10/25/2011 & 10/27/2011

Single-source shortest path problem - Dijkestra's Algorithm

First: example of a graph with a source node

Then we want the shortest path to all nodes.

Dijkestra's Algorithm works in this way:

In some sense, this algorithm "expands outward" from the **starting point**, iteratively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path.

Steps of the algorithm

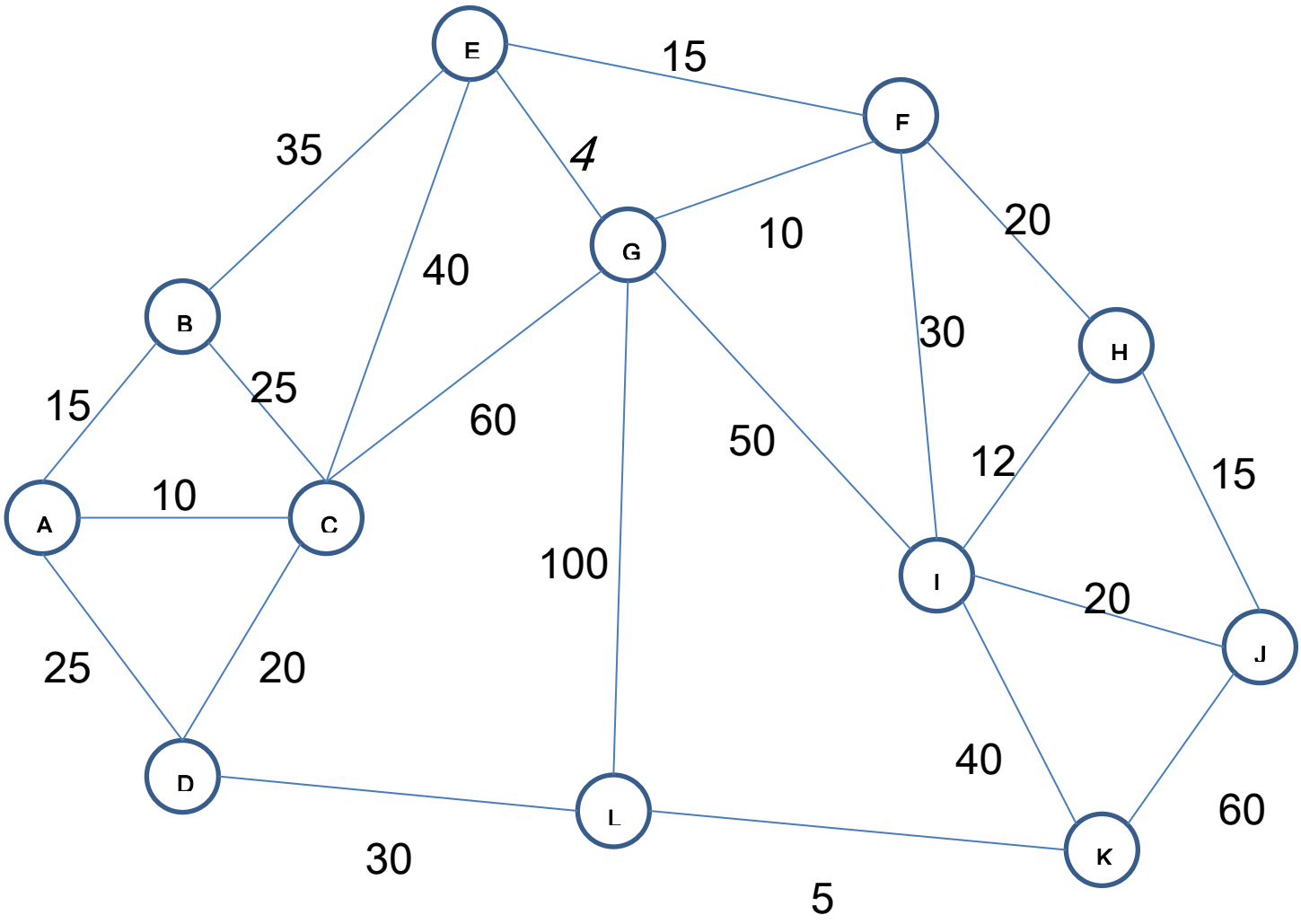
Let the node at which we are starting be called the **initial node**. Let the **distance of node Y** be the distance from the **initial node** to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes except the initial node as unvisited. Set the initial node as current. Create a set of the unvisited nodes called the *unvisited set* consisting of all the nodes except the initial node.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6+2=8. If this distance is less than the previously recorded distance, then overwrite that distance. Even though a neighbor has been examined, it is not marked as *visited* at this time, and it remains in the *unvisited set*.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again; its distance recorded now is final and minimal.
