Weighted Graphs

Overview

The previous section described both directed and non-directed graphs; there the only measure is whether a connection exists between any two particular nodes. Here, weighted graphs are being discussed; for them edges have a value, not just a binary exist or not exist. As with non-weighted graphs, weighted graphs are comprised of nodes and edges, can be directed or non-directed, connected or non-connected.

Having weights allows for calculations that cannot be performed on non-weighted graphs, such as the shortest distance between two points. The shortest distance between two nodes is referred to as the shortest path connecting them. When driving, it is important to understand the shortest route to the destination, it is not normally a good idea to go to New York from Oregon via Arizona. In other situations, it may be advantageous to know the shortest path length from some starting node to all other nodes in the graph. This combination of shortest paths is often presented as a list of distances or a min-path modification of the adjacency matrix. The first is usually derived using Dijkstra's algorithm and the second by using the Floyd-Warshaw algorithm. It is also possible to create a minimum path tree from Dijkstra's algorithm as shown below in the examples.

In the previous section on non-weighted graphs, one result was determining a minimum spanning tree or a minimal list of edges that maintains the same graph connectivity. While it is possible to use depth first and bread first traversals to find a minimum spanning tree for a weighted graph, it is not normally of interest. Instead, a common task is to determine the minimum cost spanning tree starting from a given node, where the cost of the edge weights is considered in addition to the connections. A real-world example is determining how to lay fiber optic cables between cities, where the goal is connecting all of the cities with the lowest total cost.

Representation

Nodes are represented in the same way as in non-weighted graphs.

Like with non-weighted graphs, edges can be represented either by an adjacency list or by an adjacency matrix. The edge class for an adjacency list contains more information than in an unweighted graph as shown below:

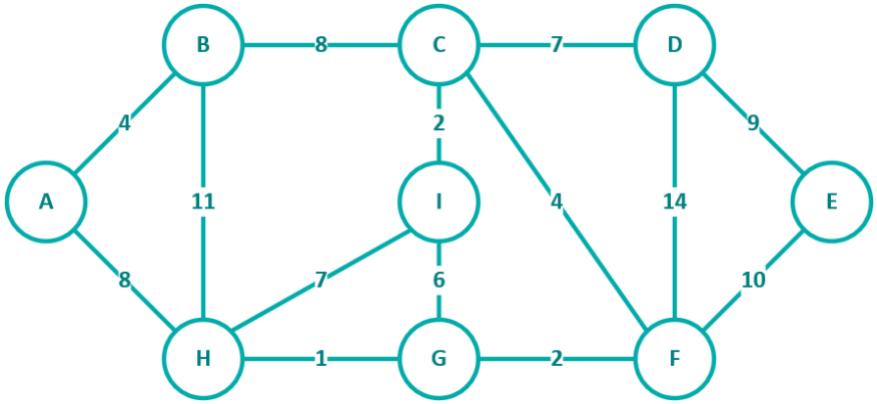
```
class Edge
   int startIndex   // the index of the starting node of this edge
   int endIndex     // the index of the ending node of this edge

   int weight      // the weight of this edge

   Edge next       // the next value, used for the edgeList
```

Adding both the startIndex and endIndex is to make the minimum cost algorithm simpler to code. As before, the variable next is used to maintain a linked list of edges.

The example code provided before for adding an edge to a graph can be simply modified to support adding the additional information to each edge.
For a non-directed graph, add two edges, one starting at each end and the two appropriate entries in the adjacency matrix.
For a directed graph, add a single edge and a single entry in the adjacency matric.



The adjacency matrix for the above weighted graph is

	A	B	C	D	E	F	G	H	I
A	-	4						8	
B	4	-	8					11	
C		8	-	7		4			2
D			7	-	9	14			
E				9	-	10			
F			4	14	10	-	2		
G						2	-	1	6
H	8	11					1	-	7
I			2				6	7	-

Again, for a non-directed graph the adjacency matrix is symmetrical along the diagonal.

Algorithms

It is possible to have a depth first or a breadth first traversal of a weighted graph and come up with a minimum spanning tree. But, since there are weights on edges, it is more common to calculate two other measures. The first is the minimum distance between two nodes and the second is the minimum cost spanning tree.

