Project 2

Due Date
5-9-16 11:59pm
(no late submissions accepted)

For this assignment you will be implementing a graph modeling flights between airports. Flights will have a start time, end time, and financial cost. The first part of the assignment is to simply construct the graph and be able to list flights to and from each airport. The second part of the assignment is to construct itineraries which optimize certain criteria.

1 Graph Overview

A graph is mathematically represented as a collection of Vertices and a collection of Edges. $G =< V, E >$ As indicated in class each Edge is represented as a pair (v_1, v_2) of vertices.

Often we define labeling function L_V and L_E which associated a collection of data with each node and each edge. e.g. $L_V(v_1) =< dataofv_1 >$ and $L_E((v_1, v_2)) =< dataofedge(v_1, v_2) >$.

When it comes to implementation of a graph we have a core programming interface

```
The Graph ADT
createGraph  : () -> (Graph *)
destroyGraph : (Graph *) -> ()
newVertex    : (Graph *, VData *) -> (Vertex)
getVData     : (Graph *, Vertex) -> (VData *)
getVertexOf  : (Graph *, VData *) -> (Vertex)
allVertices  : (Graph *) -> (Vertex List)
newEdge      : (Graph *, EData *, Vertex, Vertex) -> (Edge)
getEData     : (Graph *, Edge) -> (EData *)
getEdgeOf    : (Graph *, EData *) -> (Edge)
getSource    : (Graph *, Edge) -> (Vertex)
getTarget    : (Graph *, Edge) -> (Vertex)
allEdges     : (Graph *) -> (Edge List)
```

```
getIncidentEdges     : (Graph *, Vertex) -> (Edge List)
getAdjacentVertices  : (Graph *, Vertex) -> (Vertex List)
```

The `Vertex` and `Edge` are treated as handles or references. We assume that there is a strict correspondence between a vertex/edge and the instances of data used to label the vertex/edge. The graph should be able to convert a reference to the data into the right vertex/edge and take the vertex or edge and retrieve the right data.

2 The Problem Description

In this section we will present the commandline interface for the problem. You can decide how much of the Graph ADT you want to implement in order to solve the problem. For this problem, even if you do not use the Graph ADT, much of the work you will need to do to solve the problem will be the same work. The Graph ADT can help keep your thoughts organized.

Command Line Interface

```
Flight <APC1> <APC1> <Cost> <start time> <end time>
List <APC>
List *
Cheapest <APC1> <APC2>
Earliest <APC1> <APC2>
```

The `<APC>` is a 3 character airport code, e.g. LAX, SFC, SEA, PRT, LBB, ONT,...

The `<Cost>` is a positive integer

The `<start/end time>` is a positive integer between 0 and 1000.

The `Flight <APC> <APC> <Cost> <start time> <end time>` command

- Get the vertices for the airport codes. (If they don't exist yet, create new vertices)
- Create a new edge from `<APC1>` to `<APC2>` annotated with the cost, start and end time.

The `List <APC>` command

- Prints `Airport <APC> Not Found` on its own line when no vertex with the same name is in the graph.

- `printf("Flights From %s\n", <APC>);`
- Get the list of edges for the vertex labeled by `<APC>`
- Print the flight data annotating each edge in alphabetic order of target airport codes, then by start time.
- `printf("-- Flight to %s cost:%d start:%d end:%d\n", APC2, cost, start, end);`

The `List *` command

- Prints `Network Is Empty` on its own line when no vertices in graph.
- Get the list of vertices, sorted in alphabetic order of airport codes.`<APC>`
- For each vertex,
- `printf("Flights From %s\n", <APC>);`
- Print the flight data annotating each edge in alphabetic order of target airport codes.
- `printf("-- Flight to %s cost:%d start:%d end:%d\n", APC2, cost, start, end);`

The `Cheapest <APC1> <APC1>` command

- Use Dijkstra's Algorithm to compute the trip from `APC1` to `APC2` which minimizes the cost.
- Print: `Cheapest From:<APC1> To:<APC2>`
- Print the flights in temporal order, from start to final end, 1 flight per line:
- `Flight <APC> <APC> cost:<Cost> start:<start time> end:<end time>`

