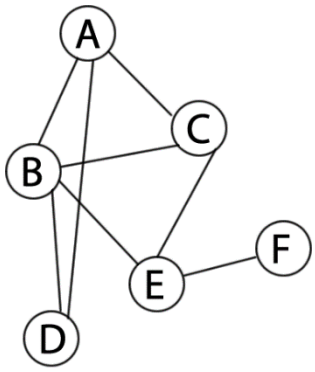
CSC 325 Adv Data Structures

Lecture 16

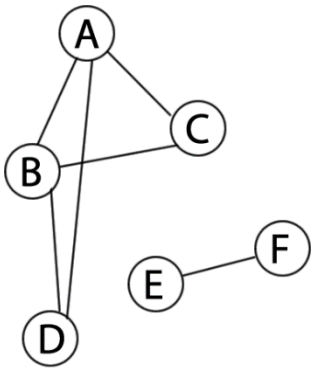
Graphs… Again

Background

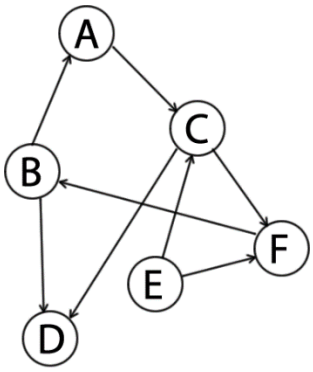
Graphs come in different flavors such as the following:



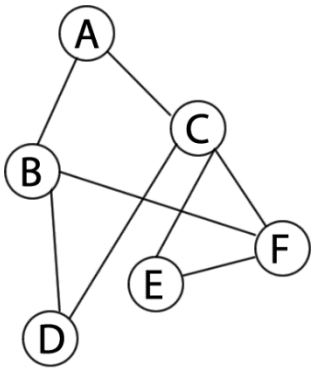
Connected



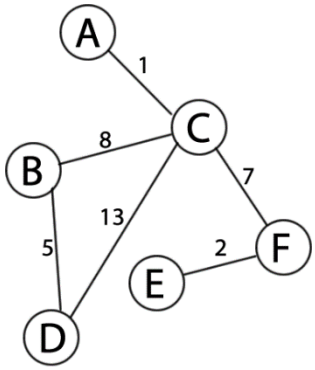
Not Connected



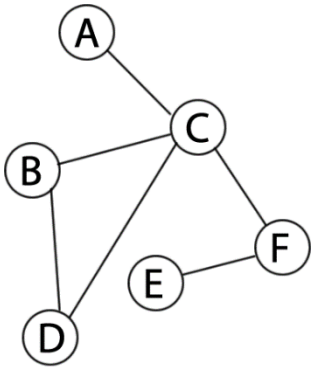
Directed



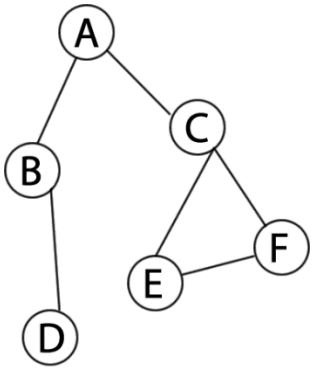
Undirected



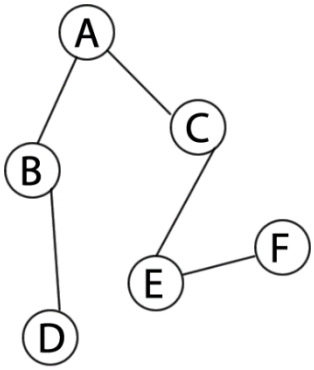
Weights



No Weights



Cyclic



Acyclic

Terminology

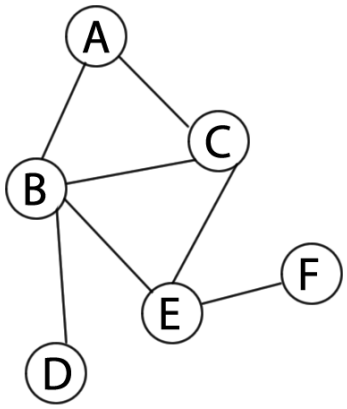
Graphs have a good bit of terminology that goes along with them. In addition to the terminology above (with the graph examples) there is:

