Shortest Paths in Acyclic Graphs

So far we have been trying to be very general with our graph algorithms, proceeding from graphs with no weights to graphs with only non-negative weights to graphs with any edge weights, positive or negative. The algorithms have become increasingly expensive as they became more general.

Here we back away from generality and give an algorithm that doesn't work for all graphs, but where it does work is efficient and simple to implement.

Suppose our graph has no cycles. This means it has a topological sort -- an ordering of the nodes consistent with the edges of the graph. Now consider what would happen if we processed the nodes of the graph in the order given by a topological sort.

By the time we process a node, we would have already processed every node that has a path leading to it. This means we would have considered every path to this node and will know the cheapest path from the source to it, regardless of whether the weights are positive or negative.

Remember the topological sort algorithm. We can use any structure we wish to maintain a WorkingSet. We start this set with all nodes that have no incoming edges. We know an acyclic directed graph must have at least one such node.

One at a time, we remove node X from the WorkingSet. We delete its outgoing edges and add to the working set any unprocessed node that now has no incoming edges. This continues until the queue empties. If some of the nodes of the graph have not been processed it must have a cycle.

We don't want to destroy our graph, so instead of deleting edges we store in each node an incoming edge count and decrement this each time the algorithm says we should delete an edge. When a node's incoming edge count is zero we add it t the WorkingSet.

For our shortest path algorithm we will use a simple linked-list queue for the WorkingSet, as it gives us constant-time insert and remove operations. We start the algorithms by walking through the edges of the graph to build the incoming edge count values and then to add to the queue any nodes with 0 incoming edges.

We give every node a cost, which initially is INFINITY for every node except the source node, which gets cost 0.

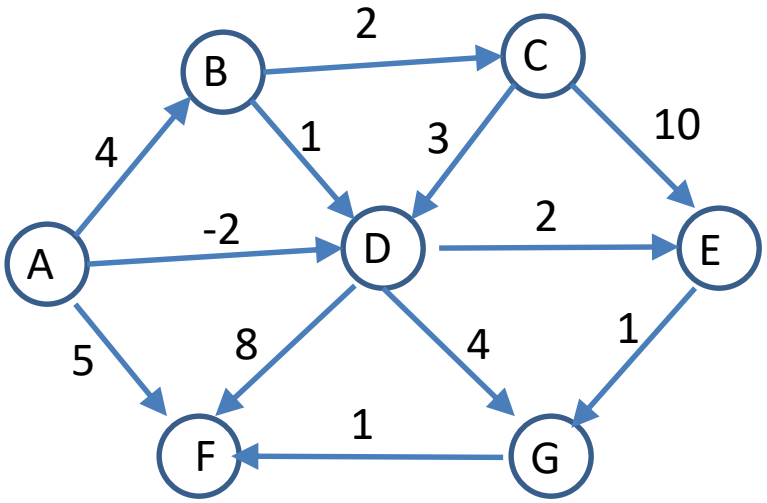
We take nodes out of the queue one at a time, walk through their outgoing edges and decrement the edge counts of each of the nodes at the end of their outgoing edges. Any node whose edge count becomes 0 is added to the queue.

This much is just the usual topological sort algorithm. We now add information that gets us the shortest path from a source node S.

Suppose we remove node X from the queue, and X has cost c, which is less than INFINITY. If there is an edge from X to Y with weight w, we compare Y's current cost with c+w. If c+w is smaller, we make Y's current cost c+w and make X Y's predecessor.

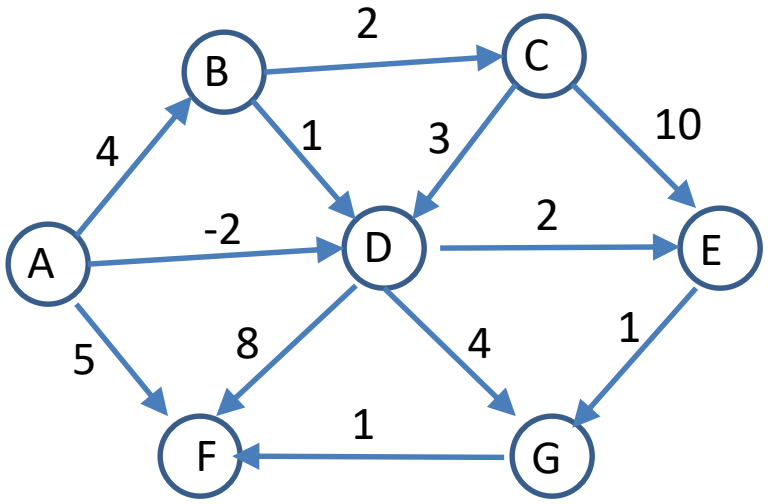
By the time Y goes into the queue, every possible path from the source to Y will have been examined, so Y's cost estimate will be the minimum cost from the source to Y.

