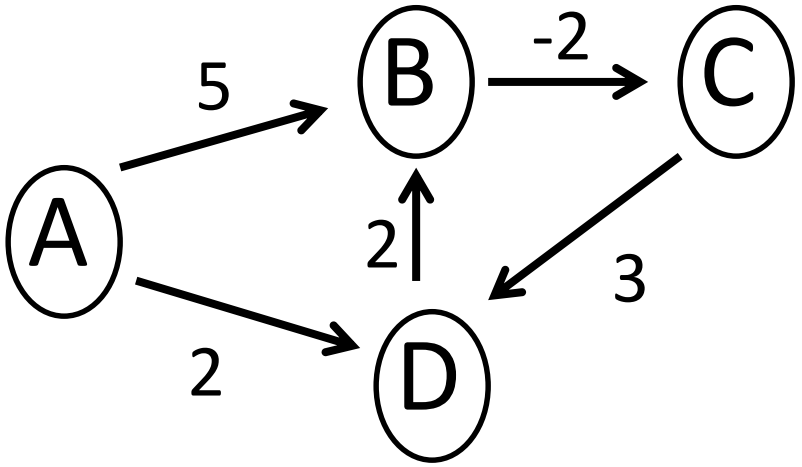
Shortest Path with Negative Weights

The Bellman-Ford Algorithm

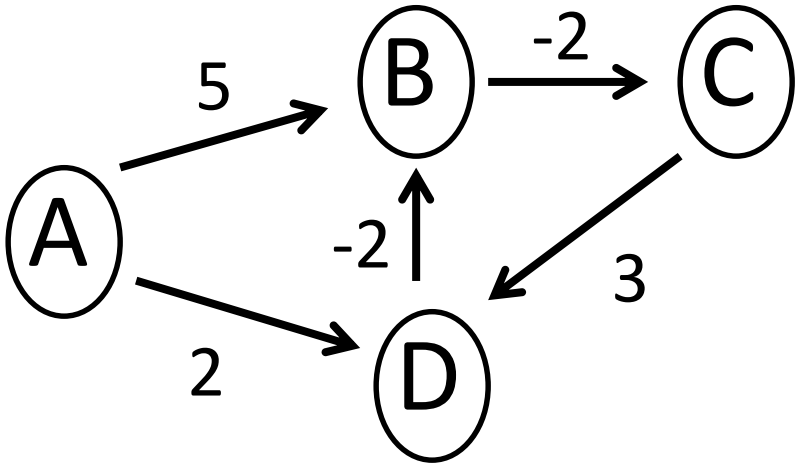
Question: What is the shortest path from A to C?



A. A -> B -> C
B. A -> D -> B -> C
C. A -> D -> B -> C -> D -> B -> C
D. There is no shortest path because there is a cycle.

Answer B: A->D->B->C which has total weight 2

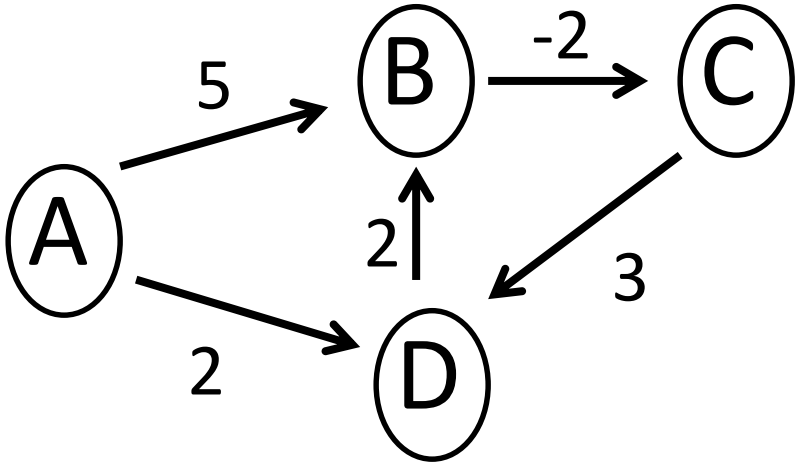
Different weights. Now what is the shortest path from A to C?



