F28DA Coursework 2

By: Simonas Banys
Date: 27/11/2017

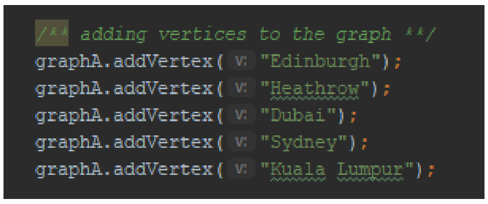
Lecturer: Dr. Manuel Maarek

Introduction:

The problem space for the project was given to implement a Flight Itinerary for the user, using a Simple Directed Weighted Graph. In part A the graph was created by manual input from the programmer for Vertices using city names and two cities for edges thus providing only a small graph with few vertices to traverse. Part B used automated graph population reading from provided dataset from file, thus creating a much larger graph which can be expanded at any time by creating more entries in the data file. The whole program implemented IFlightItinerary and IItinerary interfaces for calculating time spent in the air, time spent for connections/layovers and total time spent during the whole Itinerary.

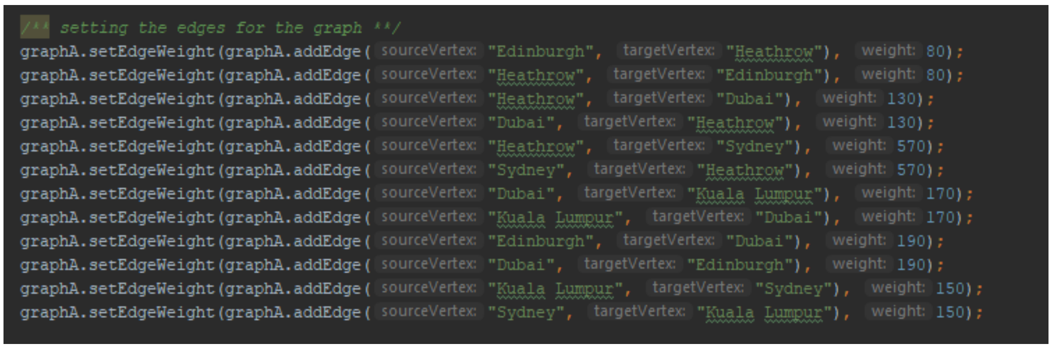
Part A

In part A the programmer created a graph and populated it with given list of vertices manually.



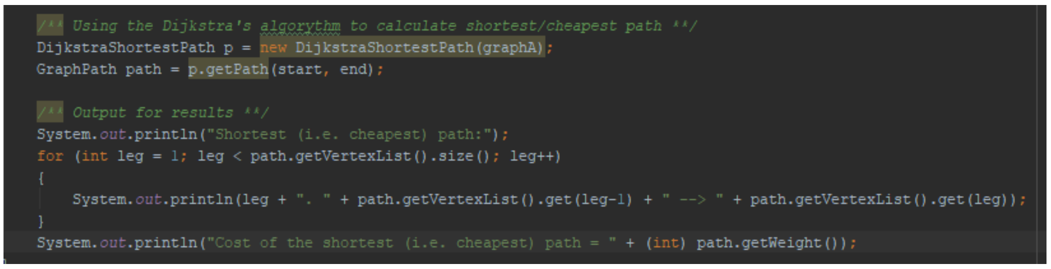
```
/** adding vertices to the graph **/
graphA.addVertex( v: "Edinburgh");
graphA.addVertex( v: "Heathrow");
graphA.addVertex( v: "Dubai");
graphA.addVertex( v: "Sydney");
graphA.addVertex( v: "Kuala Lumpur");
```