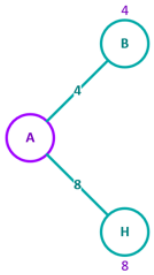
Minimum Path Length

Dijkstra's algorithm is used to calculate the distance from any starting node to all other nodes that can be reached. The algorithm is as follows:

1) Create a set *sptSet* (shortest path tree set) that keeps track of nodes included in the shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all nodes in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the beginning vertex so that it is picked first. Store these values in a second set.
3) While *sptSet* doesn't include all nodes
    a. Pick a node u which is not yet in *sptSet* and has minimum distance value.
    b. Add u to *sptSet*.
    c. Update distance value of all adjacent nodes of u. To update a distance values, iterate through all adjacent nodes. For every adjacent node v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

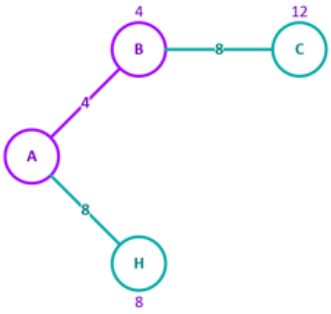
Rather than using an infinite distance, the algorithm uses the largest integer. The assumption is that no edge will have a weight this large.

Using the above weighted graph, here are the first few steps using the node A as a starting node. Add A to the sptSet and set its distance as zero

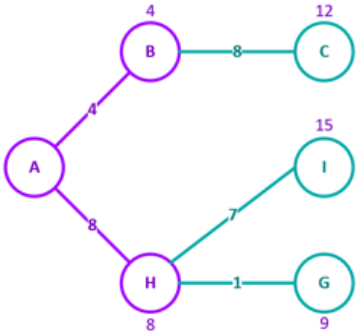


.

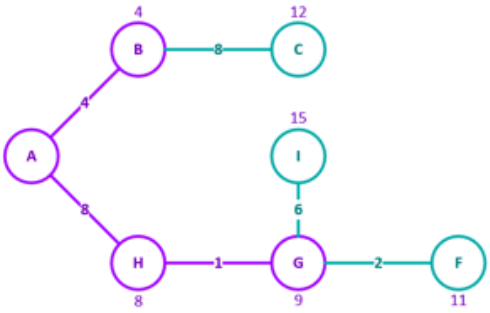
Now the distances to B and H are updated to be 4 and 8 respectively. Since B has the shortest distance, it is added to sptSet as shown in the next image.



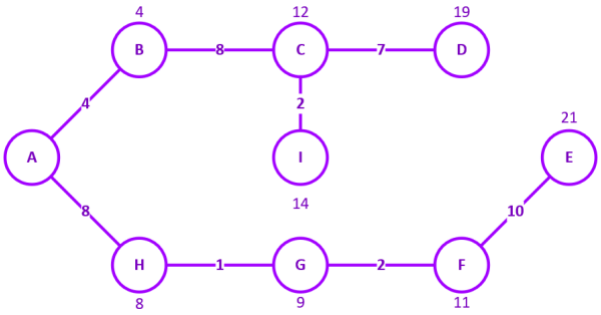
The distance to C is updated to be 12 (4 + 8). The node with the shortest distance is thus H, so it is added to sptSet.

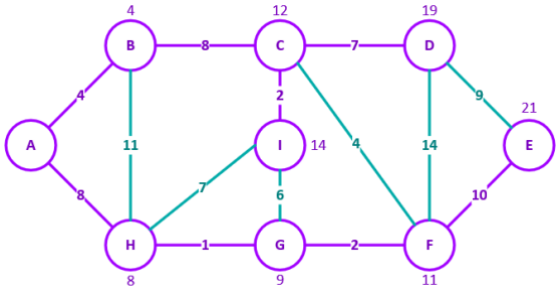


The distances to G and I are now updated to be 9 (8+1) and 15 (8+7) respectively. The node with the shortest distance is G, so it is added to sptSet.



Once G has been added to sptSet, the incremental distance to I becomes the 6 from G rather than the previous 7 from H so the edge from H to I is shown as being pruned. As a result, the distance to I is updated to be 15 (9+6) and to F to be 11 (9 + 2). Now the node with the shortest distance is F, so it is added to sptSet. After this algorithm runs to completion, the set of shortest paths is as shown below along with an image showing in teal what edges were not used in the calculation.