- Vertex - A single container in our graph. Kind of like a node with linked lists, but in graphs we call them vertices
- Edge - A link between two vertices (for undirected graphs)
- Arc - A directed link between two vertices (for directed graphs)
- Neighbors - All vertices that can be reach via a single edge from a given vertex. For example, if V_2, V_3, and V_4 all have edges connecting them to V_1, we would say the neighbors of V_1 are V_2, V_3 and V_4.
- Path - A sequence of edges going from one vertex to another

Representations

How do we represent these graphs in code? We obviously can't represent them with pretty pictures like the ones above. We need an efficient representation. To do this, we focus on the neighbors of each vertex and order our information based on this. There are three main representations:

Given the following graph:



We could represent it as an adjacency matrix which is simply an integer matrix (i.e. an array of arrays) that contain a 1 if the vertex associated with the row has an edge to the vertex associated with the column, or a 0 otherwise. This would be the adjacency matrix of the graph above:

		A	B	C	D	E	F
		0	1	2	3	4	5
A	0	0	**1**	**1**	0	0	0
B	1	**1**	0	**1**	**1**	**1**	0
C	2	**1**	**1**	0	0	**1**	0
D	3	0	**1**	0	0	0	0
E	4	0	**1**	**1**	0	0	**1**
F	5	0	0	0	0	**1**	0

If the graph had positive weights associated with it, we would use those weights instead of 1, and 0 would still be used to indicate the absence of an edge. We could also use -1 if 0 was a possible valid weight.

Another representation is an adjacency list. Given the same graph above, here is its adjacency list representation:

A	0	B	C			
B	1	A	C	D	E	
C	2	A	B	E		
D	3	B				
E	4	B	C	F		
F	5	E				

Each vertex contains a list of their neighbors.

The final way is the so-called object oriented approach. Here we create a class for Vertex and Edge objects to be instantiated.

```
// Vertex class holding its name and list of edges
class Vertex
    String name
    Edge[] edges
end

// Edge class holding the vertex it points to as well as weight
class Edge
    Vertex to
    int weight  // if needed
end

// Vertex objects
a ← Vertex("A")
b ← Vertex("B")
c ← Vertex("C")
d ← Vertex("D")
e ← Vertex("E")
f ← Vertex("F")

// List of edges for each Vertex object
a.edges ← {Edge(b), Edge(c)}
b.edges ← {Edge(a), Edge(c), Edge(d), Edge(e)}
c.edges ← {Edge(a), Edge(b), Edge(e)}
d.edges ← {Edge(b)}
e.edges ← {Edge(b), Edge(c), Edge(f)}
f.edges ← {Edge(e)}
```

All three of these approaches are different ways to represent the same graph but they all focus on representing the graph as a collection of vertices and the neighbors of each. This is efficient since most of the time, the neighbors of a vertex is the most important thing we want to know.

Traversals

Now let's talk about traversing these graphs. It's not as simple as a binary tree or linked list. In fact there are different ways we can go about visiting each vertex. We will discuss two of those ways in this class.

