1 Overview

The project URL for this lab is

https://svn.seas.wustl.edu/repositories/yourid/cse241_fl15/paths

where “yourid” is the name of your 241 SVN repository.

This lab gives you experience with shortest-path finding and the underlying priority queue data structure. The application is in planning flight itineraries for air travel.

Please start early so that you can get help if you need it!

2 Problem Description

Booking air travel can be a complex problem, especially between cities not connected by a direct flight. For this lab, we are going to take a “slice” of the full route planning problem faced by companies like Expedia, Orbitz, and PriceLine, as well as (the few remaining) human travel agents. In particular, we will consider the problem of plotting a feasible sequence of flights from a starting city to any of a number of destinations in a way that minimizes total travel time.

More concretely, 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 data set. 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 zones in the airport file.

Our goal is as follows: given a starting airport (say, STL), compute sequences of flights to all other airports reachable from the start, such that each such sequence minimizes total travel time to its destination. To simplify hugely, we will start by ignoring the scheduled takeoff and landing times of each flight and pretend that they are all “charter” flights, i.e. that they will take off immediately whenever the traveler is ready. With this simplification, the total travel time for a sequence of flights is simply the sum of times for each component. (We’ll address the fact that flights leave at fixed times each day in the extra credit portion of the lab.)

To map our problem to a well-known graph problem, consider the directed graph whose vertices are airports and whose edges are flights, with each edge weighted by its travel time (in minutes). Technically, this construction is a multigraph, as there can be two or more direct flights between any two airports. A path of minimal total travel time from airport $i$ to airport $j$ is then a shortest path from $i$ to $j$ in the sense of Dijkstra’s algorithm. Each edge is its flight’s position in the flight array.

Your job is to implement Dijkstra’s algorithm, along with its supporting priority queue, over the graph of airports and flights to find the most efficient routes from a starting airport to each destination.
3 Provided Code and Data Structures

The provided code reads the airport and flight lists from files and stores them in an object of type Input (which you can treat as opaque for now). This object is then used to construct a Multigraph object $G$, which is the main data structure you will need for Dijkstra’s algorithm.

A Multigraph is logically an array of Vertex objects, each of which has zero or more adjacent Edge objects. 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 $i$th vertex in $G$ (starting from 0)

A Vertex contains a unique integer identifier (the same as the index passed for $G.get()$ to retrieve it) 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 Edge objects adjacent to $v$. To see if there are more edges to process, call $t.hasNext()$; to get the next Edge, call $t.next()$.

An Edge 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++ versions of these structures are similar but use pointers in a few places instead of references; see Multigraph.h for details.

4 What To Do

4.1 Shortest-Path Computations

To implement the required shortest-path computations, you need to fill in the methods in ShortestPath.{java,cc}, which the driver code calls to get shortest paths between airports. Java users must implement two methods in the ShortestPaths class:

- $ShortestPaths(Multigraph G, int startId, Input input, int StartTime)$ (the constructor): given a Multi-graph $G$ and a starting vertex identifier $i$, compute weighted shortest paths from the $i$th vertex of $G$ to all other vertices. You must store the result of this computation inside your ShortestPaths object 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, please allocate and 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.

Efficiency

You must use Dijkstra’s algorithm to compute shortest paths in the constructor. Calls to returnPath() should then run in time proportional to the length (i.e. number of edges) of the returned path.
### 4.2 The Supporting Priority Queue

To support Dijkstra’s algorithm, you need a min-first priority queue over vertices. Since a priority queue is a generically useful data structure, this lab asks you to implement a template/generic class `PriorityQueue<T>` that uses integer keys but stores records of an arbitrary type `T`. You can then instantiate this class, setting `T` as needed for your application.

Your queue should support the standard insertion and extraction operations, plus the `decreaseKey` method needed to update vertices as the shortest-path algorithm runs. Note that your priority queue must hold not only keys but their associated records of type `T` as well. This shouldn’t require any massive re-engineering of the basic data structure: when you would ordinarily move a key, just move the associated record along with it. Furthermore, to make use of `decreaseKey`, you’ll need to implement a `Handle` data type (See Section 4.2.1 below) that you can use to track the position of elements in the priority queue dynamically.

You will be modifying the priority queue template in `PriorityQueue.<java,h>`. Your `PriorityQueue<T>` class should support the following methods:

- a constructor that creates an empty priority queue
- `boolean isEmpty()` – return true iff the queue contains no elements.
- `Handle insert(int key, T value)` – insert the key `key` into the queue, along with the associated object `value`. Return a `Handle` object that refers to the pair `(key, value);` the handle will be used by `decreaseKey` to find the pair later.
- `int min()` – return the smallest key in the queue.
- `T extractMin()` – remove the `(key, value)` pair with the smallest key from the queue and return its `value` component.
- `boolean decreaseKey(Handle h, int newkey)` – attempt to decrease the key of the pair referenced by the `Handle h` to the value “newkey.” If `h` refers to a `(key, value)` pair that has been removed from the priority queue, or if `newkey` is >= than the pair’s current key, do not modify the queue, and return `false` to indicate that nothing has changed. Otherwise, replace the pair’s key with `newkey`, fix up the queue, and return `true`.
- `int handleGetKey(Handle h)` – return the `key` component of the pair pointed to by the handle `h`. The result is undefined if `h` refers to a pair no longer in the queue.
- `T handleGetValue(Handle h)` – return the `value` component of the pair pointed to by the handle `h`. The result is undefined if `h` refers to a pair no longer in the queue.
- `String toString()` – print a list of pairs of the form “`(key, value)`” for every element in your priority queue. Please print the pairs starting with cell 1 of the array and going up to the highest-numbered cell. Use the `toString` method of each `value` object to print it.