For example, consider the following graph.



One possible topological ordering of the nodes is
A B C D E G F

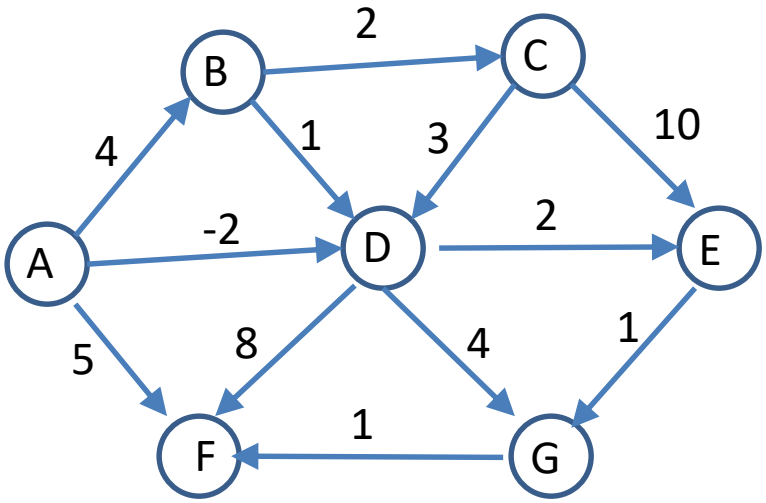
Let's find the minimum cost paths from A to each node.



Node A has cost 0. When it comes out of the queue we give costs to its adjacent nodes, B, D, and F:

A	B	C	D	E	G	F
0	4		-2			5

Next B comes out of the queue. Its cost of 4 is frozen. We give C a cost of 6. We don't change the cost of D because the path to D through B is more expensive.



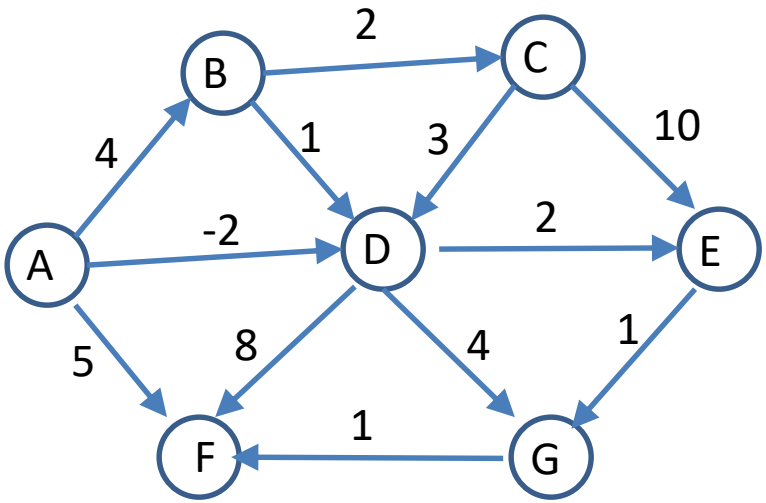
A	B	C	D	E	G	F
0	4	6	-2			5

C is the next node to come out of the queue, giving a cost of 16 to E.

A	B	C	D	E	G	F
0	4	6	-2	16		5

When D comes out of the queue we update the costs of E and G:

A	B	C	D	E	G	F
0	4	6	-2	0	2	5



E gives a path of cost 1 to G; G gives a path of cost 2 to F. Our final costs are

A	B	C	D	E	G	F
0	4	6	-2	0	1	2

How long does this take? We walk along every edge; every step is constant time. Our algorithm runs in time O(|E|). Note how much better this is than our negative-weight algorithm. This handles positive and negative weights, though it will not handle any graph for which there is a cycle.