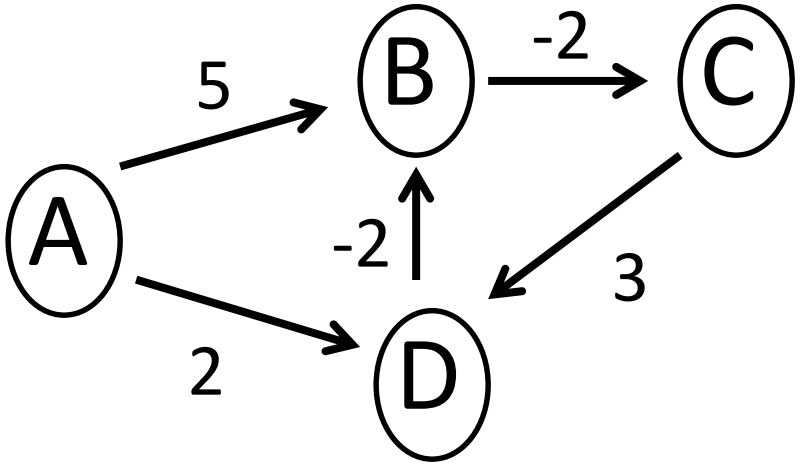
A. A -> B -> C
B. A -> D -> B -> C
C. A -> D -> B -> C -> D -> B -> C
D. There is no shortest path because there is a cycle.

Actually, none of these are quite correct. It is true that there is no shortest path, but as the previous example shows, just having a cycle doesn't by itself prevent there from being shortest paths.

What is the difference?



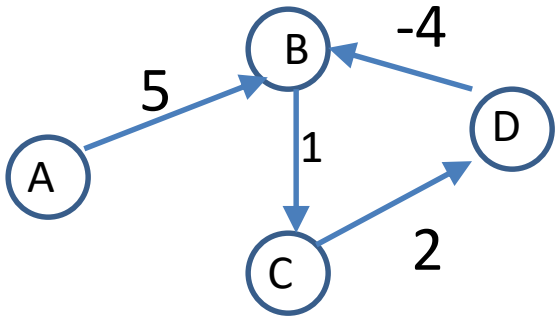
Shortest paths

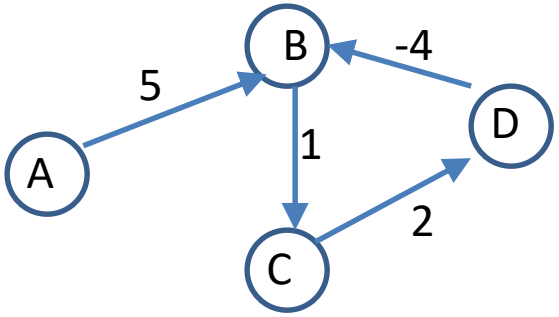


No shortest paths

Sometimes we want to find the shortest paths in a graph where the edges have weights that aren't necessarily positive. For example, suppose you are investing in currency exchange market. National currencies are sometimes overvalued or undervalued, so that if you use US dollars to buy British pounds, which you then sell for euros and use the euros to buy Japanese yen, you might find that the yen are worth more or less than the dollars you started with. You could model this with a graph whose nodes represent nations and whose edges represent the percentage gain or loss of transferring one currency for another. The cheapest path from one node to another represents the cheapest way to transform one currency into another.

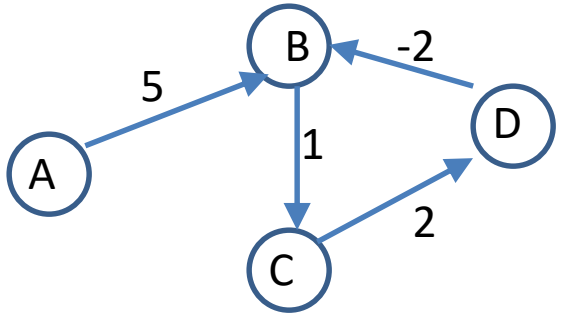
Note that once we introduce negative edge weights, there might not be cheapest paths. Consider the following graph:





There is a direct path from A to B of cost 5. However, the route A -> B -> C -> D -> B only has cost 4. If we go
A -> B -> C -> D -> B -> C -> D -> B the cost is only 3. If we went around the cycle 100 times the cost would be -95. It should be clear that there is no cheapest path from A to B; we can get a path as cheap as we wish by simply going around the cycle often enough.