5. The next *current node* will be the node marked with the lowest (tentative) distance in the *unvisited set*.
6. If the *unvisited set* is empty, then stop. The algorithm has finished. Otherwise, set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.

In the following algorithm, the code u := vertex in Q with smallest dist[], searches for the vertex u in the vertex set Q that has the least dist[u] value. That vertex is removed from the set Q and returned to the user. dist_between(u, v) calculates the length between the two neighbor-nodes u and v. The variable alt on line 15 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this alt path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
 1  function Dijkstra(Graph, source):
 2      for each vertex v in Graph:           // Initializations
 3          dist[v] := infinity ;             // Unknown distance function from source to v
 4          previous[v] := undefined ;        // Previous node in optimal path from source
 5      end for ;
 6      dist[source] := 0 ;                   // Distance from source to source
 7      Q := the set of all nodes in Graph ;
       // All nodes in the graph are unoptimized - thus are in Q
 8      while Q is not empty:                 // The main loop
 9          u := vertex in Q with smallest distance in dist[] ;
10          if dist[u] = infinity:
11              break ;                       // all remaining vertices are inaccessible from source
12          end if ;
13          remove u from Q ;
14          for each neighbor v of u:         // where v has not yet been removed from Q.
15              alt := dist[u] + dist_between(u, v) ;
16              if alt < dist[v]:             // Relax (u,v,a)
17                  dist[v] := alt ;
18                  previous[v] := u ;
19                  decrease-key v in Q;      // Reorder v in the Queue
20              end if ;
21          end for ;
22      end while ;
23      return dist[] ;
24  end Dijkstra.
```

Apply to example



Vertex	Distance	Predecessor	Is visited
A	Infinity	Null	No
B	Infinity	Null	No
C	Infinity	Null	No
D	Infinity	Null	No
E	Infinity	Null	No
F	Infinity	Null	No
