

https://jgrapht.org/
https://jgrapht.org/javadoc/
https://jgrapht.org/javadoc-1.3.0/

2.2 GitLab-Student and Maven

A coursework starting point project is available as a repository on GitLab-Student:

https://gitlab-student.macs.hw.ac.uk/f28da_19-20/f28da_19-20_cw2

As JGraphT releases are published to the Maven Central Repository, this coursework project includes the necessary dependency in the Maven setting (`pom.xml` file).

```
<groupId>org.jgrapht</groupId>
<artifactId>jgrapht-core</artifactId>
<version>1.3.0</version>
```

On GitLab-Student, or on an IDE supporting Maven, you should not need to manually install JGraphT as it will be installed by Maven. The following subsections give you however instructions on how to manually install JGraphT. These instructions could be informative to read and help you understand better how such Java library works.

2.3 (Optional) Instructions to manually install JGraphT

JGraphT package download

- Download the Open Source JGraphT graph library by following the instructions on:
  http://jgrapht.org/
  The following instructions are for `jgrapht-1.3.0` on a Unix machine using bash.
- Decompress and extract the tarball (this will create a 88M `jgrapht-1.3.0` directory), e.g.

  ```
  $ tar zxvf jgrapht-1.3.0.tar.gz
  ```

- Delete the tarball to save 45M (47M for the zip file), e.g.

  ```
  $ rm jgrapht-1.3.0.tar.gz
  ```

In the following, we assume that the extracted `jgrapht-1.3.0` directory is located at `/path/to/your/login/path/to/`

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-1.3.0.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 `/path/to/your/login/path/to/jgrapht-1.3.0/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 `/path/to/your/login/path/to/jgrapht-1.3.0/javadoc/`.

  This will make the documentation of JGraphT directly available within Eclipse.

Setup for command line compilation and execution

- Add JGraphT's core JAR file `jgrapht-core-1.3.0.jar` to your class path by adding the following commands at the end of your `.profile` file in your home directory. Alternatively, you could pass the additional class path information to `javac` and `java` with the `-cp` command line argument. The following line adds JGraphT's core JAR to your class path (this could be repeated for other JGraphT JAR, to run JGraphT's `HelloJGraphT` demo you will need to also include `jgrapht-io-1.3.0.jar`).

  ```
  export CLASSPATH=/path/to/your/login/path/to/jgrapht-1.3.0/lib/jgrapht-core↵
      -1.3.0.jar:$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` for the setting to be taken into account, e.g.

  ```
  $ source ~/.profile
  ```

JGraphT demonstration programs

Yo can compile and run the demonstration programs provided in the JGraphT source directory. To do so, go to the JGraphT source directory, compile, and execute the `HelloJGraphT` demo program (the compiled classes will be put in a `bin` directory):

```
$ cd /path/to/your/login/path/to/jgrapht-1.3.0
$ cd source/jgrapht-demo/src/main/java
$ mkdir -p bin
$ javac -d bin org/jgrapht/demo/HelloJGraphT.java
$ java -cp ./bin:$CLASSPATH org.jgrapht.demo.HelloJGraphT
```

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

3 Coursework Parts

Part A: Representing direct flights and least cost connections **Week 8–9**

Write a program `FlyingPlannerMainPartA` (containing a single main method) 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 journey between two cities consisting of one or more direct flights.
Hint: use methods from the `DijkstraShortestPath` class to find the journey. A possible interface for your program might be one where you suggest a start and an end city and the cost of the entire journey 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.

⟹ **by mid Week 9** Implement the main method your `FlyingPlannerMainPartA` program. This `FlyingPlannerMainPartA` do not need to use or implement the provided interfaces. No test is provided nor necessary for his part. You should aim to complete this part by mid Week 9.

Part B: Use provided flights dataset, add flight information — Week 9–10

You should now write a program `FlyingPlannerPartBC` (containing a single main method) which will make use of your class `FlyingPlanner` (this is the central class of your program although it does not have to have a main method).

Add flight information

Your program should be 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.
Use the additional flight information to print the least cost journey 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 journey.

An example journey for Part B (and Part C) might resemble the following when the departure city is Edinburgh and the destination Sydney:

```
Journey for Newcastle (NCL) to Newcastle (NTL)
Leg Leave            At   On     Arrive           At
1   Newcastle (NCL) 1918 KL7893 Amsterdam (AMS) 2004
2   Amsterdam (AMS) 0747 CX0831 Hong Kong (HKG) 1702
3   Hong Kong (HKG) 0748 CX7100 Brisbane (BNE)  1427
4   Brisbane (BNE)  1628 QF0640 Newcastle (NTL) 1729
Total Journey Cost    = £1035
Total Time in the Air = 1061
```

Java hint: You should use `String.format` to align the information you are printing.

Use provided flights dataset

Build your graph of flights using the provided flights dataset and its reader (`FlightsReader`). The dataset is composed of a list of airports (indexed by a three character code), and a list of flights (indexed by a flight code). The list of airports and flights originated 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.

Interfaces to implement

For the purpose of printing such journey, and to complete this part,

- your `FlyingPlanner` class should implement the `IFlyingPlannerPartB<Airport,Flight>` interface;
- your `Journey` class should implement the `IJourneyPartB<Airport,Flight>` interface;
- your `Airport` class should implement the `IAirportPartB` interface,
- your `Flight` class should implement the `IFlight` interface,

⟹ **by mid Week 10** Implement the main method of your `FlyingPlannerMainPartBC` program and the methods of `FlyingPlanner`, `Journey`, `Airport`, `Flight` according to the provided interfaces. Your `FlyingPlanner` class should past the Part B test cases of the provided `FlyingPlannerProvidedTest` JUnit test class. Implement additional test cases in `FlyingPlannerTest`. You should aim to complete this part by mid Week 10.

Part C: Advanced features — Week 10–11

Extend you `FlightPlanner` class with the following extensions.

Journey duration

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

Least hops

Extend your program to locate journeys with the fewest number of changeovers. Extend your program to offer the possibility to exclude one or more airports from the journey search.

Directly connected order

Extend your program to calculate for an airport the number of directly connected airports. Two airports are directly connected if there exist two flights connecting them in a single hop in both direction.
Extend your program to calculate the set of airports reachable from a given airport that have strictly more direct connections.
Hint: use a directed acyclic graph, available in JGraphT.

Meet-Up search

Extend your program to offer the possibility to search for a least-hop/least-price meet-up place for two people located at two different airports. The meet-up should be different than the two stating airports.
Extend your program to offer the possibility to search for a least time meet-up place for two people located at two different airports (considering a given starting time).

Interfaces to implement

To complete this part,

- your `FlyingPlanner` class should implement the `IFlyingPlannerPartC<Airport,Flight>` interface;
- your `Journey` class should implement the `IJourneyPartC<Airport,Flight>` interface;
- your `Airport` class should implement the `IAirportPartC` interface,
- your `Flight` class should implement the `IFlight` interface,

$\Longrightarrow$ **by mid Week 11** Implement the methods of `FlyingPlanner`, `Journey`, `Airport`, `Flight` according to the provided interfaces. Your `FlyingPlanner` class should past the Part C test cases of the provided `FlyingPlannerProvidedTest` JUnit test class. Implement additional test cases in `FlyingPlannerTest`. You should aim to complete this part by the deadline of the coursework on week 11.

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 Note on plagiarism and collusion

- The coursework is an **individual** coursework.
- You are permitted to **discuss** the coursework with your with your classmates. You can **get help** from lecturer and lab helpers in lab sessions. You can get help and **ask questions** to lecturer, via GitLab-Student or by email or at the beginning or end of lecture sessions, or during the office hour of the lecturer.
- Coursework reports must be written in your **own words** and any code in their coursework must be your **own code**. If some text or code in the coursework has been taken from other sources, these sources must be **properly referenced**. Failure to reference work that has been obtained from other sources or to copy the words and/or code of another student is **plagiarism** and if detected, this will be reported to the School's Discipline Committee. If a student is found guilty of plagiarism, the penalty could involve voiding the course.
- Students must **never** give hard or soft copies of their coursework reports or code to another student. Students must always **refuse** any request from another student for a copy of their report and/or code.
- Sharing a coursework report and/or code with another student is **collusion**, and if detected, this will be reported to the School's Discipline Committee. If found guilty of collusion, the penalty could involve voiding the course.
- **Special note for re-using available code**: If you are re-using code that you have not yourself written, then this must clearly be indicated. At all time, you have to make clear what part is not yours, and what in fact is your contribution. Re-using existing code and amending it is perfectly fine, as long as you are not trying to pass it on as your own work, so you must clearly state where it is taken from. A brief additional explanation of why you chose this code would be an added benefit and even adds value to your work. If your code is found elsewhere by the person marking your work, and you have not mentioned this, you may find yourself having to go before a disciplinary committee and face grave consequences.

6 Submission

Submit a `.zip` or `.tar.gz` archive file electronically on **Vision**.

- Follow the sub directory of the template files provided on GitLab-Student. This should include all [1] the `.java` source files of your program as well as your jUnit 4 test cases in `FlyingPlannerTest.java`. Do **not** include the compiled `.class` files.

  You should fork the `F28DA_19-20_CW2` on GitLab-Student and regularly commit your work. You should **not** invite any other students to your project nor share your code with other students. Remember that the final submission is through **Vision**.

- A short report in `.pdf`, `.rtf`, `.odt`, `.doc` or `.docx` format should be brief (max. 5 pages) and should indicate the implementation and representation choices you made for Part B and C, show screenshots of running Part B, known limitations of your implementation of each part, a description of test data and testing outcomes, and includes reasons why test data was chosen.

Your coursework is due to be submitted by 3:30pm on Thursday November 28th, 2019.
The course applies the University's coursework policy.

- No individual extension for coursework submissions.
- Deduction of 30% from the mark awarded for up to 5 working days late submission.
- Submission more than 5 working days late will not get a mark.
- If you have mitigating circumstances for an extension, talk to your Personal Tutor and submit a Mitigating Circumstances (MC) form with supporting documentation to the School Office.

You will be required to take part in **peer-testing** after submission. You will be using your implementation and test cases. At the end of the peer-testing period you will be asked to submit a short **reflective summary** on Vision[2]

7 Marking Scheme

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

Coding style, program compiles, program runs and produced meaningful output (Part A/Part B)	25 marks
Report with clear structure, content and appropriate length, screenshots of Part B showing normal run and erroneous use with incorrect user input.	25 marks
Implementation the methods for Part B and Part C in `FlyingPlanner`, `Airport`, `Flight`, `Journey`	25 marks
Test cases submitted, peer-testing involvement and reflective summary (or demonstration of your program if circumstances require)	25 marks

Your coursework is due to be submitted by 3:30pm on Thursday November 28th, 2019. The course applies the new submission of coursework policy.

- No individual extension for coursework submissions.
- Deduction of 30% from the mark awarded for up to 5 working days late submission.
- Submission more than 5 working days late will not get a mark.
- If you have mitigating circumstances for an extension, talk to your Personal Tutor and submit an MC form with supporting documentation to the School's Office

[1] Submit also the Java files provided (whether you have altered them or not). You should not modify the interfaces provided. If you consider you need to add or modify some method signatures in the interfaces, please speak first to the lecturer of the course.

[2] In some circumstances, a demonstration could be organised, this needs to be approved by the lecturer of the course.