CS245-02, Spring 18

Project 4: Dijkstra's algorithm.

Due date for Part 1: 04/23/2018, 11:59pm Class HashTable due.
Due date for Part 2: 05/04/2018, 11:59pm The rest of the project due.

Overview

For this project, you will implement Dijkstra's algorithm for computing the shortest path between the nodes of the graph, along with some other helper methods and classes. The nodes are major cities in the US, and the edges are "roads" connecting them. You are a given an image of the USA map[1] and the text file "USA.txt" that contains:

- the list of major US cities and their positions on the "map",
- info about the "roads" (arcs) connecting some pairs of cities and the length of each arc.

A user should be able to interact with the program by clicking on any two cities on the "map"; the program should compute the shortest path between the cities using Dijkstra's algorithm and display the path on the map (see Figure 1). Clicking on the "Reset" button should reset everything correctly so that the user can click on any two cities again.

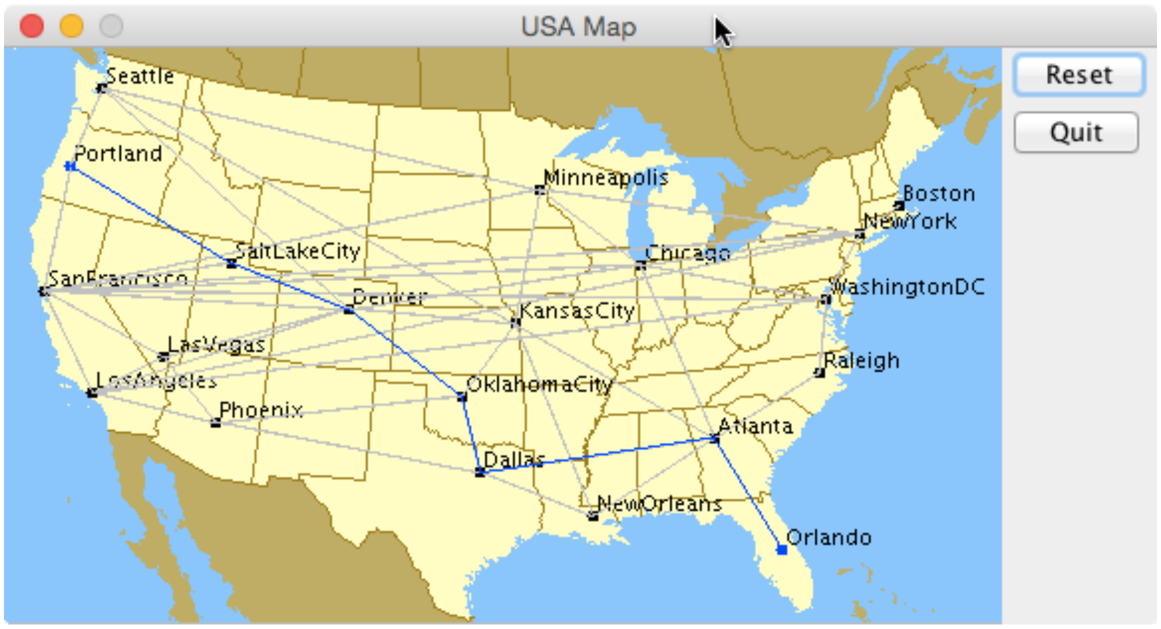

Figure 1: The user clicked on Portland and Orlando; the program computed the shortest path shown in blue.

Input File Format

[1] The image of the USA map and the graph text file are courtesy of Prof. Julie Zelenski.