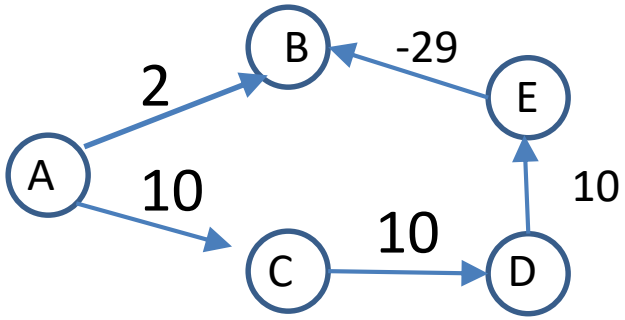
Our other shortest path algorithms (unweighted, non-negative weights) were not bothered by cycles. Cycles themselves are not the problem here. In the following graph



the cheapest path from A to B has cost 5, from A to C has cost 6 and from A to D has cost 8. Problems occur if the weights on the edges of a cycle sum to a negative number. Such a "negative-cost cycle" prevents there from being minimum cost paths.

Our first two algorithms for finding shortest paths were based on the fact that when we pulled a node out of our data structure (a queue for the unweighted problem, a priority queue for the positive-weights problem) we knew the shortest path to it. That doesn't apply here, since there might be an unexplored node with an edge back to our node that has a large negative cost. We can't know the shortest path to anything without exploring all edges in the graph.

For example, consider this graph with A as the source:



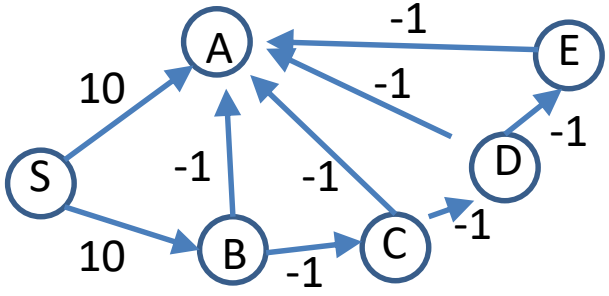
You might initially think the cheapest path to B is A => B, which costs 2, but there is a cheaper path of cost 1: A => C => D => E => B

We return to our queue data structure, which explores nodes based on the number of edges in their path from the source. Let X be the node at the head of the queue. When we remove X from the queue we know the cheapest path to it *so far*.

What does this mean? We first put into the queue all of the nodes with a path of length 1 (i.e, 1 edge). As we take these out we add behind them nodes with a path of length 2, behind these we add nodes with a path of length 3, and so forth. We could keep track of the number of edges on the path that its current cost represents. Each time we add a node to the queue the length of the best path to it is one more than the length of the best path to its predecessor.

For each of the nodes pointed to by X we can calculate a new cost: the cost of the path to X plus the cost of the edge from X to this node. If this is cheaper than the previous cost of the node, we update its cost to this and add it to the queue. If the queue ever empties out, that means there are no unexplored cheaper paths and we must have found all of the minimum cost paths. The queue won't empty out if there is a negative-cost cycle.

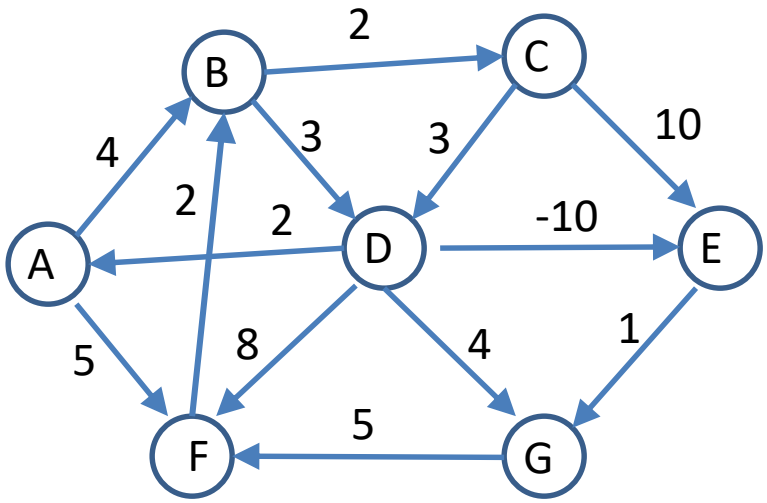
How can we detect a negative-cost cycle? Nodes will be added to the queue over and over. Consider the following graph:

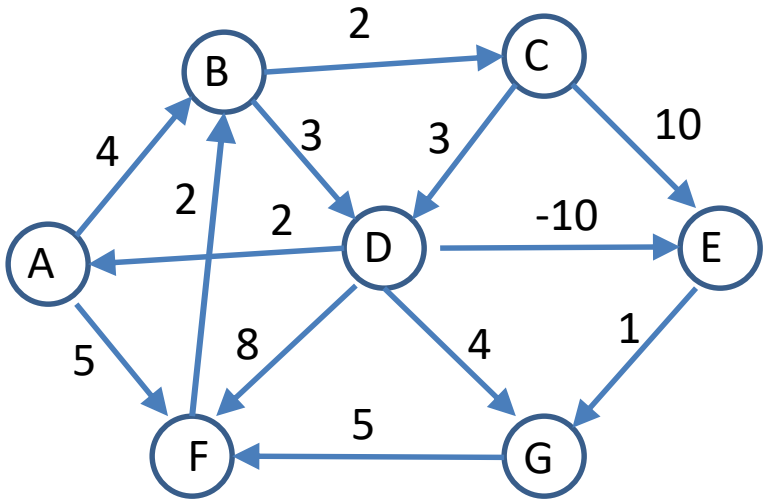


Here node A is added to the queue once for each other node in the graph.

Now suppose the graph has n nodes and there is a path to node X with n edges that is cheaper than any shorter path. If the graph has n nodes and this path has n edges (and so n+1 nodes), this means there is a repeated node, which means there is a cycle in the path. If the n-edge path is cheaper than the same path without the cycle, this must be a negative-cost cycle. So if this ever happens, there is no solution to the shortest path problem.

Here's an example. We will use A as the source node. For each step we will list the current state of the queue. We will give a second list with all of the nodes, the weight of the best path to them, their predecessor on this path, and the number of edges on this path.

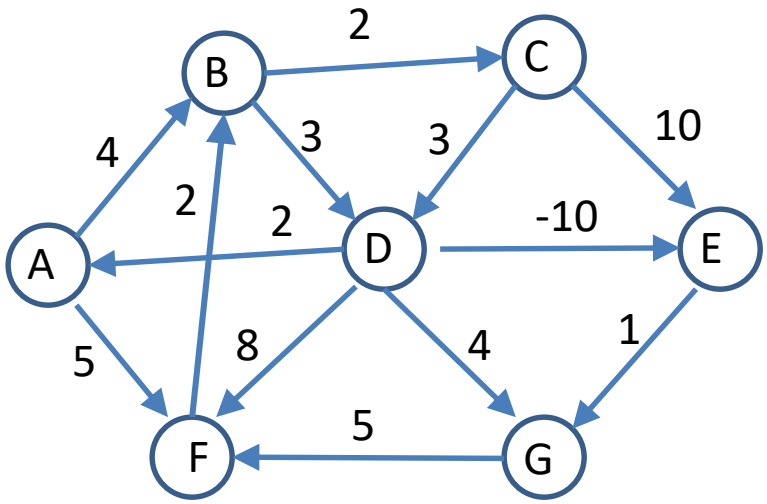




(A, 0, null, 0)
(B, INF, null, INF)
(C, INF, null, INF)
(D, INF, null, INF)
(E, INF, null, INF)
(F, INF, null, INF)
(G, INF, null, INF)

Queue: [A]

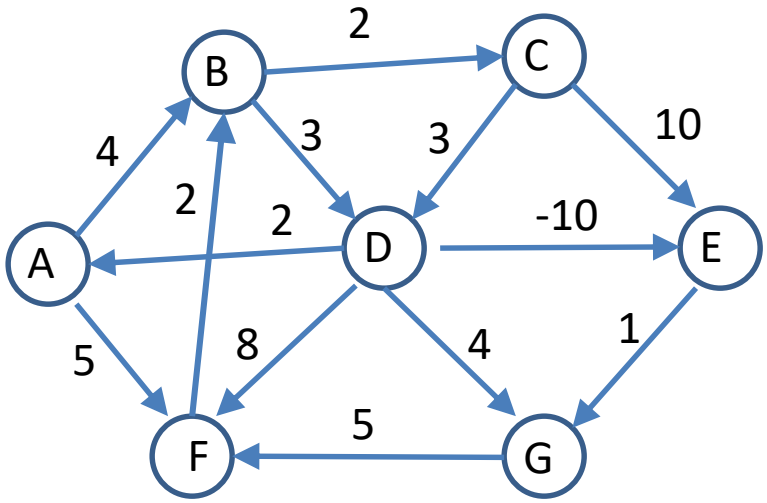
We start by removing A from the queue and adding its children B and F, updating their properties



(A, 0, null, 0)
(B, 4, A, 1)
(C, INF, null, INF)
(D, INF, null, INF)
(E, INF, null, INF)
(F, 5, A, 1)
(G, INF, null, INF)

Queue: [B, F]

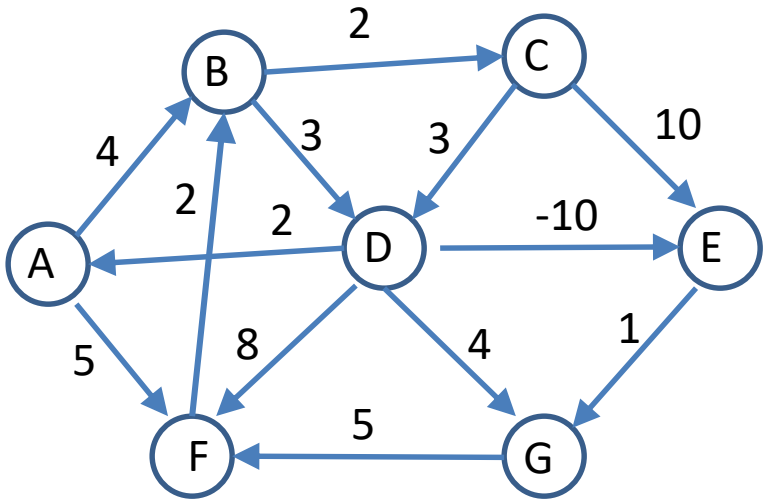
Next we remove B, which is connected to C and D



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, INF, null, INF)
(F, 5, A, 1)
(G, INF, null, INF)

Queue: [F, C, D]

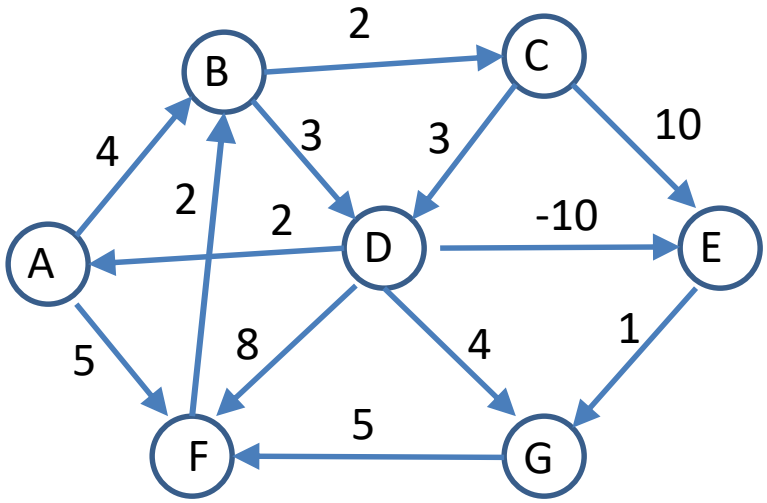
We remove F from the queue; it is connected only to B and we have a cheaper path to B, so no updates are needed.



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, INF, null, INF)
(F, 5, A, 1)
(G, INF, null, INF)

Queue: [C, D]

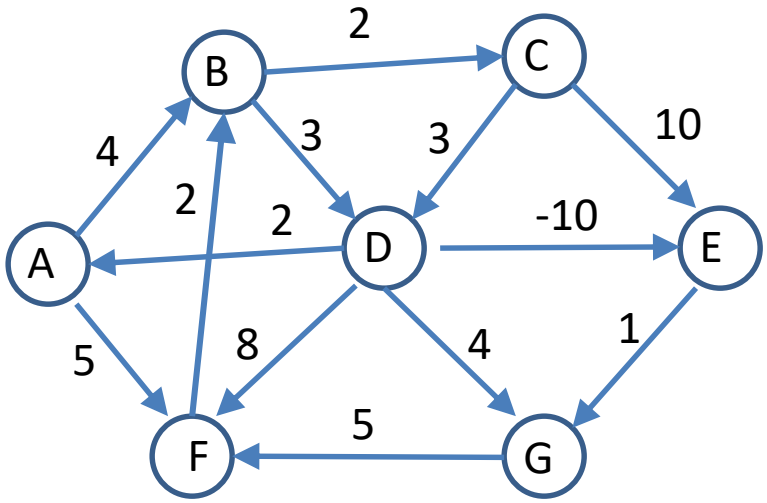
We remove C from the queue; it is connected to D but we have a cheaper path. C is also connected to E.



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, 16, C, 3)
(F, 5, A, 1)
(G, INF, null, INF)

Queue: [D, E]

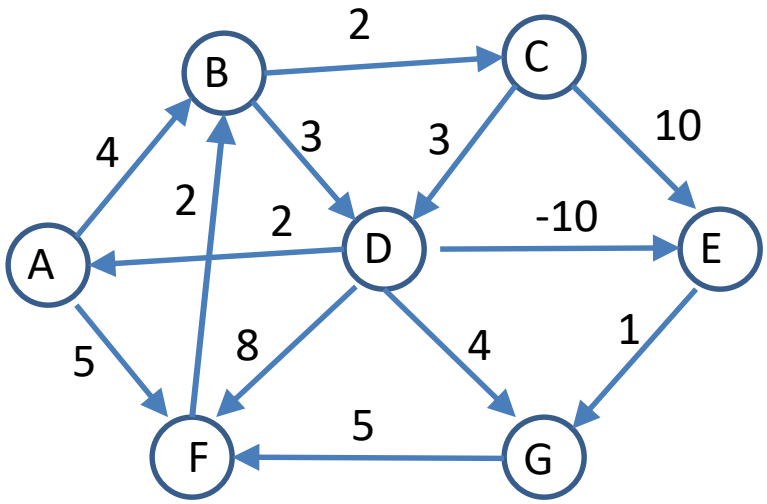
D give us bad connections to A and F, an improved connection to E (which we need to add to the queue again) and a new connection to G



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, -3, D, 3)
(F, 5, A, 1)
(G, 11, D, 3)

Queue: [E, E, G]

