Shortest Weighted Path With Nonnegative Weights

Dijkstra's Algorithm

See Section 14.6.2 of the text.

In the last class 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 and a predecessor node. Give the source node a value 0 and all other nodes value INFINITY. Initially make all nodes have null for their predecessors.

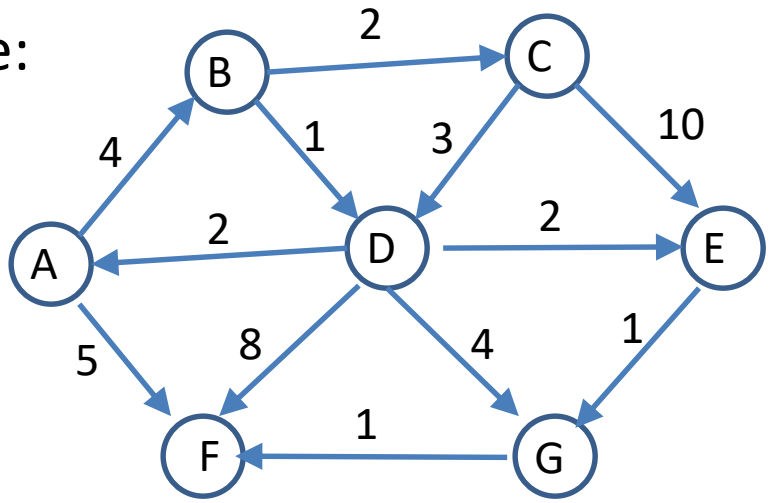
Maintain a queue of nodes. Initially the source node is added to the queue. Perform the following steps until the queue is empty:

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. The next algorithm will be less efficient, but it 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:



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 a node 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.

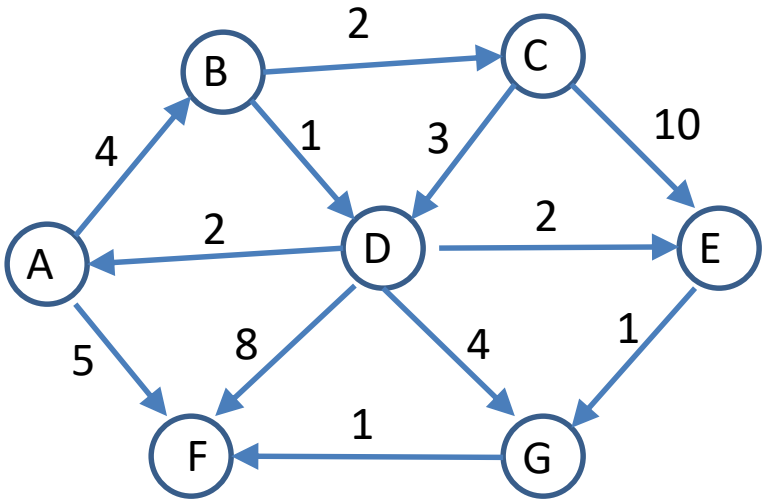
To be clear, we know the best path to a node when it comes OUT of the priority queue, not when it goes in. In fact, we add nodes to the queue when they are the endpoints of outgoing edges of nodes we have taken from the queue. We could actually add a node to the priority queue more than once. When we add a node to the queue we are saying: if we took the path through this predecessor it will cost such-and-such.

Naturally, we want the cheapest path of all of those that are currently in the queue, but we also need to be sure that we have found the cheapest path in the entire graph to the node.

How do we know when we remove a node from the priority queue that we have the shortest path to it. The priority queue guarantees that the path to this node is the shortest of all of the paths in the queue. They would have to expand on one of the paths that is already in the queue and the one we are removing is the cheapest of those. Any alternative path would have to start with something at least as expensive as the one we have chosen, and then add onto it. The result can't possibly be cheaper than the path our algorithm finds.

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.

