**Graphs 7**

**Dijkstra's Algorithm and Cities**

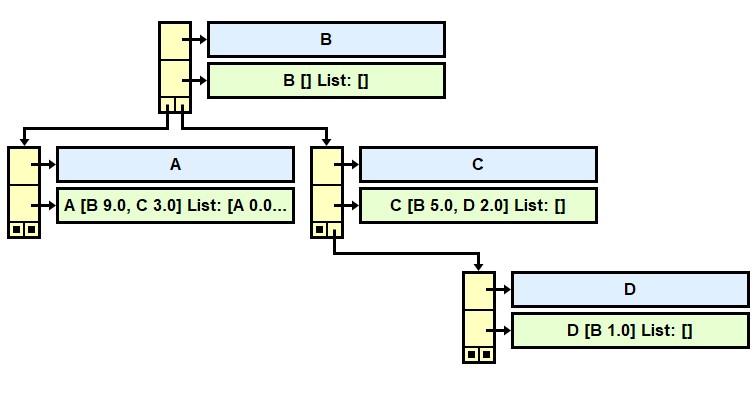
Recall that Floyd's Algorithm solved the *all-source shortest path problem*. Dijkstra's Algorithm in Graphs 6 solved the *single-source shortest-path problem.* Dijkstra's Algorithm can also report the actual shortest path, which we will do now.

We need to update our Dijkstra code to record the vertices we passed through from source to target. We do so by adding a new field previous of type wVertex in the PQelement class, like so:

class PQelement implements Comparable<PQelement>   
{  
 private wVertex vertex;   
 private Double distanceToVertex;   
 private wVertex previous; //for Dijkstra 7

(We also update PQelement's toString so that we can see the previous field.) As Dijkstra's Algorithm progresses through, it updates previous to store the old vertex. At the end, the steps in the path have been stored in PQelement, though in reverse order. You will, of course, find a way to reverse it before printing the path.

Here is the graph we used before as an example. We run Dijkstra’s Algorithm with the source “A”. The result is our familiar TreeMap of wVertex objects.



1

9

5

3

2

Unfortunately, the jGrasp picture does not show all the data in the PQelement . If we could see all the data stored at wVertex "A", we would see:

A [B 9.0, C 3.0] List: [A 0.0 Previous: null, B 6.0 Previous: D, C 3.0 Previous: A, D 5.0 Previous: C]

One of your tasks is re-write the toString in PQelement to show the word "Previous: " and to display the previous field.

Suppose this is the result in the "A" wVertex:

A [B 9.0, C 3.0] List: [A 0.0 Previous: null, B 6.0 Previous: D, C 3.0 Previous: A, D 5.0 Previous: C]

Then what is the actual path from A to B? Write the code here:

public List<String> getShortestPathTo(wVertex target)   
 {

}

In the actual Lab07, we will run Dijkstra’s Algorithm on the weighted graph of our cities.

**4** Pierre

**2** Peoria

2

3

5

3

**0** Pendleton

**5** Pittsburgh

8

**7** Pueblo

**6** Princeton

2

3

10

4

5

**3** Phoenix

5

**1** Pensacola

What is the shortest distance from Pueblo to Pendleton?\_\_\_\_ What path?\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What is the shortest distance from Pittsburgh to Phoenix?\_\_ \_\_ What path? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Assignment**

Open your old AdjListWeighted. Comment in and implement AdjListWeightedInterfaceWithCities.

Update toString method in the PQelement class so it also displays the previous field.

getShortestPathTo returns the actual path from source to target.

The readData method reads the data from two text files, the names (cities.txt) and the weights (cityEdgeListWeighted.txt.)

The driver is Dijkstra\_7\_Driver.java. You will turn in AdjListWeighted.

**Extension**

Replace the priority queue with a queue, list, set, or your own heap. Get it to work.

**Big-O Analysis**

The Big-O for Dijkstra's Algorithm depends critically on how quickly you select the next vertex. Our code used a priority queue. Java's priority queue, as you may recall, is implemented by a *heap*. Since searching through a heap is O(log V), our Dijkstra code performs in O(V\*E\*log V). If you use a regular *queue, list, or set,* Dijkstra’s algorithm requires a repeated linear search of all the vertices, resulting in O(V2 \*E).

**Sample Run**

Here is a sample run of Dijkstra\_7\_Driver:

Dijkstra's Algorithm with cities!  
Enter file of cities: cities  
Enter file of edges: cityEdgesWeighted

Enter source: Peoria

The entire graph is:

Pendleton [Pueblo 8.0] List: []

Pensacola [Phoenix 5.0] List: []

Peoria [Pueblo 3.0, Pittsburgh 5.0] List: [Pensacola 9.0 Previous: Pittsburgh, Pendleton 8.0 Previous: Pierre, Pierre 6.0 Previous: Pueblo, Princeton Infinity Previous: null, Pueblo 3.0 Previous: Peoria, Peoria 0.0 Previous: null, Phoenix 14.0 Previous: Pensacola, Pittsburgh 5.0 Previous: Peoria]

Phoenix [Pueblo 3.0, Pittsburgh 10.0] List: []

Pierre [Pendleton 2.0] List: []

Pittsburgh [Pensacola 4.0, Phoenix 10.0] List: []

Princeton [Princeton 5.0, Pittsburgh 2.0] List: []

Pueblo [Pendleton 8.0, Pierre 3.0] List: []

Data structure of the source vertex: Peoria

Peoria [Pueblo 3.0, Pittsburgh 5.0] List: [Pensacola 9.0 Previous: Pittsburgh, Pendleton 8.0 Previous: Pierre, Pierre 6.0 Previous: Pueblo, Princeton Infinity Previous: null, Pueblo 3.0 Previous: Peoria, Peoria 0.0 Previous: null, Phoenix 14.0 Previous: Pensacola, Pittsburgh 5.0 Previous: Peoria]

Distance from source to Pensacola: 9.0

path: [Peoria, Pittsburgh, Pensacola]

Distance from source to Pendleton: 8.0

path: [Peoria, Pueblo, Pierre, Pendleton]

Distance from source to Pierre: 6.0

path: [Peoria, Pueblo, Pierre]

Distance from source to Princeton: Infinity

path: [Princeton]

Distance from source to Pueblo: 3.0

path: [Peoria, Pueblo]

Distance from source to Peoria: 0.0

path: [Peoria]

Distance from source to Phoenix: 14.0

path: [Peoria, Pittsburgh, Pensacola, Phoenix]

Distance from source to Pittsburgh: 5.0

path: [Peoria, Pittsburgh]

**Good Practice**

Trace Dijkstra's Algorithm on the Pittsburgh-Pensacola-Phoenix triangle. Show how the minimum-List are updated.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pittsburgh | Pensacola | Phoenix |
| start v=Pittsburgh |  |  |  |
| after v= |  |  |  |
| after v= |  |  |  |
| after v= |  |  |  |

Pittsburgh

4

10

Phoenix

5

Pensacola

Record the processing of the PQelements and how each minimum distance is calculated.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pit  0 |  |  |  |  |  |

Here is the priority queue pq:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |