**Graphs 6**

**Weighted Edge List: Dijkstra's Algorithm**

The general problem is to find the shortest distances between two locations. There are several approaches:

* brute force, try all paths: it works, but is slow. Floyd's is an example. O(V3)
* DFS (depth-first search): gives an answer, but not necessarily the shortest. O(V + E)
* BFS (breadth-first search): give the shortest path in the sense of the smallest number of vertices or edges, but not necessarily the shortest weight (or distance) path. O(V + E)
* the *greedy* approach: always take the shortest outgoing path, without any long-term planning. Dijkstra's Algorithm is a classic example of a greedy algorithm. It is also an example of *dynamic programming*, meaning, to arrive at the answer by successive approximation. If we use a priority queue, the Big-O is O(V+E\*log V)

Let’s develop Dijkstra’s algorithm. The internet has many discussions, videos, and demonstrations of Dijkstra's algorithm. Here some good demonstrations:

<https://www.youtube.com/watch?v=UG7VmPWkJmA>

<https://www.youtube.com/watch?v=8Ls1RqHCOPw>

<https://www.youtube.com/watch?v=pVfj6mxhdMw>

The general idea is to keep both edge-distances and a minimum distance from the source to every other vertex. As the vertices are processed, the distances back to the source are updated with shorter distances, if possible. The algorithm runs faster if the next vertex to be processed is always the smallest distance of the available vertexes (that's the priority queue part).

Look at this graph and visually calculate the shortest distance from A to B. \_\_\_\_

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | A | B | C | D |
| start here | 0 | ∞ | ∞ | ∞ |
| after v=A |  |  |  |  |
| after v=C |  |  |  |  |
| after v=D |  |  |  |  |
| after v=B |  |  |  |  |

9

1

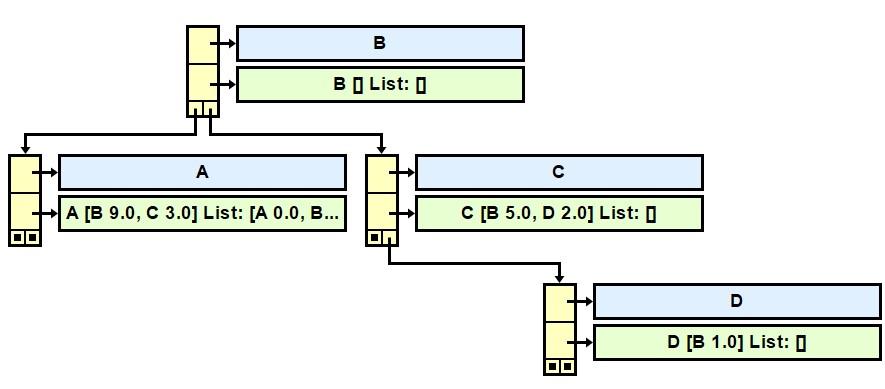
2

3

5

Dijkstra's Algorithm calculates the shortest distance from some source vertex to every other vertex. Let's say the source is A. The distances to the other vertices start at infinity. Take the not-yet-visited neighbor that is closest to A and use its edge-distance to compute, if possible, a new shorter distance back to each already-processed vertex. Somehow, update the new distance for each vertex. Repeat until all the vertices have been visited. At the end, A "knows" that A's distance to itself is 0, that A's distance to B is 6, that A's distance to C is 3, and that A's distance to D is 5. Genius.

What data structure can support all this? There are several. We will use this one:



I hope you see a

Map<String, wVertex> vertexMap = new TreeMap<String, wVertex>();

I hope you see, in A, the beginnings of this List: [A 0.0, B 6.0, C 3.0, D 5.0]. That's the answer. That's the minimum distances from A to every other vertex.

I hope you see that A (the blue box) maps to a weighted vertex wVertex (the green box). A wVertex is somewhat like a Vertex:

1. wVertex stores its own name.
2. wVertex has a Set that stores Neighbor objects.
3. wVertex has an arrayList of PQelement.
4. Here is the wVertexInterface:

public String getName();  
public ArrayList<PQelement> getAlDistanceToVertex();  
public PQelement getPQelement(wVertex v);  
public Double getDistanceToVertex(wVertex v);  
public void setDistanceToVertex(wVertex v, double m);  
public Set<Neighbor> getNeighbors();   
public void addAdjacent(wVertex v, double d);   
public String toString();

We need a new class, called the PQelement class.

1. PQelement stores a wVertex.
2. PQelement stores the distance to that wVertex.
3. PQelement implements a compareTo method, based on distance, to make the priority queue work.
4. PQelement has a toString method which you will need to implement.

We need a new class, called the Neighbor class. (Some programmers just use a Map<wVertex,Double>.)

1. Neighbor stores a wVertex object.
2. Neighbor stores the distance to that wVertex object.
3. Since we will put Neighbor objects into Sets, Neighbor needs to function in HashSets and TreeSets. Neighbors are "equal" if they have the same name. Make it so.

All these classes need constructors, accessors, modifiers, and toString.

Let's develop Dijkstra’s algorithm in more detail, this time including the priority queue:



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A  0 |  |  |  |  |  |  |  |  |  |

9

1

5

2

3

|  |  |
| --- | --- |
| A has a list of Neighbors (that is, neighboring wVertexes and edgeDistances).  In the wVertex A (the source) create an arrayList of PQelements, one PQelement for each wVertex, all set to infinity except the one for A which is set to 0  Add the PQelement corresponding to the source (A) to the pq | A🡪 [ A|0 B|∞ C|∞ D|∞]  pq🡪 [ A|0] |
| Loop while the pq still has elements |  |
| The nearest vertex, A, is removed from the pq. | A🡪 [ A|0 B|∞ C|∞ D|∞]  pq🡪 [] |
| A's Neighbor B is processed. Since 0+9 < ∞, remove B and update B's   distanceToVertex to 9. B is re-placed in the pq | A🡪 [ A|0 B|9 C|∞ D|∞]  pq 🡪 [ B|9] |
| A's Neighbor C is processed. Since 0+3 < ∞, remove C and update C's   distanceToVertex to 3. C is re-placed in the pq | A🡪 [ A|0 B|9 C|3 D|∞]  pq 🡪 [ B|9 C|3] |
| The nearest vertex, C, is automatically removed from the pq. | A🡪 [ A|0 B|9 C|3 D|∞]  pq 🡪 [ B|9] |
| C's Neighbor B is processed. Since 3+5 < 9, remove B and update its   distanceToVertex to 8. B is re-placed in the pq. | A🡪 [ A|0 B|8 C|3 D|∞]  pq 🡪 [ B|8] |
| C's Neighbor D is processed. Since 3+2 < ∞, remove D and updates its   distanceToVertex to 5. D is placed in the pq. | A🡪 [ A|0 B|8 C|3 D|5]  pq 🡪 [ B|8 D|5] |
| The nearest vertex, D, is automatically removed from the pq. | A🡪 [ A|0 B|8 C|3 D|5]  pq 🡪 [ B|8] |
| D's Neighbor B is processed. Since 5+1 < 8, remove B and update its   distanceToVertex to 6. B is re-placed in the pq. | A🡪 [ A|0 B|6 C|3 D|5]  pq 🡪 [ B|6] |
| The nearest vertex, B, is automatically removed from the pq. | A🡪 [ A|0 B|6 C|3 D|5]  pq 🡪 [] |
| B has no neighbors. The pq is now empty.  End Loop  The wVertex A is now storing in the array of PQelements [ A|0 B|6 C|3 D|5]  The minimum distance from A to A is 0, to B is 6, to C is 3, and to D is 5. |  |

Write the pseudocode for Dijkstra's Algorithm, aka minimumWeightPath.

private void minimumWeightPath(String vName) //Dijkstra's

{

}

The graph object, implementing an interface, is similar to AdjList, updated to use wVertex.

class AdjListWeighted implements AdjListWeightedInterface {  
 //we want our map to be ordered alphabetically by vertex name  
 private Map<String, wVertex> vertexMap = new TreeMap<String, wVertex>();

/\* constructor is not needed! \*/

public Set<wVertex> getVertices()

public Map<String, wVertex> getVertexMap()  
 public wVertex getVertex(String vName)   
 public void addVertex(String vName)

public void addEdge(String source, String target, double d)

public void minimumWeightPath(String vName); //Dijkstra's

public String toString();

**Assignment:** Finish Neighbor, PQelement, wVertex and AdjListWeighted. The driver is Dijkstra\_6\_Driver. Notice that we hard-code all inputs and test each class individually. You will turn in AdjListWeighted.

**Sample Run** (Dijkstra\_6\_Driver.java using AdjListWeighted)

Testing wVertex  
 get the names:  
 alpha  
 beta  
 get the list of Neighbors:   
 [beta 5.0]  
 [alpha 3.0]

Testing Neighbor  
 Neighbor's toString(): alpha 100.0  
 Neighbor's toString(): alpha 101.0  
 Neighbors are equal if the names are the same: 0  
   
 Testing PQelement  
 PQelement's toString(): alpha 10.0  
 PQelement's toString(): beta 11.0  
 Comparing two PQelements returns the difference in distance: -1  
 toString() shows the name, the name and distance to the neighbor(s), and the list of PQelements:   
 alpha [beta 5.0] List: []  
 beta [alpha 3.0] List: []  
   
 Testing wVertex's PQelements  
 Adding gamma and delta to alpha's list of PQelement  
 Get alpha's list of reachables: [gamma 20.0, delta 21.0]  
 Get the PQelement which is wVertex g: gamma 20.0  
 Get the PQelement which is wVertex d: delta 21.0  
   
 Hard-coding vertices and neighbors with weights.  
 Get the vertex by name "A": A [B 9.0, C 3.0] List: []  
 Get the vertices: [A [B 9.0, C 3.0] List: [], B [] List: [], C [B 5.0, D 2.0] List: [], D [B 1.0] List: []]  
 Get the map: {A=A [B 9.0, C 3.0] List: [], B=B [] List: [], C=C [B 5.0, D 2.0] List: [], D=D [B 1.0] List: []}  
 The whole graph:  
 A [B 9.0, C 3.0] List: []  
 B [] List: []  
 C [B 5.0, D 2.0] List: []  
 D [B 1.0] List: []  
   
 Dijkstra's Algorithm!  
 Enter source: A  
   
 After processing, the entire graph is:  
 A [B 9.0, C 3.0] List: [A 0.0, B 6.0, C 3.0, D 5.0]  
 B [] List: []  
 C [B 5.0, D 2.0] List: []  
 D [B 1.0] List: []  
   
 State of the source vertex: A  
 A [B 9.0, C 3.0] List: [A 0.0, B 6.0, C 3.0, D 5.0]  
   
 The source A knows the distance to each target:  
 Distance to A: 0.0  
 Distance to B: 6.0  
 Distance to C: 3.0  
 Distance to D: 5.0