Breadth-First Search

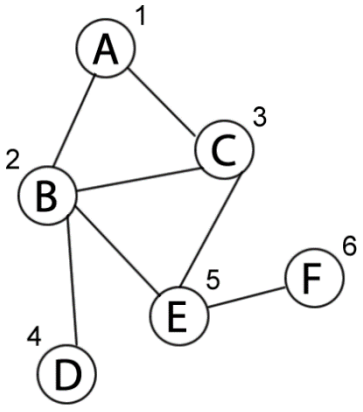
The first way is called Breadth-First Search (or BFS for short). The idea is to start with a vertex, visit all the neighbors of that vertex, then visit the neighbors of those neighbors, then the neighbors of those and so on. We do this until all vertices have been visited. In essence we try to stay as close as possible to the starting point, only going out one level at a time. Here is some pseudocode to help explain this procedure:

```
function makeAdjacencyList()
    list ← List of 6 vertices  // empty list for 6 vertices
    list at a ← {b, c}         // list containing neighbors of vertex 'a'
    list at b ← {a, c, d, e}   // list containing neighbors of vertex 'b'
    list at c ← {a, b, e}      // list containing neighbors of vertex 'c'
    list at d ← {b}            // list containing neighbors of vertex 'd'
    list at e ← {b, c, f}      // list containing neighbors of vertex 'e'
    list at f ← {e}            // list containing neighbors of vertex 'f'
    return list
end

adjList ← makeAdjacencyList()  // create our adjacency list and save it

function BFS(startVert)
    queue ← Queue()                // start with an empty Queue
    seenList at startVert ← true  // mark starting vertex as "seen"
    queue.enqueue(startVert)       // enqueue it so we can visit it later
    while queue is not empty  // go until we have no more vertices left to visit
        curVert ← queue.dequeue() // grab next vertex to visit
        visit(curVert)  // visit it, which may include printing or whatever
        foreach neighbor in adjList at curVert  // go to each neighbor
            if not (seenList at neighbor) then  // if we haven't seen it
                seenList at neighbor ← true // mark it so it's not added again
                queue.enqueue(neighbor) // add it to the queue for a later visit
            end
        next
    end
end
```

I chose to represent our graph as an adjacency list but any of the representations will work. To represent it as an adjacency list, I used our list abstract data type. The graph is then basically an array of lists. For BFS, we use a queue to store our vertices. If we start at vertex A, the visited order of the vertices is as follows:



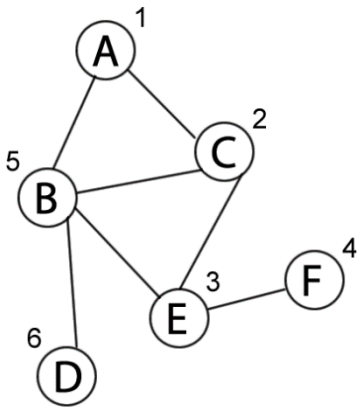
Depth-First Search

There is another way to traverse our graph called Depth-First Search (or DFS). Here is the algorithm for DFS:

```
adjList ← makeAdjacencyList()

function DFS(startVert)
    stack ← Stack()                // start with an empty Stack
    seenList at startVert ← true  // mark starting vertex as "seen"
    stack.push(startVert)          // push it so we can visit it later
    while stack is not empty  // go until we have no more vertices left to visit
        curVert ← stack.pop()  // grab next vertex to visit
        visit(curVert)  // visit it, which may include printing or whatever
        foreach neighbor in adjList at curVert  // go to each neighbor
            if not (seenList at neighbor) then  // if we haven't seen it
                seenList at neighbor ← true // mark it so it's not added again
                stack.push(neighbor)    // add it to the stack for a later visit
            end
        next
    end
end
```

If you notice, this algorithm is *exactly* the same as BFS! The only difference is that we are using a stack instead of a queue. This allows us to travel as far away from the source, as quickly as possible, returning only if we reach a dead end. This time the visited order (starting from A) is:

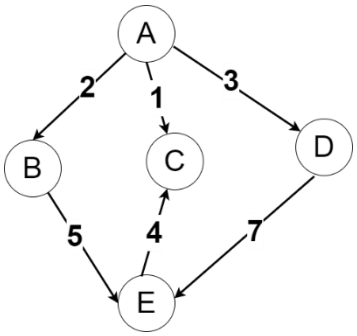


Basically, the process for both BFS and DFS is the following:

- Enqueue/Push the starting vertex and mark it as seen
- Do the following as long as our queue/stack isn't empty
    - Dequeue/Pop a vertex and visit it
    - Do the following for each neighbor
        - If not seen
            - Enqueue/Push and mark as seen

What if our graphs had weights on them? Perhaps the weights indicated some cost for going along each edge. What if we wanted to not only know of a way to traverse all vertices in the graph, but we also wanted to know the least cost way to do so? This is the motivation behind our next concept.