G	Infinity	Null	No
H	Infinity	Null	No
I	Infinity	Null	No
J	Infinity	Null	No
K	Infinity	Null	No
L	Infinity	Null	No

Source node: L
Initialize distance[L] = 0

Vertex	Distance	Predecessor	Is visited
A	Infinity	Null	No
B	Infinity	Null	No
C	Infinity	Null	No
D	Infinity	Null	No
E	Infinity	Null	No
F	Infinity	Null	No
G	Infinity	Null	No
H	Infinity	Null	No
I	Infinity	Null	No
J	Infinity	Null	No
K	Infinity	Null	No
L	0	Null	No

1) Take node of least distance: L
   Update distance of vertices connected to L

Vertex	Distance	Predecessor	Is visited
A	Infinity	Null	No
B	Infinity	Null	No
C	Infinity	Null	No
D	Dist[L] + 30 = 30	L	No
E	Infinity	Null	No
F	Infinity	Null	No
G	Dist[L] + 100 = 100	L	No
H	Infinity	Null	No
I	Infinity	Null	No
J	Infinity	Null	No
K	Dist[L] + 5 = 5	L	No
L	0	Null	**Yes**

2) Take node of least distance: K
   Update distance of vertices connected to K

Vertex	Distance	Predecessor	Is visited
A	Infinity	Null	No
B	Infinity	Null	No
C	Infinity	Null	No
D	30	L	No
E	Infinity	Null	No
F	Infinity	Null	No
G	100	L	No

H	Infinity	Null	No
I	Dist[K] + 40 = 45	K	No
J	Dist[K] + 60 = 65	K	No
K	5	L	**Yes**
L	0	Null	**Yes**

3) Take node of least distance: D
   Update distance of vertices connected to D

Vertex	**Distance**	**Predecessor**	**Is visited**
A	Dist[D] + 25 = 55	D	No
B	Infinity	Null	No
C	Dist[D] + 20 = 50	D	No
D	30	L	**Yes**
E	Infinity	Null	No
F	Infinity	Null	No
G	100	L	No
H	Infinity	Null	No
I	45	K	No
J	65	K	No
K	5	L	**Yes**
L	0	Null	**Yes**

4) Take node of least distance: I
   Update distance of vertices connected to I

Vertex	**Distance**	**Predecessor**	**Is visited**
A	55	D	No
B	Infinity	Null	No
C	50	D	No
D	30	L	**Yes**
E	Infinity	Null	No
F	Dist[I] + 30 = 75	I	No
G	Dist[I] + 50 = 95 < Dist[j] 100, so update	I	No
H	Dist[I] + 12 = 57	I	No
I	45	K	**Yes**
J	Dist[I] + 20 = 65 = same Dist[J] so need to change	K	No
K	5	L	**Yes**
L	0	Null	**Yes**

5) Take node of least distance: C
Update distance of vertices connected to C

Vertex	Distance	Predecessor	Is visited
A	Dist[C] + 10 = 60 > 55, so no change	D	No
B	Dist[C] + 25 = 75	C	No
C	50	D	**Yes**
D	30	L	**Yes**
E	Dist[C] + 40 = 90	C	No
F	75	I	No
G	Dist[C] + 60 = 110 > 95, so no change	I	No
