Lab 4 Dijkstra Algorithm & Priority Queue

Project Description

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. The total travel time for a sequence of flights is simply the sum of times for each component.

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.

File Included within This Project

Main.java ShortestPaths.java PriorityQueue.java Node.java
Handle.java Multigraph.java Edge.java Vertex.java FileParser.java
Input.java PQTest.java PrettyTime.java Query.java QueryReader.java
Tokenizer.javaautorgrader-results.txt JRE System Library

Priority Queue

A. Support for Priority Queue

Before implementing the priority queue, I created a new data structure Node, which has-a T type value, an integer key and a handle as instance variables. The Node objects are elements that will be stared in the binary heap.
The Node class has methods as following:
Node(int key, T value), construct a new node.
void setKey(int key), set a new key for a Node object.
int getKey(), return an integer key.
T getValue(), return a T type value
Handle getHandle(), return a handle.

The Handle class has-a integer index as an instance variable, which locates a node's position in the binary heap. So, we can find the node in constant time. The handle class has methods as following:
void setIndex(int i)
int getIndex()

B. Priority Queue Implementation

The implementation for a priority queue is a binary heap, which is coded using an array in practice. In this case, I use an ArrayList *heap* to implement the priority queue. The methods are as following.

In the constructor, I initialize the arraylist and set the first slot to be null (heap.add(null)), thus, elements in the heap start from 1 instead of 0.

Handle insert(int key, T value)
Insert a pair (key, value) into the queue, and return a Handle to this pair so that we can find it later in constant time.
First, create a new node with the key and T type value, and add it to the end of the heap. Then, call *swap(int i, int j)* to swap the new node with its parent if there is violation due to adding the new node to the last of the heap.

void swap(int i, int j)
This method takes in indexes for two nodes which are to be swapped. After modifying the heap, this method is able to move a (key, value) pair from one heap node to another and update the associated Handle.

T extractMin()
Extract the (key, value) pair associated with smallest key in the queue and return its "value" object. First, replace the root with the last element. Then, remove the last element. Finally, heapify from the root.

boolean decreaseKey(Handle h, int newkey)
Look at the (key, value) pair referenced by Handle h. If that pair is no longer in the queue, or its key is <= newkey, do nothing and return false. Otherwise, replace "key" by "newkey", fixup the queue, and return true.

Shortest Paths

I create a new data structure *Element* in the Shortestpaths class, each Element objects are stored in priority queue for insert, extract, decreaseKey implementations. These elements are also stored in an ArrayList named store, for query.

void Initialize(PriorityQueue<Element> pq, Multigraph G, **int** startId)
Given a graph G and a start id, this method initializes every non-start elements with infinite distance, while the start element with distance 0.
Store all elements in an arraylist *store* according to their id.

The constructor constructor computes the actual shortest paths and maintains all the information needed to reconstruct them.

The returnPath() function should use this information to return the appropriate path of edge ID's from the start to the given end.