CSE 241 Algorithms and Data Structures **Lab 3a**

Using Shortest Paths to Plan an Itinerary

Assigned: 10/31/2012 *Due Date: 11/28/2012*

Important Note: Lab 3 is divided into two parts. You are now reading Part (b), which is worth 75% of the total credit for the lab, plus up to 15% extra credit.

Both parts of Lab 3 are formally due at the same time. **Important: You will need a working heap code from Part (a) to complete Part (b). You should do Part (a), which is described in a separate handout, FIRST. You need to copy all your code for a correctly working Lab 3a, except Lab3a.java, into your Lab 3b direcotry, before you start on Part (b).**

1 Problem Description

We are given a file `flights.txt` containing a schedule of airline flights, each of which consists of a name (e.g., Continental 74) and the following four values:

- *startAirport* – airport from which the flight departs
- *endAirport* – airport at which the flight arrives
- *startTime* – time at which the flight leaves
- *endTime* – time at which the flight arrives

Airports are specified by their 3-letter codes; you can find a list of these codes in the file `airports.txt` included with the test dataset. If you don't recognize some of the codes, Google will lead you to sites that will translate them. Flight departure and arrival times are given relative to the local time zone of the start and end airports, but the provided code converts these into Greenwich Mean Time (GMT) for you using the time zone info in the airport file.

We want to find a way to fly from some starting airport (say, STL) to any of a number of destination airports. We particularly want to find the *shortest path in terms of total travel time.* We'll start out with a simple version of this problem that ignores the precise schedule constraints of each flight, then add more realism later.

2 The Task: Schedule-Oblivious Shortest Paths (75%)

For this part of the lab, you can ignore the starting and ending times of each flight and concentrate only on their lengths. With this simplification, the problem reduces to finding the shortest path from the source to the destination in the (multi)graph with one vertex per airport and one weighted, directed edge per flight.

2.1 Multigraph Data Structure

I have provided a reasonable multigraph implementation in the file `Multigraph.java`. A `Multigraph` is a linear array of vertices, each of which has zero or more adjacent edges. Important methods associated with an object G of type `Multigraph` include:

- `G.nVertices()` – return the number of vertices in G
- `G.get(i)` – return the ith vertex in G (starting from 0)

Each vertex of G has type `Vertex` and contains a unique integer identifier (the same as its index used for `G.get()`) and a list of adjacent edges. Important methods associated with an object v of type `Vertex` include:

- `v.id()` – return the integer identifier for v
- `v.adj()` – return an iterator t of type `Vertex.EdgeIterator` that permits enumeration of all edges adjacent to v. To see if there are more edges to process, call `t.hasNext()`; to get the next edge, call `t.next()`.

Each directed edge of G has type `Edge` and contains four values: references to the vertices at its two endpoints, an integer weight, and a unique integer identifier. Important methods associated with an object e of type `Edge` include:

- `e.id()` – return the integer identifier for e
- `e.from()` – return reference to the "from" Vertex of e
- `e.to()` – return reference to the "to" Vertex of e
- `e.weight()` – return the weight of e

The C++ version of the `Multigraph` structure is similar but uses pointers in a few places instead of references. See the class definitions for details.

2.2 Mapping Between Problem and Multigraph

We have provided driver code to read the airport and flight lists in from files, construct a multigraph from them, and process queries from a file. Airport and flight information is all stored in an object of type `Input`; for details, see `Input.java` and its use in the main driver code. The `Input` object stores three things: an array of all airports, an array of all flights, and a mapping from airport codes to their indices in the airport array.

Besides simply reading in the data, the driver code builds a weighted multigraph from it as follows. First, all times are converted to GMT. Second, a vertex is added to the graph for every airport. Finally, an edge is added for every flight connecting its departure and arrival airports, with weight equal to its length (in minutes).

For convenience in printing, the integer identifier for each vertex is its airport's position in the airport array, while the identifier for each edge is its flight's position in the flight array.

2.3 What To Do

To get this part of the lab working, you need to fill in the methods in `ShortestPath.java`, which the driver code calls to get the shortest path between airports. There are two methods to be implemented in the `ShortestPaths` class:

- `ShortestPaths(Multigraph G, int startId, Input input, int StartTime)` (the constructor): given a Multigraph G and a starting vertex identifier i, compute weighted shortest paths from the ith vertex of G to all other vertices. You must *store* the result of this computation for later querying. For now, ignore the last two parameters of the constructor.
- `int [] returnPath(int endId)`: given an ending vertex identifier j, find a shortest path from i to j and return the identifiers of all *edges* on this path (ordered from first to last) in an array. Here, i is the starting vertex specified in the constructor. If the start and end are the same, just return an array of length 0 (do *not* return null).