H	57	I	No
I	45	K	**Yes**
J	65	K	No
K	5	L	**Yes**
L	0	Null	**Yes**

6) Take node of least distance: A
Update distance of vertices connected to A

Vertex	Distance	Predecessor	Is visited
A	55	D	**Yes**
B	Dist[A] + 15 = 70 < 75, so update	A	No
C	50	D	**Yes**
D	30	L	**Yes**
E	90	C	No
F	75	I	No
G	95	I	No
H	57	I	No
I	45	K	**Yes**
J	65	K	No
K	5	L	**Yes**
L	0	Null	**Yes**

7) Take node of least distance: H
Update distance of vertices connected to H

Vertex	Distance	Predecessor	Is visited
A	55	D	**Yes**
B	70	A	No
C	50	D	**Yes**

D	30	L	**Yes**
E	90	C	No
F	Dist[H] + 30 = 87 > 75 so no change	I	No
G	95	I	No
H	57	I	**Yes**
I	45	K	**Yes**
J	Dist[H] + 15 = 72 > 65 so no change	K	No
K	5	L	**Yes**
L	0	Null	**Yes**

8) Take node of least distance: J
   Update distance of vertices connected to J

Vertex	**Distance**	**Predecessor**	**Is visited**
A	55	D	**Yes**
B	70	A	No
C	50	D	**Yes**
D	30	L	**Yes**
E	90	C	No
F	75	I	No
G	95	I	No
H	57	I	**Yes**
I	45	K	**Yes**
J	65	K	**Yes**
K	5	L	**Yes**
L	0	Null	**Yes**

9) Take node of least distance: B
   Update distance of vertices connected to B

Vertex	**Distance**	**Predecessor**	**Is visited**
A	55	D	**Yes**
B	70	A	**Yes**
C	50	D	**Yes**
D	30	L	**Yes**
E	Dist[B] + 35 = 105 > 90 so no change	C	No
F	75	I	No
G	95	I	No
H	57	I	**Yes**
I	45	K	**Yes**
J	65	K	**Yes**

K	5	L	**Yes**
L	0	Null	**Yes**

10) Take node of least distance: F
    Update distance of vertices connected to F

Vertex	**Distance**	**Predecessor**	**Is visited**
A	55	D	**Yes**
B	70	A	**Yes**
C	50	D	**Yes**
D	30	L	**Yes**
E	Dist[F] + 15 = 90 = same as Dist[E] so no need to update	C	No
F	75	I	**Yes**
G	Dist[F] + 10 = 85 < 95 so update	F	No
H	57	I	**Yes**
I	45	K	**Yes**
J	65	K	**Yes**
K	5	L	**Yes**
L	0	Null	**Yes**

11) Take node of least distance: G
    Update distance of vertices connected to G

Vertex	**Distance**	**Predecessor**	**Is visited**
A	55	D	**Yes**
B	70	A	**Yes**
C	50	D	**Yes**
D	30	L	**Yes**
E	Dist[G] + 4 = 89 < 90 so no update	G	No
F	75	I	**Yes**
G	85	F	**Yes**
H	57	I	**Yes**
I	45	K	**Yes**
J	65	K	**Yes**
K	5	L	**Yes**
L	0	Null	**Yes**

12) Take node of least distance: E