The vertices then received edges to connect them and weights for the edges to be used in shortest/cheapest path calculations.



```
/** setting the edges for the graph **/
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Edinburgh",  targetVertex: "Heathrow"),  weight: 80);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Heathrow",  targetVertex: "Edinburgh"),  weight: 80);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Heathrow",  targetVertex: "Dubai"),  weight: 130);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Dubai",  targetVertex: "Heathrow"),  weight: 130);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Heathrow",  targetVertex: "Sydney"),  weight: 570);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Sydney",  targetVertex: "Heathrow"),  weight: 570);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Dubai",  targetVertex: "Kuala Lumpur"),  weight: 170);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Kuala Lumpur",  targetVertex: "Dubai"),  weight: 170);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Edinburgh",  targetVertex: "Dubai"),  weight: 190);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Dubai",  targetVertex: "Edinburgh"),  weight: 190);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Kuala Lumpur",  targetVertex: "Sydney"),  weight: 150);
graphA.setEdgeWeight(graphA.addEdge( sourceVertex: "Sydney",  targetVertex: "Kuala Lumpur"),  weight: 150);
```

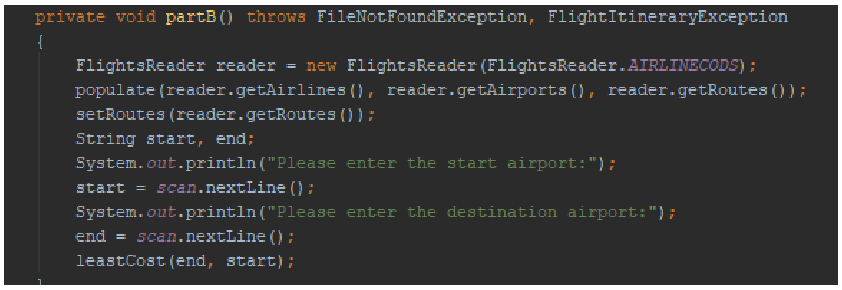
It is a very basic implementation of the Flight map graph. The graph uses Dijkstra's Shortest Path algorithm to calculate it.



```
/** Using the Dijkstra's algorythm to calculate shortest/cheapest path **/
DijkstraShortestPath p = new DijkstraShortestPath(graphA);
GraphPath path = p.getPath(start, end);

/** Output for results **/
System.out.println("Shortest (i.e. cheapest) path:");
for (int leg = 1; leg < path.getVertexList().size(); leg++)
{
    System.out.println(leg + ". " + path.getVertexList().get(leg-1) + " --> " + path.getVertexList().get(leg));
}
System.out.println("Cost of the shortest (i.e. cheapest) path = " + (int) path.getWeight());
```

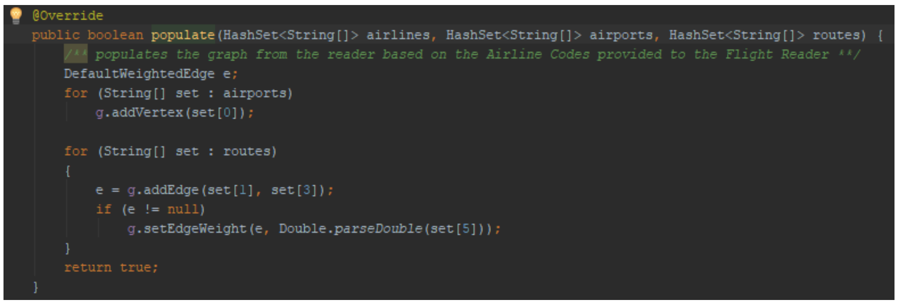
Part B

In part B the program is automated to read the flights data from a file.



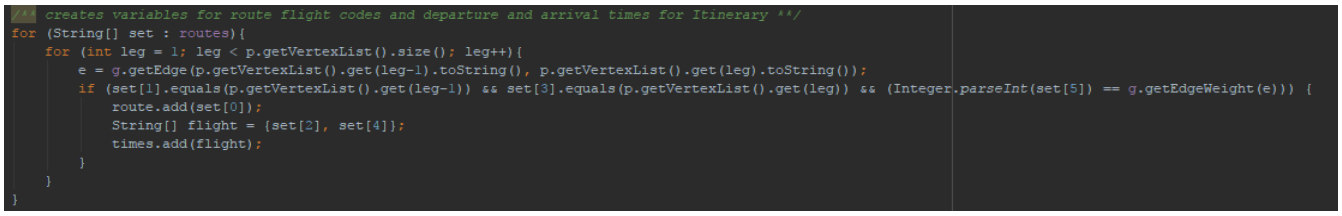
```
private void partB() throws FileNotFoundException, FlightItineraryException
{
    FlightsReader reader = new FlightsReader(FlightsReader.AIRLINECODS);
    populate(reader.getAirlines(), reader.getAirports(), reader.getRoutes());
    setRoutes(reader.getRoutes());
    String start, end;
    System.out.println("Please enter the start airport:");
    start = scan.nextLine();
    System.out.println("Please enter the destination airport:");
    end = scan.nextLine();
    leastCost(end, start);
```

It uses a separate class called FlightsReader to read the data and then store it.



