Shortest Weighted Path With Nonnegative Weights

Dijkstra's Algorithm

See Section 14.6.2 of the text.

Last week we found the shortest path from a specific source node to every other node in the graph, measuring the length of a path by the number of its edges.

The algorithm was: give each node a value field and a predecessor field. Give the source node a value 0 and all other nodes value INFINITY. Initially make all nodes have null for their predecessors.

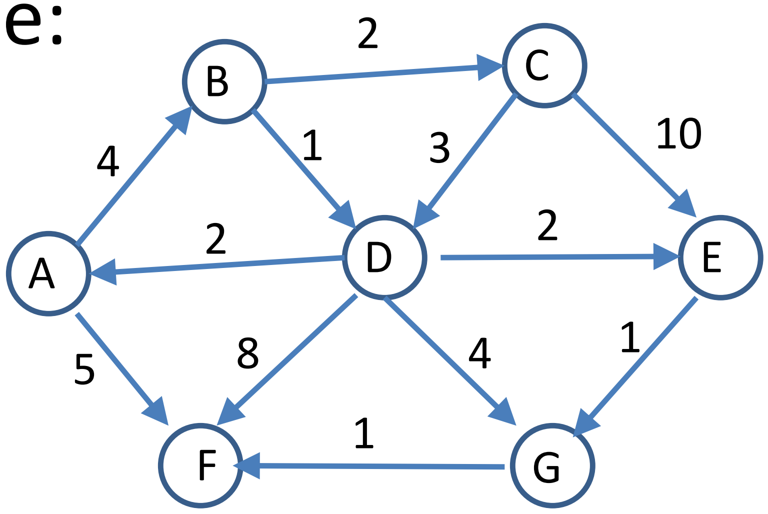
Maintain a queue of nodes. Initially the source node is added to the queue. Perform the following steps until the queue is empty:

a) Remove the head of the queue. Call this node X.
b) For each outgoing edge from X to another node Y, if the value of Y is INFINITY, make the new value of Y be the value of X + 1, make the predecessor of Y be X, and add Y to the queue.

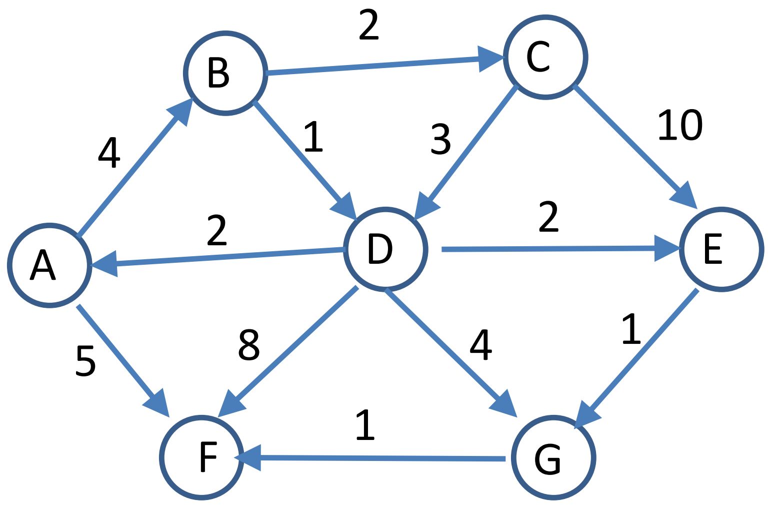
Dijkstra's algorithm covers the extension of this to graphs with non-negative edge weights.
For this algorithm we will assume that all weights are non-negative. We will see another, less efficient, algorithm that will handle the case of negative weights.

We measure the "length" of a path by the sum of the weights on its edges. We want to find the shortest (cheapest) path from a source node S to every other node in the graph.

For example:



Clicker Q: What is the cost (or length or weight) of the cheapest path from A to G?