What if the graph has weights associated with it? How about if the graph is directed? Well, for the directed part, we can represent that with an adjacency list:

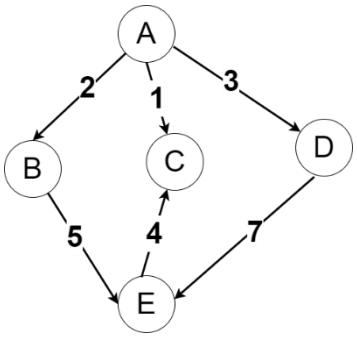


A	0	B	C	D
B	1	E		
C	2			
D	3	E		
E	4	C		

But we lose the weight information. We could make an alteration to the adjacency list representation like so:

A	0	B:2	C:1	D:3
B	1	E:5		
C	2			
D	3	E:7		
E	4	C:4		

However, I think the adjacency matrix representation works better for graphs with weights. Here is what that would look like:



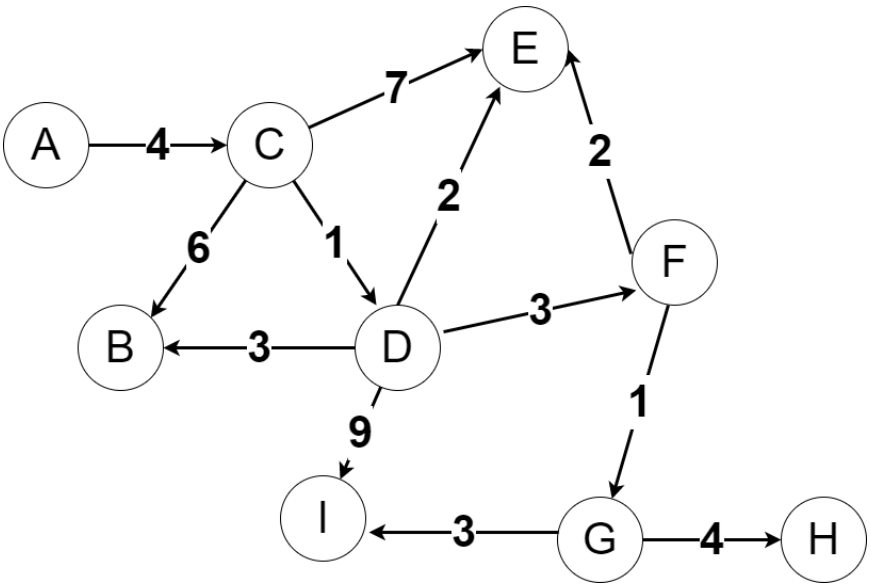
		A 0	B 1	C 2	D 3	E 4
A	0	0	2	1	3	0
B	1	0	0	0	0	5
C	2	0	0	0	0	0
D	3	0	0	0	0	7
E	4	0	0	4	0	0

Like before, we use zero to indicate that there is no edge there. Any value greater than zero, indicates an edge and is present and has the weight displayed. Now that we know how to represent weights and directed edges, let's see what we can do with this information.

Now that we have weights on our edges (technically these are called "arcs" if they are directed), we can use this information to find the shortest path from one vertex to another. In fact, we can find the shortest path from a vertex, to every other vertex. This is called the "single source shortest path". We take a single source (a single vertex of our choosing) and find the shortest path to every other vertex from that source. For example, if we take vertex A above, we can find the shortest path from A to B, from A to C, from A to D and from A to E. We need to know which path to take in each case and the weights will let us know what the length of that path is. There are several algorithms to find this. Let's go over one of the most common ones, Dijkstra's.