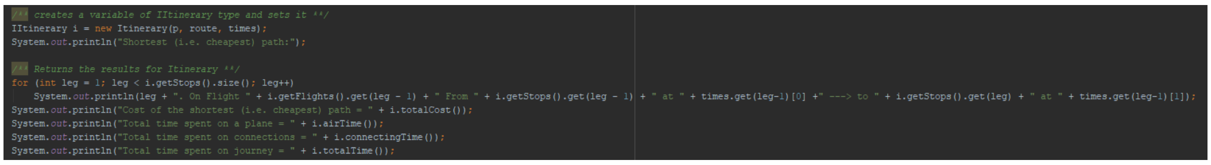
```
@Override
public boolean populate(HashSet<String[]> airlines, HashSet<String[]> airports, HashSet<String[]> routes) {
    /** populates the graph from the reader based on the Airline Codes provided to the Flight Reader **/
    DefaultWeightedEdge e;
    for (String[] set : airports)
        g.addVertex(set[0]);

    for (String[] set : routes)
    {
        e = g.addEdge(set[1], set[3]);
        if (e != null)
            g.setEdgeWeight(e, Double.parseDouble(set[5]));
    }
    return true;
}
```

Then calls to another method to populate the graph. First it adds the airports as vertices, then adds edges between them, as the graph does not allow multiple edges in the same direction between vertices a check is required before setting their weight if an edge already exits.
Once population is done the program sets a hash set of routes as a global variable for the class to be used throughout all the methods in the class. Program receives the input from user and calculates the least cost for the travel based on origin and destination.
Firs the program uses the Dijkstra's algorithm to calculate the shortest path. After it finds the departure and arrival times for the itinerary and puts them in an array list to be used for calculating time.



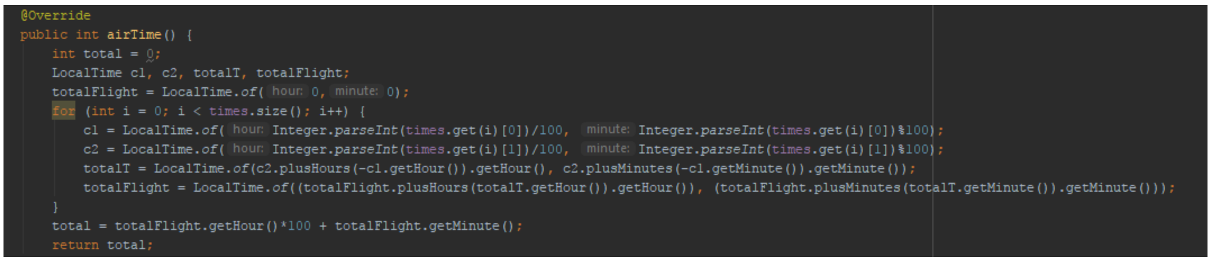
```
/** creates variables for route flight codes and departure and arrival times for Itinerary **/
for (String[] set : routes){
    for (int leg = 1; leg < p.getVertexList().size(); leg++){
        e = g.getEdge(p.getVertexList().get(leg-1).toString(), p.getVertexList().get(leg).toString());
        if (set[1].equals(p.getVertexList().get(leg-1)) && set[3].equals(p.getVertexList().get(leg)) && (Integer.parseInt(set[5]) == g.getEdgeWeight(e))) {
            route.add(set[0]);
            String[] flight = {set[2], set[4]};
            times.add(flight);
        }
    }
}
```

Once that is completed the program creates a IItinerary type variable and proceeds with calculations.