The `Earliest <APC1> <APC1>` command

- Use Dijkstra's Algorithm to compute the trip from `APC1` to `APC2` which gives the earliest time of arrival.
- Print: `Earliest From:<APC1> To:<APC2>`
- Print the flights in temporal order, from start to final end, 1 flight per line:
- `Flight <APC> <APC> cost:<Cost> start:<start time> end:<end time>`

3 Dijkstra's Algorithm

Dijkstra's Algorithm (using Adaptable Priority Queue) will find the shortest path from a given node to every other node in the graph that is reachable from the start node. The APQ gives higher priority to smaller cost/valuation. Simply reverse the parent child order so that the smaller value is in the parent.

1. Starting Condition
   - Add the start vertex to the APQ with priority/cost/endtime = 0
   - Add every other vertex to the APQ with priority/cost/endtime = some MAX integer value
   - Make sure you can store the current best edge and total-cost/final-endtime for getting to each vertex.
2. Repeat Until APQ is empty or until end vertex removed from APQ
   (a) Let CV = The vertex with least cost/endtime by dequeueing from APQ
   (b) For each edge incident to CV (CV is the source vertex)
      i. If the edge's flight starts earlier than CV's best flight's end time, skip the edge.
      ii. Otherwise check if the edge's flight's total-cost/final-endtime for the target vertex is better and update the best flight and cost/endtime for the target if it is better. (update the APQ entry for target with new cost/endtime).
3. Once the final destination vertex is removed the APQ, the proper itinerary is recovered by walking from the end to the start vertices following the source of the best flights to each vertex along the way.
4. To print the flights in order, you can use recursion to walk the best edges from end to start until the start is reached and print the flight of each edge after the recursive call returns.

4 Recommended Approach

1. Implement main.c code for handling commandline interface.
2. Core Implementation of Graph and get List command to print properly. (80
   (a) Write out data structures for vertex, edges, and their data elements.
   (b) Write list utilities for list of edges and list of vertices.

(c) (One way to re-use list is to have all lists contian (void*) pointers and then case to the specific data structure type when needed and safe.)

3. Add APQ and make alterations as needed to facilitate Dijkstra's algorithm.

5 Example

Input Commands

```
List *
Flight LAX PTL 100 5 200
List LAX
List SEA
List PTL
Flight LBB SFE 50 300 400
Flight LBB PHE 200 50 250
Flight LBB SFE 45 400 500
Flight PHE LAX 200 300 400
Flight LAX PTL 400 450 500
Flight PTL SEA 900 800 850
Flight SFE PTL 100 550 700
Flight PTL SEA 700 750 900
Flight SEA PTL 100 850 900
List *
Cheapest LBB SEA
Earliest LBB SEA
```

Output

```
Network Is Empty
Flights From LAX
-- Flight to PTL cost:100 start:5 end:200
Airport SEA Not Found
Flights From PTL
Flights From LAX
-- Flight to PTL cost:100 start:5 end:200
-- Flight to PTL cost:400 start:450 end:500
Flights From LBB
-- Flight to PHE cost:200 start:50 end:250
-- Flight to SFE cost:50 start:300 end:400
-- Flight to SFE cost:45 start:400 end:500
Flights From PHE
```

```
-- Flight to LAX cost:200 start:300 end:400
Flights From PTL
-- Flight to SEA cost:900 start:800 end:850
-- Flight to SEA cost:700 start:750 end:900
Flights From SEA
-- Flight to PTL cost:100 start:850 end:900
Flights From SFE
-- Flight to PTL cost:100 start:550 end:700
Cheapest From:LBB To:SEA
Flight LBB SFE cost:45 start:300 end:400
Flight SFE PTL cost:100 start:550 end:700
Flight PTL SEA cost:700 start:750 end:900
Earliest From:LBB To:SEA
Flight LBB PHE cost:200 start:50 end:250
Flight PHE LAX cost:200 start:300 end:400
Flight LAX PTL cost:400 start:450 end:500
Flight PTL SEA cast:900 start:800 end:850
```