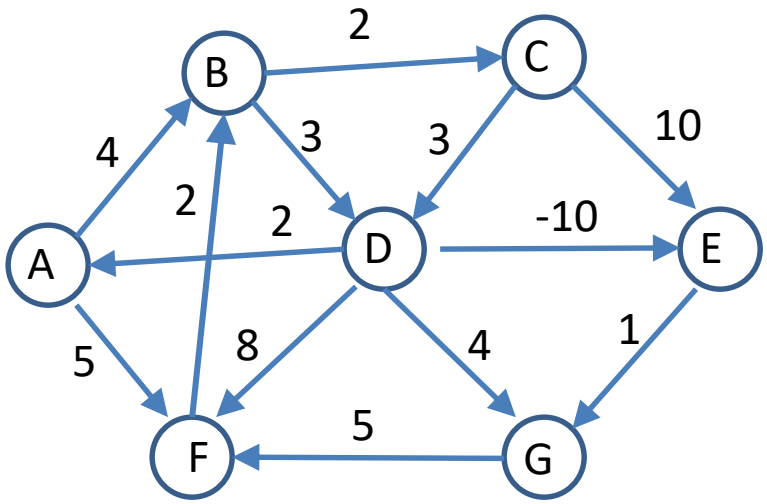
E gives us a better path to G, which we add to the queue again. Note that nothing changes when we remove the second E from the queue .



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, -3, D, 3)
(F, 5, A, 1)
(G, -2, E, 4)

Queue: [G, G]

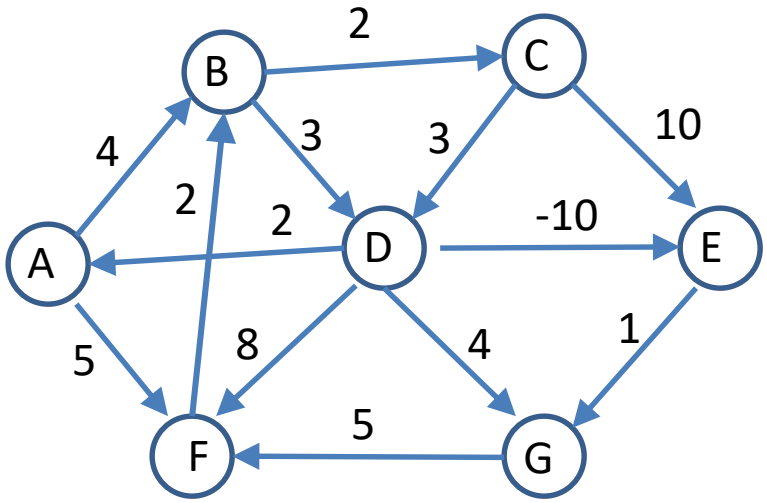
G gives us a better path to F.



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, -3, D, 3)
(F, 3, G, 5)
(G, -2, E, 4)

Queue: [F]

F does not give us a better path to B, so this step ends with the queue empty and we halt.



(A, 0, null, 0)
(B, 4, A, 1)
(C, 6, B, 2)
(D, 7, B, 2)
(E, -3, D, 3)
(F, 3, G, 5)
(G, -2, E, 4)

Queue: [ ]

Notice what would happen if the edge from F to B had weight 0. This would give us a new cheaper path to B of length 3 and we would keep going. The negative weight cycle B->D->E->G->F->B would keep going around until we found a path longer than the 7 nodes of the graph.

How long does this take?

We must add each vertex to the queue at most $|V|$ times, because each time we add it the number of edges on the best path to it increases. Each time we remove a vertex from the queue we potentially walk along all of the edges of the graph. so $O(|E| |V|)$ is our estimate. For most graphs $|V| < |E| < |V|^2$, so this gives an estimate between $O(|V|^2))$ and $O(|V|^3)$.

Our estimate for finding shortest paths for a graph with non-negative weights was O(|E| log(|E|)). Think about the difference between O(|E| |V|) and O(|E|log(|E|)) as the price of using negative weights.

In lab 9 we build some enormous graphs using IMDB data about movies and actors. A graph with 10,000 nodes and 100,000 edges is easy to construct. Here $|E||V|$ is 10^9. $|E|\log(|V|)$ is $5*10^5$ (using base-10 logs). The shortest path algorithm with positive weights runs faster by a factor of about 2000. This means that if finding shortest paths with positive weights takes 30 seconds, finding them with negative weights might take 16 hours. Of course, sometimes you can't avoid the negative weights and being slow might be better than not solving the problem at all.