    Update distance of vertices connected to E

Vertex	Distance	Predecessor	Is visited
A	55	D	**Yes**
B	70	A	**Yes**
C	50	D	**Yes**
D	30	L	**Yes**
E	89	G	**Yes**
F	75	I	**Yes**
G	85	F	**Yes**
H	57	I	**Yes**
I	45	K	**Yes**
J	65	K	**Yes**
K	5	L	**Yes**
L	0	Null	**Yes**

Now we have the shortest distances and their predecessors.

So for example what is the distance to B?
It is 70 and the path is?
Follow the path backwards from B:

B -> A -> D -> L

Minimum spanning tree - Prim's Algorithm

First: example of a graph

Second: Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Algorithm steps:

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

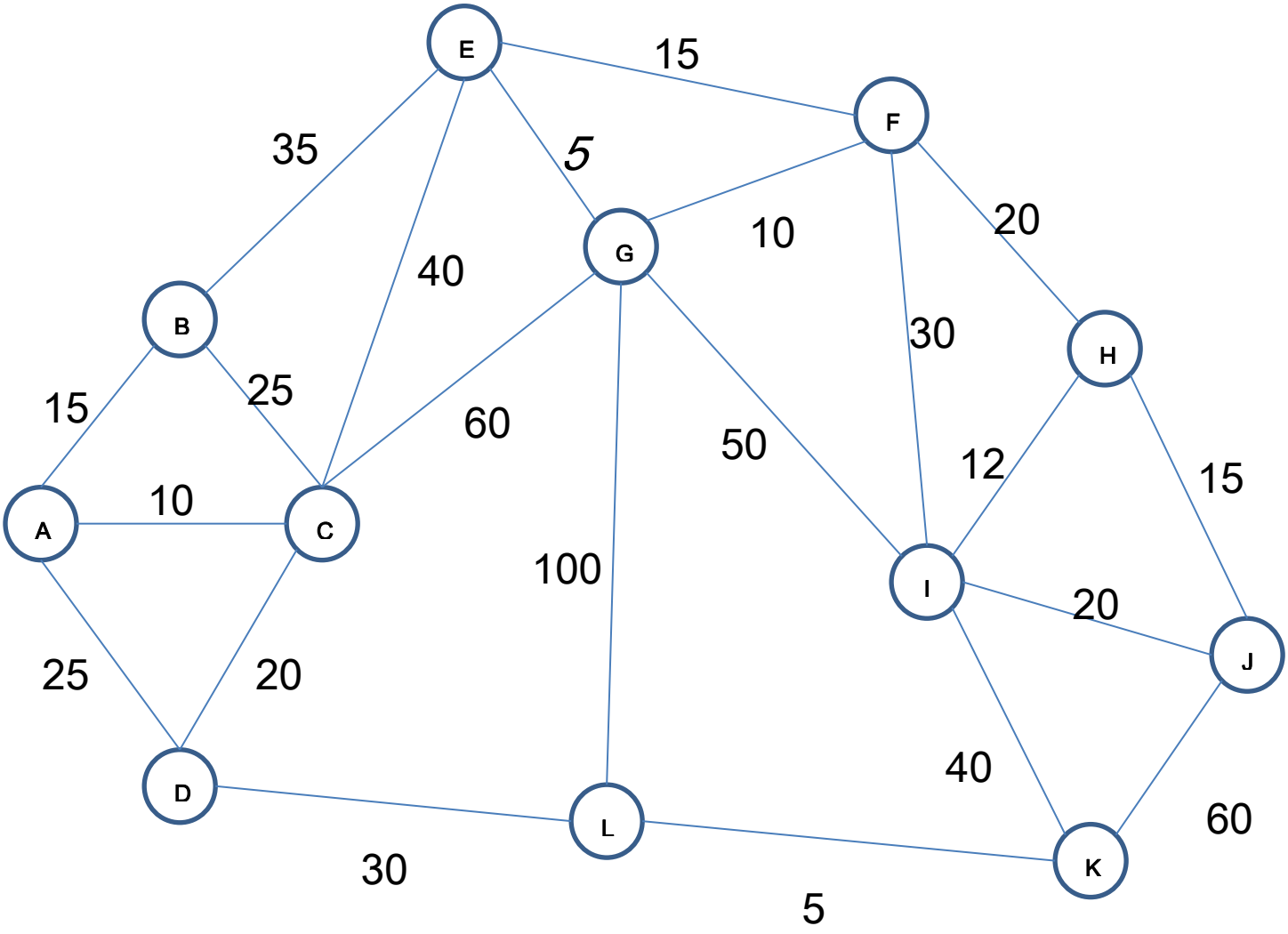
Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
Initialize: Vnew = {x}, where x is an arbitrary node (starting point) from V, Enew = {}
Repeat until Vnew = V:
Choose an edge (u, v) with minimal weight such that u is in Vnew and v is not (if there are multiple edges with the same weight, any of them may be picked)
Add v to Vnew, and (u, v) to Enew
Output: Vnew and Enew describe a minimal spanning tree

Pseudo code

```
Prim(G=(V, E), w, r)
{
        // initialization
        For each vertex u in V
        {
                Priority[u] = infinity
                Parent[u] = null
        }

        Priority[r] = 0
        Q = V    // Q: set of remaining vertices to add to the minimum spanning tree
        While Q is not empty
        {
                Let u = vertex with minimum priority in Q
                For each vertex v in Q and w(u, v) < Priority[v]
                {
                        Priority[v] = w(u, v)
                        Parent[v] = u
                }
        }
}
```

Apply to example



Vertex	Priority of vertex	Parent of vertex	Is in Q?
A	Infinity	Null	Yes
B	Infinity	Null	Yes
C	Infinity	Null	Yes
D	Infinity	Null	Yes
E	Infinity	Null	Yes
F	Infinity	Null	Yes
G	Infinity	Null	Yes
H	Infinity	Null	Yes
I	Infinity	Null	Yes
J	Infinity	Null	Yes
K	Infinity	Null	Yes
L	0	Null	Yes

1) L has the least priority

Add it to the minimum spanning tree

Update priorities of vertices connected to L

Vertex	Priority of vertex	Parent of vertex	Is in Q?