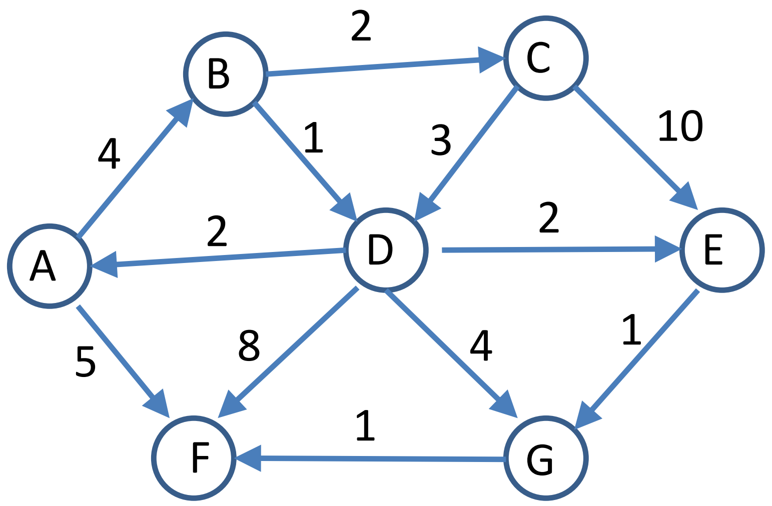
A. 6
B. 8
C. 9
D. 17

What made the unweighted case work was that we first put into the queue the only node whose value was 0, then all of the nodes whose value was 1, then all whose value was 2 and so forth. We know our algorithm works because if there was a shorter path to it we would have put it into the queue sooner.

We can treat this algorithm similarly. We first get the only node whose path has sum 0 (i.e, the source node). We then find the next minimum-cost node, then the next and so forth. We will use a priority queue to store our "working set" of nodes. Each time we finish with a node we put its children into the priority queue.

Here is why we can't have negative edge costs. When we remove an item from our priority queue, we need to be sure that we have its cheapest path. If we allowed negative edge weights, there might be a roundabout path we haven't explored yet that has a large negative weight, giving a path to the node that is cheaper that what we have found so far. In that case our algorithm wouldn't work. So for now we'll stick to non-negative weights.

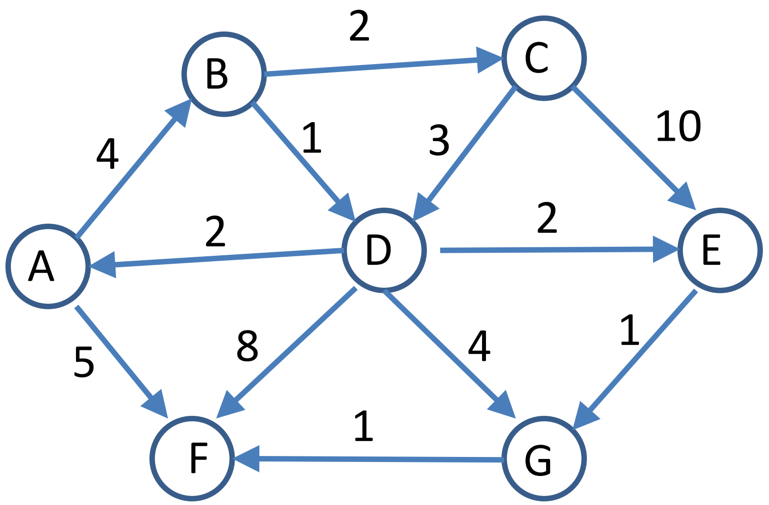
For example, suppose we have this graph and our source node is A:



Initially our priority queue has only A with value 0.

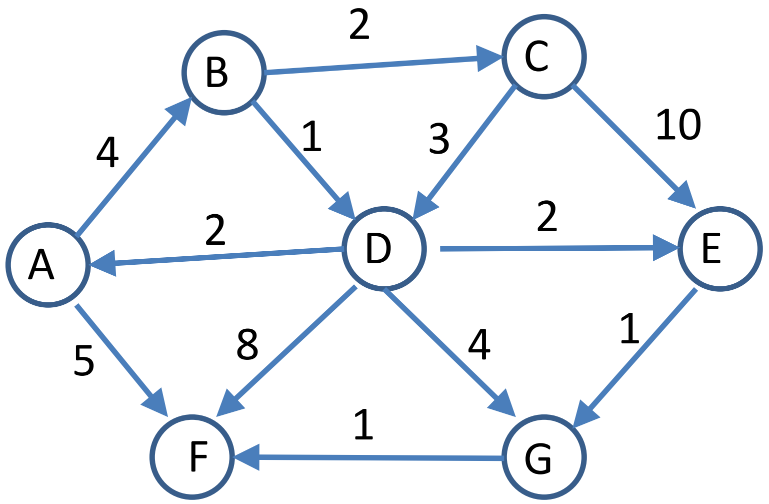
Queue: {(A 0)}

We remove A from the queue, output it, and add the nodes at the end of A's outgoing edges to the queue



Queue: {(B 4) (F 5)}
Output: [(A 0)]