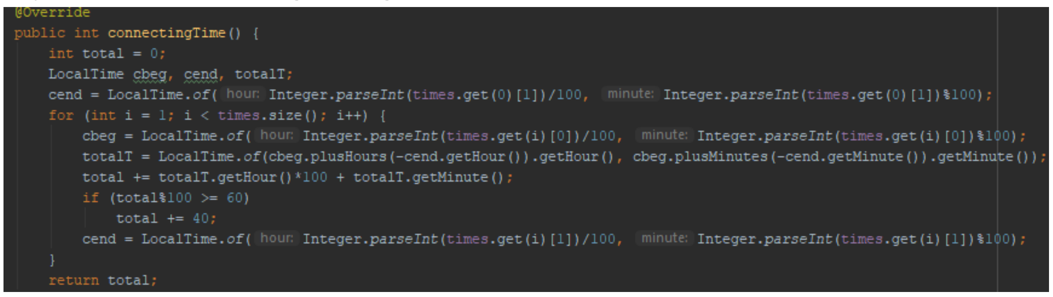
```
/** creates a variable of IItinerary type and sets it **/
IItinerary i = new Itinerary(p, route, times);
System.out.println("Shortest (i.e. cheapest) path:");

/** Returns the results for Itinerary **/
for (int leg = 1; leg < i.getStops().size(); leg++)
    System.out.println(leg + ". On Flight " + i.getFlights().get(leg - 1) + " From " + i.getStops().get(leg - 1) + " at " + times.get(leg-1)[0] +" ---> to " + i.getStops().get(leg) + " at " + times.get(leg-1)[1]);
System.out.println("Cost of the shortest (i.e. cheapest) path = " + i.totalCost());
System.out.println("Total time spent on a plane = " + i.airTime());
System.out.println("Total time spent on connections = " + i.connectingTime());
System.out.println("Total time spent on journey = " + i.totalTime());
```

The air time is calculating by taking the already received data about departure and arrival times for flights for the itinerary and adding separate flight times together:



```
@Override
public int airTime() {
    int total = 0;
    LocalTime c1, c2, totalT, totalFlight;
    totalFlight = LocalTime.of( hour: 0, minute: 0);
    for (int i = 0; i < times.size(); i++) {
        c1 = LocalTime.of( hour: Integer.parseInt(times.get(i)[0])/100, minute: Integer.parseInt(times.get(i)[0])%100);
        c2 = LocalTime.of( hour: Integer.parseInt(times.get(i)[1])/100, minute: Integer.parseInt(times.get(i)[1])%100);
        totalT = LocalTime.of(c2.plusHours(-c1.getHour()).getHour(), c2.plusMinutes(-c1.getMinute()).getMinute());
        totalFlight = LocalTime.of((totalFlight.plusHours(totalT.getHour()).getHour()), (totalFlight.plusMinutes(totalT.getMinute()).getMinute()));
    }
    total = totalFlight.getHour()*100 + totalFlight.getMinute();
    return total;
```

The connecting time is calculated as a difference in time between the arrival time of last flight and departure time of next flight using the Local Time class (it was used for air time as well).



```
@Override
public int connectingTime() {
    int total = 0;
    LocalTime cbeg, cend, totalT;
    cend = LocalTime.of( hour: Integer.parseInt(times.get(0)[1])/100, minute: Integer.parseInt(times.get(0)[1])%100);
    for (int i = 1; i < times.size(); i++) {
        cbeg = LocalTime.of( hour: Integer.parseInt(times.get(i)[0])/100, minute: Integer.parseInt(times.get(i)[0])%100);
        totalT = LocalTime.of(cbeg.plusHours(-cend.getHour()).getHour(), cbeg.plusMinutes(-cend.getMinute()).getMinute());
        total += totalT.getHour()*100 + totalT.getMinute();
        if (total%100 >= 60)
            total += 40;
        cend = LocalTime.of( hour: Integer.parseInt(times.get(i)[1])/100, minute: Integer.parseInt(times.get(i)[1])%100);
    }
    return total;
```

The time spent for travel is calculating by just adding the air and connecting times together.

Conclusion

Given more time the program could be further improved. Further classes and methods implemented to expand its functionality. It is worth to note that in some cases it has been perceived that different devices provide different results for the same input. The issue has been addressed and corrected, though may still behave in such fashion.