You are given a file called "USA.txt" that contains the graph data in the format below. The keyword NODES is on the first line, followed by the number of nodes, followed by the lines describing each node (one node per line). Each node is defined by the name of the city, and the x and y coordinates of the city on the image. Please note that the x and y coordinates are given so that you can display the nodes on the map and click on them; they are **not** used in Dijkstra's algorithm. Then the file lists the arcs (edges), starting with the ARCS keyword. Each edge has the origin and the destination city, as well as the cost ("weight") associated with this road between the two cities.

```
NODES
<num nodes>
<name of the city> <x-coordinate> <y-coordinate>
…
ARCS
<name of the city> <name of the city> <cost>
…
```

Note that if two cities city1 and city2 are connected via an edge, this edge will show up in the input text file only **once**. Remember to add an edge going **in the opposite direction** to your graph, when you read from the file and add the data to the graph. For instance, when you read the edge "Dallas NewOrleans 340", you should add two edges to your graph: one from Dallas to NewOrleans, another one from NewOrleans to Dallas. The names of cities in the file do *not* contain white spaces, so "New Orleans" is stored as "NewOrleans" etc.

You need to read the graph data from the file and store it in a `Graph` object (see `loadGraph` method in class `Graph`). Nodes should be stored in an array called `nodes`, while graph connectivity info should be stored in `adjacencyList`. As you read cities from the file, assign an integer id to each node. Refer to the next section for details.

Graph Representation

Please refer to class `Graph` in the starter code. Your graph should store:

- An array of nodes
  `private CityNode[] nodes;`
  The index of each node in this array is the node's id and is used in Dijkstra's algorithm. The nodes are added to the array (and hence are assigned id-s) in the order in which we read them from the text file.
- An adjacency list
  `private Edge[] adjacencyList;`
  The adjacency list stores a linked list of outgoing edges for each vertex, as we discussed in class. An "edge" is defined in a separate class called `Edge`.
  Each Edge stores the *neighbor* (the id of the neighboring vertex), the *cost* and the reference to the *next* Edge in the linked list of edges.
  Note that you are **not** allowed to use class LinkedList from the Collections framework in this project.

You should also store the total number of edges (`numEdges`) and the total number of nodes (`numNodes`), although you can always access the latter as `nodes.length`.

Each node stores the name of the city and the location of the node on the map (see class `CityNode`; this class has been provided to you).

Example: For the logical graph in Figure 2a, you should store the array of nodes as shown in Figure 2b (note that for clarity, we only show the name of the city in each node in Figure 2b). Figure 2c shows the adjacency list for the graph.

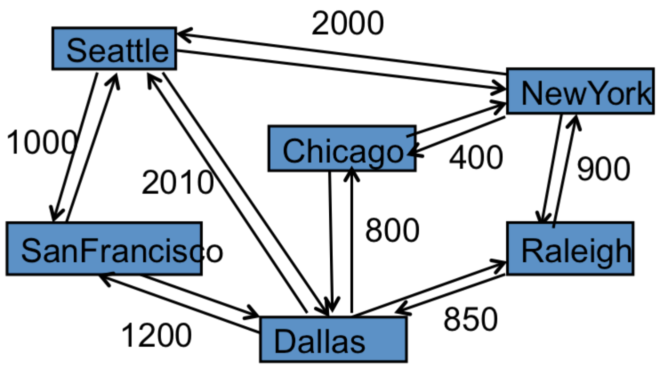

Figure 2a: The logical graph.

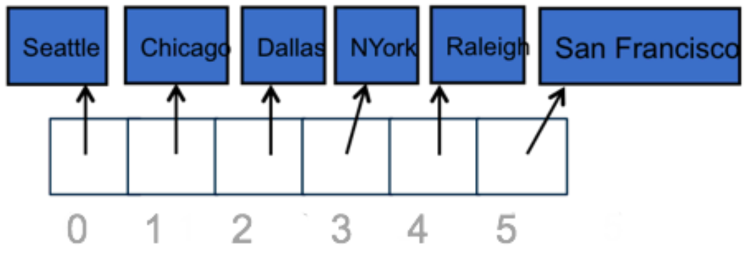

Figure 2b. The array of **nodes** for the graph in Figure 2a.