We remove the head of the queue, B, then add its children with their weights added to B's

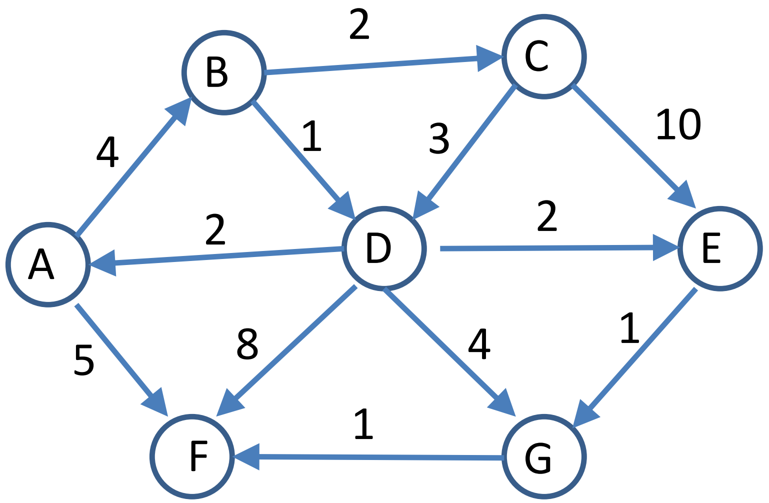


Queue: {(F 5) (D 5) (C 6)}
Output: [(A 0) (B 4)]

The next smallest path is to F, which has no children:

Queue: {(D 5) (C 6)}
Output: [(A 0) (B 4) (F 5)]

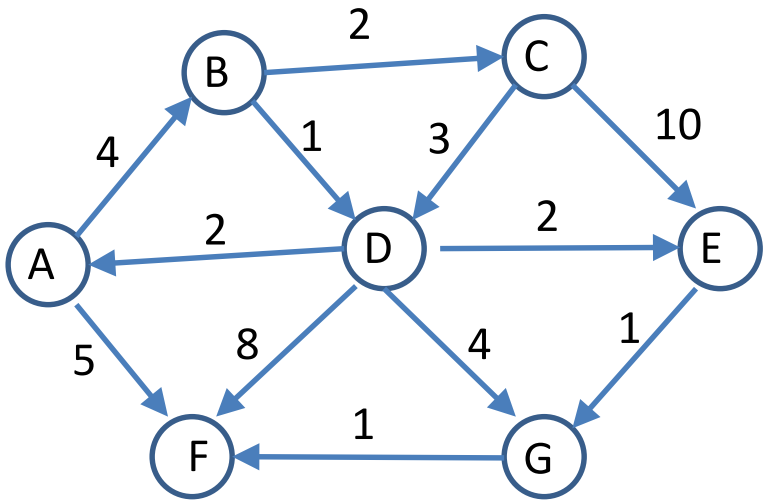
D has edges to A and F, which have been finished, and to E and G



Queue: {(C 6) (E 7) (G 9)}
Output: [(A 0) (B 4) (F 5) (D 5)]

Next we have

Queue: {(E 7) (G 9)}
Output: [(A 0) (B 4) (F 5) (D 5) (C 6)]



Here is a restatement of where we are:

Queue: {(E 7) (G 9)}

Output: [(A 0) (B 4) (F 5) (D 5) (C 6)]

Note that the edge from E to G gives a cheaper path than what we found before, so we add this edge to the queue

Queue: {(G 8) (G 9)}

Output: [(A 0) (B 4) (F 5) (D 5) (C 6) (E 7)]

One more step finishes the process:

Queue: {(G 9)}
Output: [(A 0) (B 4) (F 5) (D 5) (C 6) (E 7) (G 8)]

We have multiple copies of a node in the priority queue with different values because our queue doesn't automatically restructure when one of its values changes. Once the node has been removed from the queue we skip over any remaining copies of it when we remove the head of the queue.

Clicker Question: How would we estimate the running time of this algorithm? We consider every edge and there are |E| edges. Each time we look at an edge we potentially insert another node into the priority queue. Remember that a node can be in the queue more than once. So where does this come out to?

A. O(log(|E|))
B. o(|E|)
C. O(|E|*log(|E|))
D. O($|E|^2$)

How long does this algorithm take? We might add a node to the priority queue for each edge of the graph, so the size of the queue might be |E|. Each removal of the head of the queue takes log(|E|). Altogether, the running time is
O(|E| log(|E|). This is slightly worse than our estimate for unweighted graphs.