The interface specs are similar for C++, except that the `returnPath()` function returns a `vector<int>` rather than an array.

To compute shortest paths in the constructor, you should use Dijkstra's algorithm, implemented on top of the `PriorityQueue<T>` class you wrote for Part (a). Calls to `returnPath()` should then run in time proportional to the length of the returned path.

2.4 Testing Your Lab

The driver for Lab 3b takes three arguments: the airport file, the flight file, and a *query file* with one or more queries. A query is a single line of the text with (for now) the form

```
0 <start> <end> [<end> ...]
```

where *start* is a starting airport, and each *end* is an ending airport. (The first value on the line must be 0.) Given such a query, the driver will use your shortest path code to find and print shortest paths, in terms of total flight time, between the start and each of the ends. The autograder will test your program on a number of queries; feel free to test with others yourself.

What to Turn In

As for previous labs, your repository should contain your code, a correctly modified control file, and a README describing anything interesting you encountered in the course of implementing your lab. Feel free to copy any source files you need from your Lab 3a repository to your Lab 3b repository as part of your implementation.

3 Extra Credit: Schedules Matter! (15%)

The abstraction of the previous section isn't a very good model of real air travel. We are ignoring the fact that flights leave at a small number of fixed times, and that a given itinerary is not valid unless you can make all required connections.

To make the problem more realistic, we impose the following scheduling constraints. First, each individual leg of the itinerary (including the first!) must be preceded by a layover of at least $\mu = 45$ minutes, to allow for delays and so forth. Second, the actual layover time is determined by the flight schedule. Hence,

1. If it is 1700 hours (5:00 PM) and a given flight leaves at 2100 hours (9:00 PM), then the cost of taking that flight is $21 - 17 = 4$ hours *plus* the actual flight time.

2. If it is 1700 hours and a given flight leaves at 1500 hours (3:00 PM), then you must wait until the following day, and the cost is $15 - 17 + 24 = 22$ hours plus the actual flight time. If we arrive at the airport at 1700 hours, this rule applies to every flight leaving prior to 1745 hours (5:45 PM) because of our required layover of at least $\mu = 45$ minutes.

The problem is now as follows: given that we arrive at the starting airport at some hour of the day h_0, find an itinerary that gets us to the destination airport in the *least total time, including all layovers.*

For this section, the 0 at the beginning of each query is replaced by the time h_0, expressed in a 4-digit number. For example, 1700 means 5:00 PM, while 0945 means 9:45 AM. All times in the query files are assumed to be GMT.

3.1 What To Do and Turn In

Modify the Lab 3 code to make the shortest path engine schedule-aware. Use the last two arguments of the `ShortestPath` class constructor to implement the rules described above to find the shortest feasible flight schedule, including layovers, from the source to each reachable destination. The time passed into the constructor is given in minutes since midnight GMT, which is also the format used to store all flight times internally, so you shouldn't have to do any yucky time zone conversion. Use edges' unique identifiers to look up their flights' start and end times in the provided `Input` object.

if the startTime argument passed to the constructor is 0, you should use the non-layover-aware shortest path algorithm of the previous section. Use the new code only if this time is a non-zero value. If you want, you can put your original and modified shortest path computations in two functions in the `ShortestPaths` class and call one or the other depending on whether this argument is 0.

Please check in your modified code, and enable the `extra` directive in the control file to ensure that the new mode is tested. Please also describe your extensions in the README.

3.2 Modifying the Shortest Path Computation

As part of this assignment, you should figure out how to make Dijkstra's algorithm handle the extra constraints imposed in this section. Here are a couple of hints to get you started:

- You should not have to change your priority queue implementation, or the basic outline of Dijkstra's algorithm.
- Consider a shortest path from the starting airport s to some destination u, and let y be any intermediate airport on this path. How does the shortest-path distance from s to u compare to that from s to y? Does the correctness proof for Dijkstra's algorithm still go through?
- Suppose you arrive at an airport h minutes after midnight and want to take a flight that departs h' minutes after midnight. Then the layover time is given by $\mu + (h' - h - \mu + 2880) \bmod 1440$. (The extra 2880 guarantees that the left-hand side of the mod is non-negative, since not all languages have the same semantics for mod applied to a negative number.)