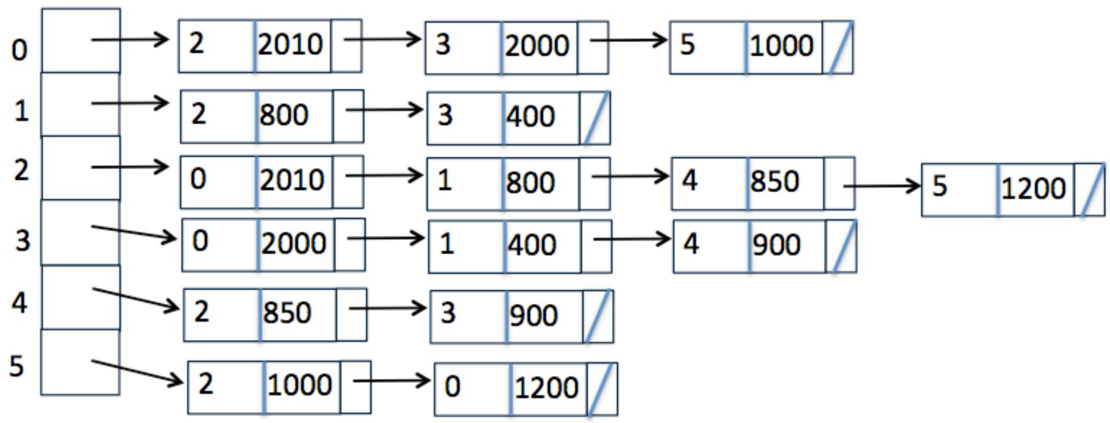

Figure 2c. The adjacency list for the graph in Figure 2a.
For each vertex vi, we store the linked list of Edge-s, where each Edge contains the id of the vertex that vi is going to, the cost, and the pointer to the next outgoing "edge" for vi. For

instance, for vertex 0, we have three outdoing edges: the one connecting this vertex with vertex 2 (with the weight of 2010), the one connecting this vertex with vertex 3 (with the weight of 2000), and the one connecting it with vertex 5 (with the weight of 1000).

You also need to fill in code in a `loadGraph` method in class Graph that loads all data from the text file into the graph; as well as fill in code in methods to add a node and to add an edge.

In order for the provided `GUIApp` class to be able to display nodes, labels, edges and the shortest path in the graph, you need to fill in code in the following methods in class `Graph`:

- `public Point[] getNodes()` - Returns an array of locations of each node.
- `public String[] getCities()` - Returns an array of city names
- `public Point[][] getPath(List<Integer> pathOfNodes)` - Takes a list of node ids on the path, and computes a list of "line segments" connecting those nodes, where for each line segment, we store an array of two points (the "beginning" point and the "end" point of the line segment).

Hash Table

You need to implement a hash table to store the elements where the **key is the name of the city**, and the value is the corresponding integer Id of the node. We should be able to query the hash table using commands such as: `int nodeId = table.get("SanFrancisco").`

For this project, you are **not** allowed to use any existing hash table implementations (such as classes HashMap, LinkedHashMap, TreeMap from Java Standard Library). You need to implement the hash table from scratch yourself. You can use any hashing strategy, though open hashing is the easiest to implement. Figure 3 shows an example of a hash table for the graph above:

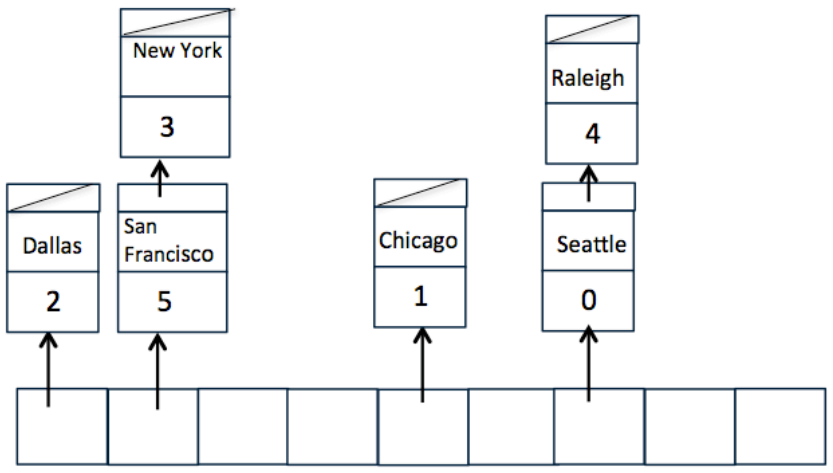

Figure 3: A hash table that uses open hashing to map keys (the names of cities) to the node id-s.