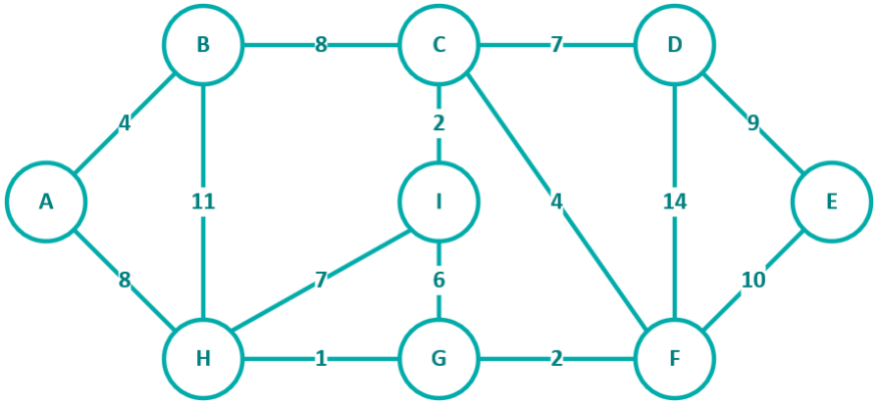
Although the algorithm described above simply creates a new set of the distances from the starting node, it is possible to add a step that keeps track of what node is immediately previous to the one just added to the sptSet. This allows for the generation of a set of minimum paths by working backward from a given node to its predecessor and then its predecessor and so on until the starting node is reached.

All Pairs Minimum Path

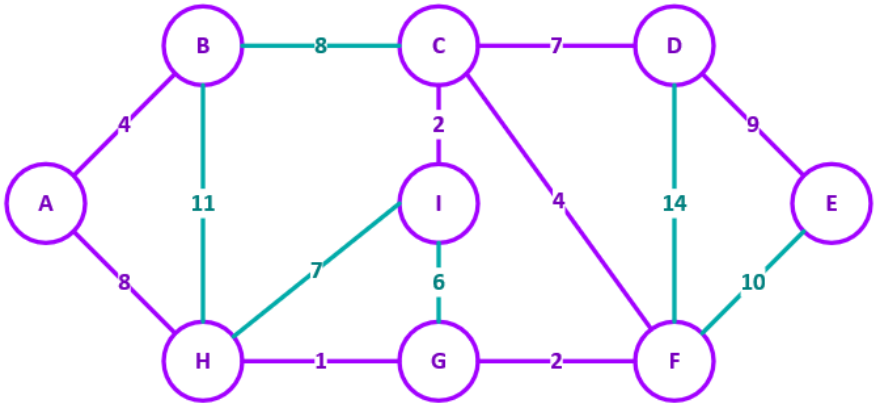
The Floyd-Warshaw algorithm can be used to modify an adjacency matrix to show the minimum distance from any node to any other node. This is similar to the algorithm discussed under non-weighted graphs to create a connectivity matrix from an adjacency matrix. It will not be covered in more detail in this document as it is readily available on the web and is similar to what was already covered.

Minimum Cost Spanning Tree

As stated before, a common problem is how to connect some set of nodes with the minimum cost. This might be in laying out a circuit board or microprocessor chip or it might be in designing a fiber optic system to connect multiple cities and states.



Using the algorithm below, the minimum cost spanning tree for this graph is:



Notice that the minimum cost spanning tree starting at A is different from the shortest path tree shown before. Specifically, the path from A to C goes through H, G, and F. Although this seems counter-intuitive, adding up the total weights of the edges shows that the minimum cost spanning tree has indeed a lower total weight.

Previously minimum spanning trees were created using a either a breadth first traversal with a FIFO or a depth first traversal with a LIFO of nodes. This was possible because all paths are equal in weight. Because the weight of each edge is a significant factor, the algorithm for minimum cost spanning tree requires a modified min priority queue of edges.

As previously discussed, priority queues can be implemented on an array by adding at the next location and then searching for the minimum value, or by inserting into an ordered array and removing the smallest value, or most efficiently by using a heap. None of these approaches works for the min priority queue required here.

Since it is possible to add a second edge terminating in the same node (like HI vs GI in the graph above), it is necessary to modify the method used to add a new value.

1. First, see if there is already an edge in the PQ ending at the same node as the edge to be added.
2. If such an edge is found, compare the weight of the two edges.
3. If the new edge has a smaller weight, replace the edge currently in the PQ with the new edge, otherwise do nothing.
4. If there is no other edge in the PQ with that ending node, then add the new edge in the next location.

It is not possible to implement such a priority queue efficiently using a heap or an ordered array. It must be implemented on an unordered array where the removeSmallest method requires a linear search.

Using this type of priority queue, the algorithm is basically the same as a breadth first traversal, except in a breadth first traversal the output is the name of the next node encountered and here the output is the edge that was returned from the priority queue showing both its beginning and end.

The algorithm in pseudo code can be stated simply as:

```
mark starting node as visited and add all its edges to PQueue

while PQueue not empty
   find and remove shortest edge from PQueue
   save ending node of that edge as current
   mark current as visited
   output this edge as part of min-cost tree


  foreach edge starting at current
     if end not visited
        addQueue(edge)

reset all nodes to not visited
```