Here is an example of how our algorithm works. Suppose we have this graph and our source node is A:

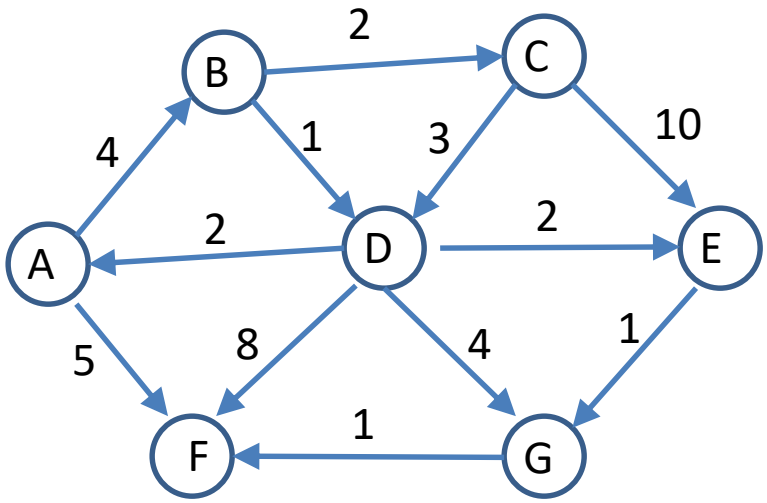


We will use triple (Y,X,C) ito meam “node Y has predecessor X and cost C)”.

Initially our priority queue has only A with value 0.

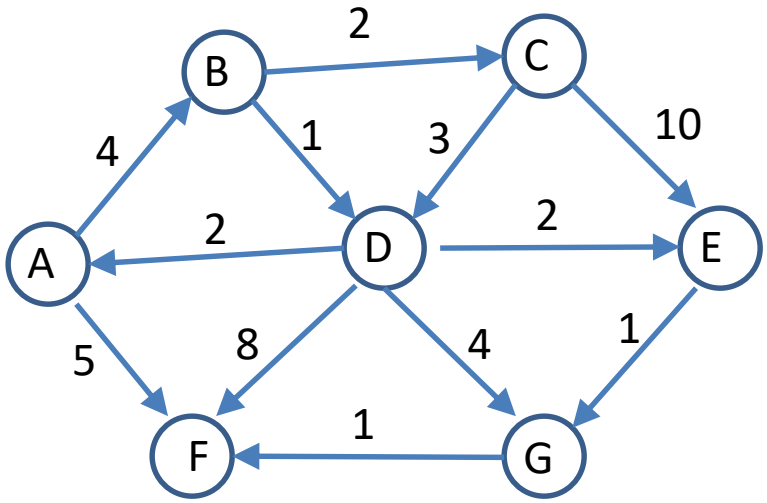
Queue: {(A null, 0)}

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,A,4) (F,A,5)}
Output: [(A,null.0)]

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

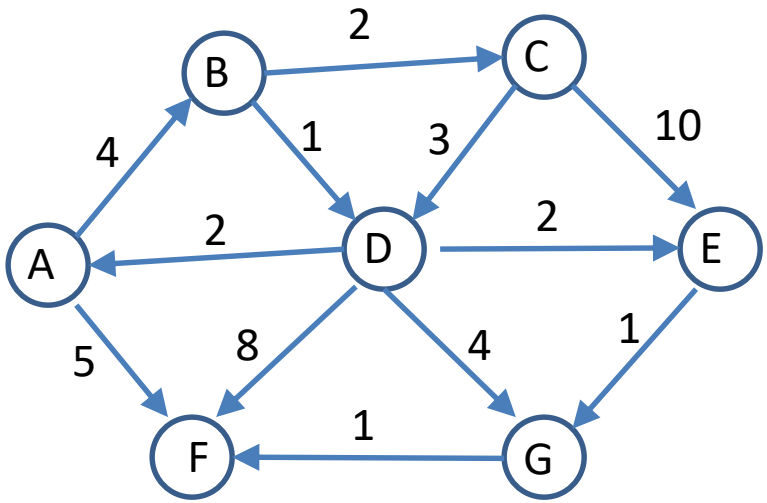


Queue: {(F,A5,) (D,B,5) (C,B,6)}
Output: [(A,null,0) (B,A,4)]