C++ users will need to modify the above specification slightly. The most important change is that `Handles` are passed explicitly by reference (Java does this implicitly); hence, you should substitute `Handle *` wherever you see `Handle` above. (The value parameter of type `T` is passed by value.) The other change is that you should implement an output operator `<<` for your queue instead of a `toString` method. Because C++ objects don’t have a default `operator<<` function, you’ll need to implement such a function for any type you use to parameterize your priority queue.

**Note:** the above design is not very clean because `Handle` should really be an opaque type that can return its key and `value`, rather than just asking the priority queue to do it. The reason I’m not asking for the nicer design is that the internals of the `Handle` would have to be exposed to your `PriorityQueue` but hidden from the rest of the world. You can do this using, e.g., the `friend` declaration in C++ or the package facility in Java, but it’s a detail that you shouldn’t worry about right now.
Efficiency

You should use an array-based binary heap to implement your priority queue. Your `insert`, `extractMin`, and `decreaseKey` methods should all run in worst-case time \( O(\log n) \), where \( n \) is the number of elements in the queue. The `isEmpty`, `min`, `handleGetKey`, and `handleGetValue` methods should all run in constant time. Your queue should be able to hold arbitrarily many elements; you should implement size doubling (either explicitly or via a language-supplied resizable array type) to meet this requirement.

### 4.2.1 Advice on Implementing Heaps with Keys, Values, and Handles

A **Handle** is just a piece of information that lets you find where a given key currently appears in the heap. For an array-based heap, it suffices to maintain an integer index that points to the cell containing the key. You'll need to modify the index stored in a **Handle** from inside your **PriorityQueue** class whenever it changes, that is, whenever the key referenced by the **Handle** moves up or down the heap. You should also invalidate a **Handle** when its key is deleted from the heap, so that it can be recognized as invalid if it is later passed to `decreaseKey`.

Any of your methods that modify the heap must be able to move a \((key, value)\) pair from one heap node to another and update the associated **Handle**. You'll probably want to keep track of key, value, and handle together in each heap node. To make sure that all three items always get moved together, I suggest implementing a **swap** method that takes two heap nodes, exchanges all their data at once, and updates their **Handles**. If you use only this method to move data around in the heap, you won’t have to worry about copying all the right fields. You can implement all the heap rearrangements we talked about in class in terms of your **swap** operation.

One final bit of advice: there are two common design patterns for a heap with handles. Some people make their heap an array of handle objects, while others create an element data structure, separate from the handle, that holds the key and record and a handle reference. Either is fine. Please be aware that if you use the first approach in Java, you may need to uncomment the warning suppression directive at the top of `PQTest.java` to get your code to compile and pass the autograder.

### 4.3 Testing Your Lab

The driver for this lab operates in two modes. If presented with the explicit string `pqtest` as the first argument, followed by an integer between 1 and 3, it launches one of several tests (found in `PQTest.{java,cc}`) that exercise your priority queue implementation (but not your shortest-path code). The first few autograder test cases run these tests. Note that different implementations may not produce exactly the same heap, but the order of the extracted keys and the contents associated with each handle should always be the same.

Otherwise, the driver takes three arguments: the airport and flight files described above, 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 such queries.

### 4.4 What to Turn In

To complete the lab, you must ensure that your SVN repository contains all of the following:

- Your code, in particular your implementation of the **PriorityQueue** and **ShortestPaths** classes and any auxiliary classes.
- A brief document describing your implementation (you need not recapitulate the material we covered in class) and anything else interesting you noticed while implementing the algorithms.

If your implementation is buggy and you were unable to fix the bugs, please tell us what you think is wrong and give us a test case that shows the error.

Put this document in your project repository in the same directory as your code. The file should be in PDF format and should be named `README.pdf`. 
We will test your implementation’s correctness after it is turned in by running a suite of test cases on the code in your repository. You may not have access to all the test cases that we use for validation.

5 Extra Credit: Schedules Matter! (10%)

The pure shortest-path problem isn’t a very good model of the itinerary planning problem because it ignores the fact that flights leave at only a small number of fixed times, and that a given itinerary is not valid unless you can make all required connections. In this section, you will augment your lab to observe these constraints while still finding a feasible sequence of flights occupying the least total travel time.

We impose the following scheduling constraints. First, each individual leg of an itinerary (including the first!) must be preceded by a layover of at least $\mu = 45$ minutes, to allow for flight 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 you 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 you arrive at the starting airport at some time of day $t_0$, 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 initial time of day $t_0$, expressed as 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.

5.1 What To Do and Turn In

Modify your lab 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 starting time of day 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.

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 modified implementation only if this argument is non-zero. If you want, you can put your original and modified shortest path computations in two separate functions in the ShortestPaths class and call one or the other depending on whether the argument is 0.

You will need to consider for each edge the takeoff time of its associated flight. To access this time, you can access flight data via the Input object’s flights[] field, which is an array of all flights. The flight corresponding to an edge with identifier $i$ can be accessed as Input.flights[i]. The elements of the array are objects of type Flight with an integer field startTime giving the starting time of the flight in minutes past midnight GMT. For more information, see the implementation of the Input class.

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

5.2 Modifying the Shortest Path Computation

As part of the extra credit work, you must figure out how to make Dijkstra’s algorithm handle the extra constraints imposed above. 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 \( \mu + (h' - h - \mu + 2880) \mod 1440 \). (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.)