A	Infinity	Null	Yes
B	Infinity	Null	Yes
C	Infinity	Null	Yes
D	30	L	Yes
E	Infinity	Null	Yes
F	Infinity	Null	Yes
G	100	L	Yes
H	Infinity	Null	Yes
I	Infinity	Null	Yes
J	Infinity	Null	Yes
K	5	L	Yes
	0	Null	*No*

2) K has the least priority

Add it to the minimum spanning tree through edge (L, K)

Update priorities of vertices connected to K

Tree edges: (L, K)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
A	Infinity	Null	Yes
B	Infinity	Null	Yes
C	Infinity	Null	Yes
D	30	L	Yes
E	Infinity	Null	Yes
F	Infinity	Null	Yes
G	100	L	Yes
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	*No*
	0	Null	No

3) D has the least priority

Add it to the minimum spanning tree through edge (D, L)

Update priorities of vertices connected to D

Tree edges: (L, K), (D, L)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
A	25	D	Yes

B	Infinity	Null	Yes
C	20	D	Yes
	30	L	No
E	Infinity	Null	Yes
F	Infinity	Null	Yes
G	100	L	Yes
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

4) C has the least priority
   Add it to the minimum spanning tree through edge (C, D)
   Update priorities of vertices connected to C

   Tree edges: (L, K), (D, L), (C, D)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
A	10	C	Yes
B	25	C	Yes
	20	D	No
	30	L	*No*
E	40	C	Yes
F	Infinity	Null	Yes
G	60	C	Yes
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

Priority of G changed for the second time
Priority of A changed for the second time

5) A has the least priority
   Add it to the minimum spanning tree through edge (A, C)
   Update priorities of vertices connected to A

   Tree edges: (L, K), (D, L), (C, D), (A, C)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	*No*
B	15	A	Yes

	20	D	No
	30	L	No
E	40	C	Yes
F	Infinity	Null	Yes
G	60	C	Yes
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

Priority of B changed for the second time

6) B has the least priority
   Add it to the minimum spanning tree through edge (A, B)
   Update priorities of vertices connected to B

   Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	No
	15	A	*No*
	20	D	No
	30	L	No
E	35	B	Yes
F	Infinity	Null	Yes
G	60	C	Yes
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

Priority of E changed for the second time

7) E has the least priority
   Add it to the minimum spanning tree through edge (E, B)
   Update priorities of vertices connected to E

   Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B), (E, B)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	No
	15	A	No
	20	D	No
	30	L	No
	35	B	*No*

F	15	E	Yes
G	5	E	Yes
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

Priority of G changed for the third time

8) G has the least priority
   Add it to the minimum spanning tree through edge (E, G)
   Update priorities of vertices connected to G

   Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B), (E, B), (E, G)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	No
	15	A	No
	20	D	No
	30	L	No
	35	B	No
F	10	G	Yes
	5	E	*No*
H	Infinity	Null	Yes
I	40	K	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

Priority of F changed for the second time

9) F has the least priority
   Add it to the minimum spanning tree through edge (G, F)
   Update priorities of vertices connected to F

   Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B), (E, B), (E, G), (G, F)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	No
	15	A	No
	20	D	No
	30	L	No
	35	B	No
	10	G	*No*

	5	E	No
H	20	F	Yes
I	30	F	Yes
J	60	K	Yes
	5	L	No
	0	Null	No

Priority of I changed for the second time

10) H has the least priority
    Add it to the minimum spanning tree through edge (F, H)
    Update priorities of vertices connected to H

    Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B), (E, B), (E, G), (G, F), (F, H)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	No
	15	A	No
	20	D	No
	30	L	No
	35	B	No
	10	G	No
	5	E	No
	20	F	*No*
I	12	H	Yes
J	15	H	Yes
	5	L	No
	0	Null	No

Priority of I changed for the third time
Priority of J changed for the second time

11) I has the least priority
    Add it to the minimum spanning tree through edge (H, I)
    Update priorities of vertices connected to I: no change to J's priority

    Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B), (E, B), (E, G), (G, F), (F, H), (H, I)

Vertex	Priority of vertex	Parent of vertex	Is in Q?
	10	C	No
	15	A	No
	20	D	No
	30	L	No
	35	B	No
	10	G	No
	5	E	No

	20	F	No
	12	H	*No*
J	15	H	Yes
	5	L	No
	0	Null	No

12 J has the least priority
Add it to the minimum spanning tree through edge (H, J)

Tree edges: (L, K), (D, L), (C, D), (A, C), (A, B), (E, B), (E, G), (G, F), (F, H), (H, I), (H, J)

So we see that a node's priority can change several times while we are computing the minimum spanning tree.