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

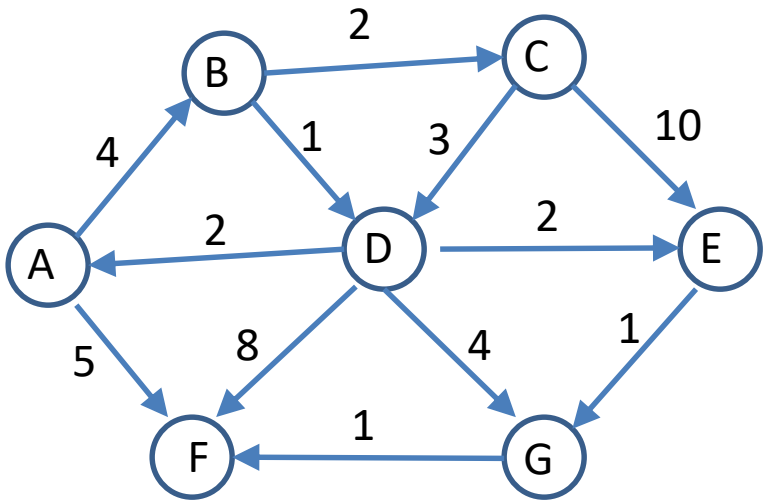
Queue: {(D,B,5) (C,B,6)}
Output: [(A,null,0) (B,A,4) (F,A,5)]

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



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

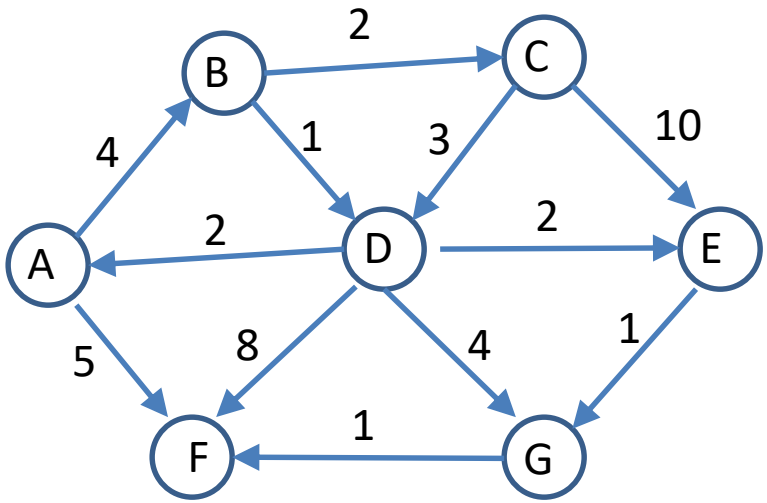
Next we have
Queue: {(E,D,7) (G,D,9) (E,C,16)}
Output: [(A,null.0) (B,A,4) (F,A,5) (D,B,5) (C,B,6)]



Here is a restatement of where we are:
Queue: {(E,D,7) (G,D,9) (E,C,16)}
Output: [(A,null,0) (B,A,4) (F,A,5) (D,B,5) (C,B,6)]

E is in the priority queue twice; we take out the cheaper instance: (E, D, 7). This gives us a cheaper path to G:

Queue: {(G,E,8) (G,F,9) (E,C,16)}
Output: [(A,null,0) (B,A,4) (F,A,5) (D,B,5) (C,B,6) (E,D,7)]



Queue: {(G,E,8) (G,F,9) (E,C,16)}
Output: [(A,null,0) (B,A,4) (F,A,5) (D,B,5) (C,B,6) (E,D,7)]

One more step finishes the process

Queue: {(G,D,9) (E,C,16)}
Output: [(A,null,0) (B,A,4) (F,A,5) (D,B,5) (C,B,6), (E,D,7) (G,E,8)

In the “Output” we have assigned final weights and predecessors to every node as we removed it from the priority queue. We still remove the last entries for G and E from the queue, but since they are already in the output we ignore them.

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.

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 for unweighted graphs.