You will use this hash table in class `Graph` :

- insert keys and values into it when you read the nodes from the file,
- use the table to find the node id given the name of the city, when you read the edges from the text file.

Dijkstra's Algorithm

Once you fill out data structures and methods in the `Graph` class, you can work on the `Dijkstra` class. The user will specify the source and destination vertices by clicking on them. The provided `GUIApp` would automatically call the method `computeShortestPath(CityNode origin, CityNode destination) from the Dijkstra` class. This is where your code for the Dijkstra's algorithm is going be located.

First, create a Priority Queue (see the next section.). Then you need to create a Dijkstra table which stores:

- The current estimate of the shortest distance from the source node to every node in the graph.
- The "parent" node for each node (the previous node on the current "shortest" path).

The distances are initialized to Double.POSITIVE_INFINITY and all path entries are initialized to -1. See Figure 4.

	Cost	Path
0	0	-1
1	∞	-1
2	∞	-1
3	∞	-1
4	∞	-1
5	∞	-1

Figure 4: Dijkstra's table in the beginning of the algorithm: the estimated cost to all vertices from vertex 0 is infinity, and all the path entries have an initial value of -1.

As you iterate over the nodes to initialize the Dijkstra's table, you also need to insert an element with the nodeID and corresponding priority into the PriorityQueue.

While the priority queue is not empty, you would remove the vertex with the smallest "distance", and check/update the distances of its neighbors – **also updating the costs in the priority queue** (see reduceKey() method in the PriorityQueue class below).

Once the algorithm is done, you will need to figure out how to use the Dijkstra's table to find the indices of all the nodes on the shortest path .

Priority Queue

The Priority Queue is required to efficiently implement Dijkstra's algorithm. You are required to write this class by yourself, by using a MinHeap as the internal representation. PriorityQueue needs to have the following methods:

`void insert(int nodeId, int priority)` - Inserts nodeId with the given priority into the priority queue.
`int removeMin()` - Removes the node with the smallest priority from the queue, and returns the id of the removed node.
`void reduceKey(int nodeId, int newPriority)` Reduces the priority of the given node in the priority queue to newPriority, rearranging the queue (minHeap) as necessary.

To implement **reduceKey** efficiently**, you will need to keep track of where each element is in the priority queue.** The simplest way to do it is to store an array of pointers into the priority queue (`int[] positions`). This array should map each nodeId to the index in the min heap.

GUI (Graphical User Interface)

The GUI for this project has been provided to you in the `GUIApp` class, you do *not* need to modify it. When you first run the starter code (run class Driver), it will show you just the image of the map. In order for nodes and edges of the graph to show up, you need to write loadGraph method and the helper methods in class Graph that are called by `GUIApp` (see above).

Testing

You are responsible for testing your project. Hopefully, GUI makes testing a bit easier for this project. Test HashTable and PriorityQueue classes separately before combining them into the final project. The instructor will provide a very basic test for the project.

Commits and Code Reviews

Please note that this project has two deadlines. Only HashTable class is due by the first deadline. The **whole** project is due by the second deadline. This is a hard project, start early, especially on the second part!
You are required to have at least 5 meaningful github commits for this project, made over >= 3 days. Assignments with less than 5 commits, or all commits made in one-two days, will not be getting any credit.

We will invite several students for a code review of project Dijsktra. You need to be able to explain your code and answer instructor's questions about the code to get any credit for the project.
This project must be completed individually. You are not allowed to discuss implementation details with anybody apart from the instructor, TAs or the tutors in the CS labs.