Dijkstra's

Let's use the following graph for this algorithm:



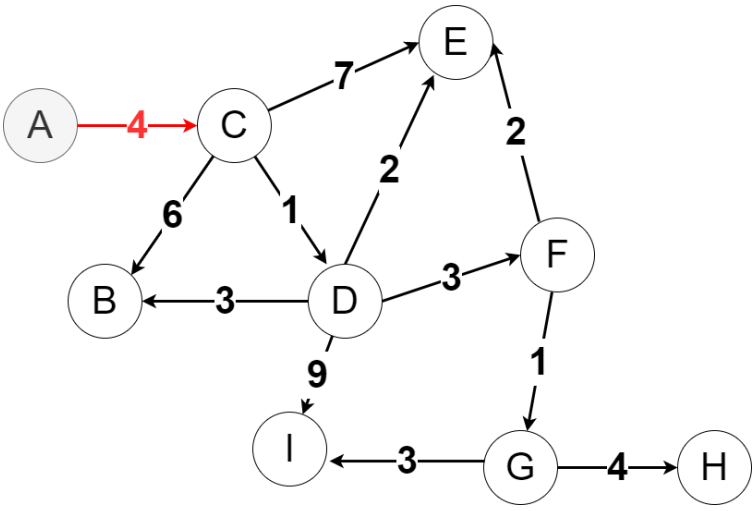
We want to find the shortest path from A to every other vertex. We are going to need a list of vertices we haven't yet visited and a list of path lengths. Since we haven't visited any vertex yet, our list of vertices initially include all vertices in the graph:

Unvisited: [A, B, C, D, E, F, G, H, I]

Since we don't know the path lengths yet, we initially start all lengths off at infinity (with 0 for A since going from A to A doesn't cost anything):

0	∞	∞	∞	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H	I

Let's officially visit A and see what all vertices we can reach from A.



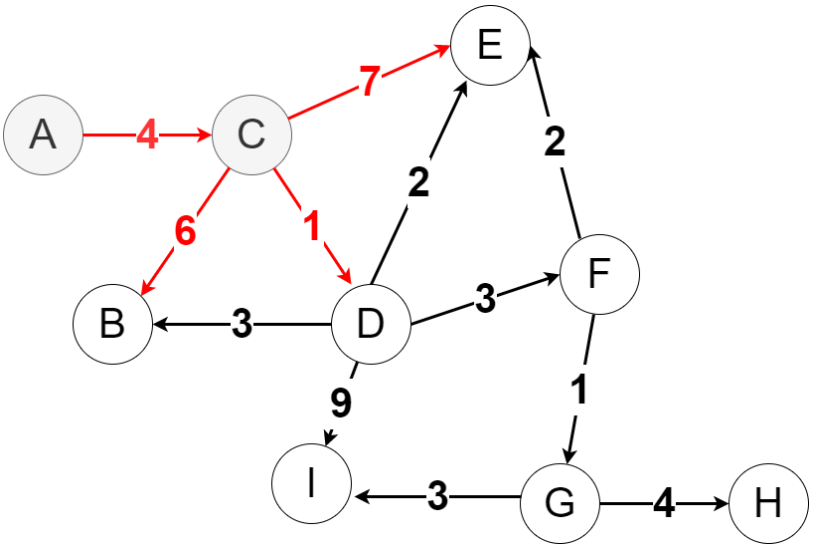
Since we have discovered that we can reach C from A at a total distance of 4, we compare this to the distance we know so far. Well, 4 is smaller than infinity, so we update the distance:

0	∞	4	∞	∞	∞	∞	∞	∞
A	B	C	D	E	F	G	H	I

Now that we have successfully visited A, let's remove it from our list:

Unvisited: [~~A~~, **B, C, D, E, F, G, H, I**]

Next, we take the vertex with the shortest known path from A, which is C. Let's see all the vertices we can reach from C:



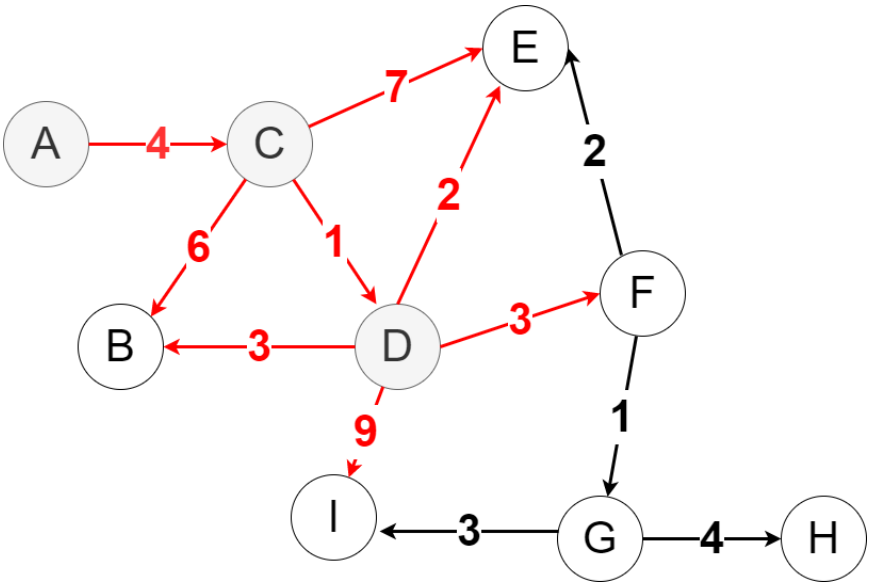
We see that we can go from C to E with a cost of 7, combine that with the current cost getting to C (which is 4) and we arrive at a total cost of 11 to get to E. This is less than our previous cost of infinity, so we update our cost to get to E. We do the same for D (which is the cost to get to C plus the cost to get to D, which is 4 + 1) and we do this for B as well. Let's update our table:

0	10	4	5	11	∞	∞	∞	∞
A	B	C	D	E	F	G	H	I

Now that we have visited C, let's remove it from our list:

Unvisited: [~~A~~, **B**, ~~C~~, **D, E, F, G, H, I**]

Let's look at the vertex with the least cost that we haven't yet visited, that being vertex D:



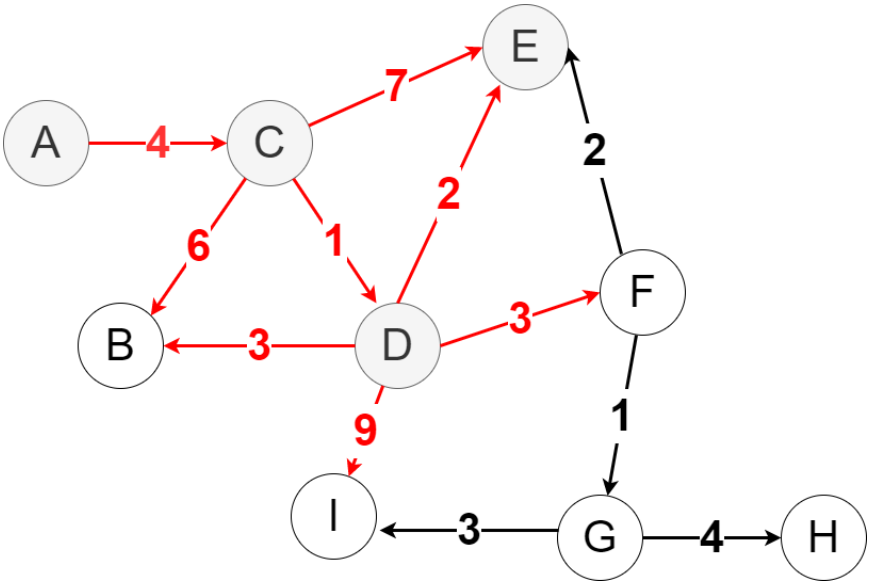
Now we can examine all adjacent vertices to D to see if any of them form a path that is shorter than what we currently have. We can see that we can get to E at a cost of 2. Taken into account the smallest cost to get to D, which is 5, add the cost to get to E, 2 and that gives us a total cost of 7. This is smaller than the cost we already knew about which was 11 (i.e. A to C then C to E). Let's update all of our path costs:

0	8	4	5	7	8	∞	∞	14
A	B	C	D	E	F	G	H	I

Now that we have visited D, let's cross it off of our list:

Unvisited: [~~A~~, **B,** ~~C~~, ~~D~~, **E, F, G, H, I**]

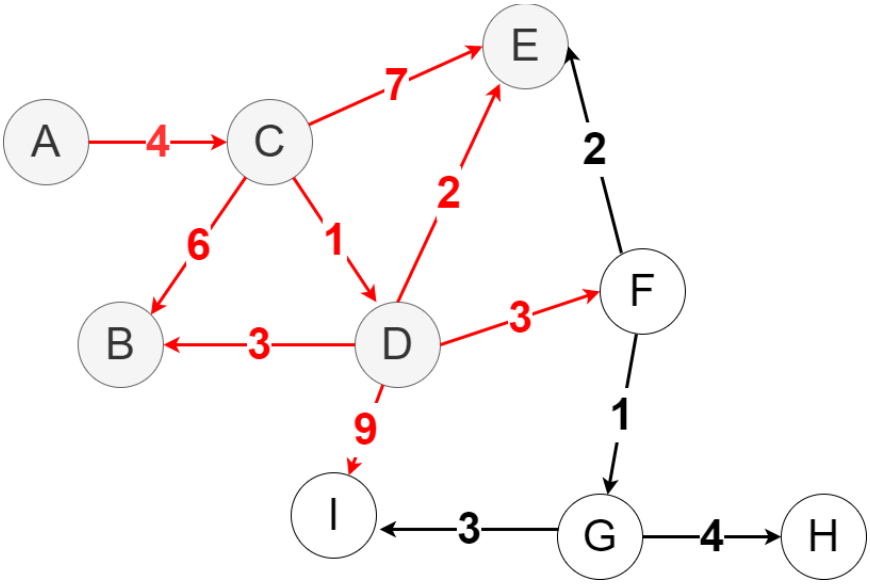
Let's pick the next vertex that we haven't yet visited but has the current smallest cost from A, which is E.



Unfortunately, E doesn't help us much, since there are no outgoing arcs from E. So let's just cross it off of our list and continue:

Unvisited: [~~A~~, **B**, ~~C~~, ~~D~~, ~~E~~, **F, G, H, I**]

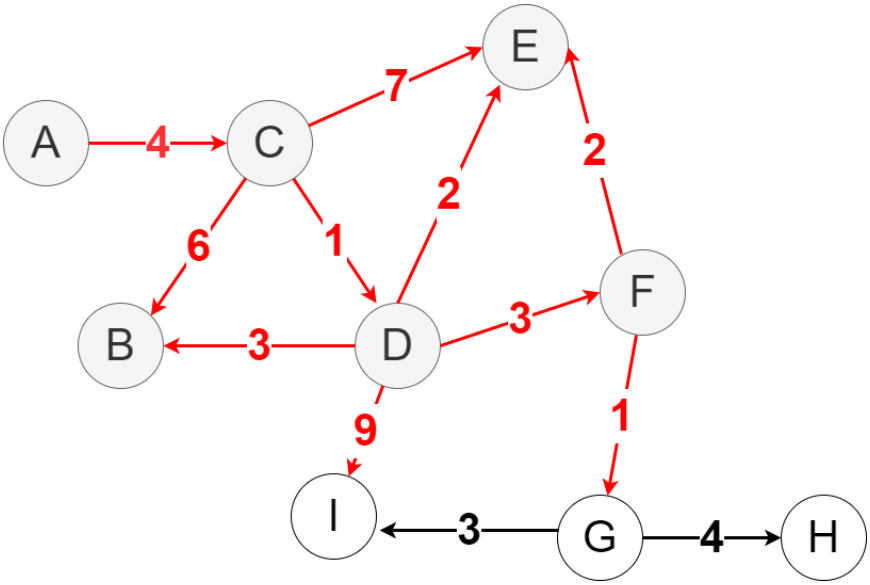
Next unvisited vertex with smallest cost is B. So let's visit that:



Once again we aren't going to get much help here. Let's cross it off the list:

Unvisited: [~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, **F, G, H, I**]

Next vertex is F.



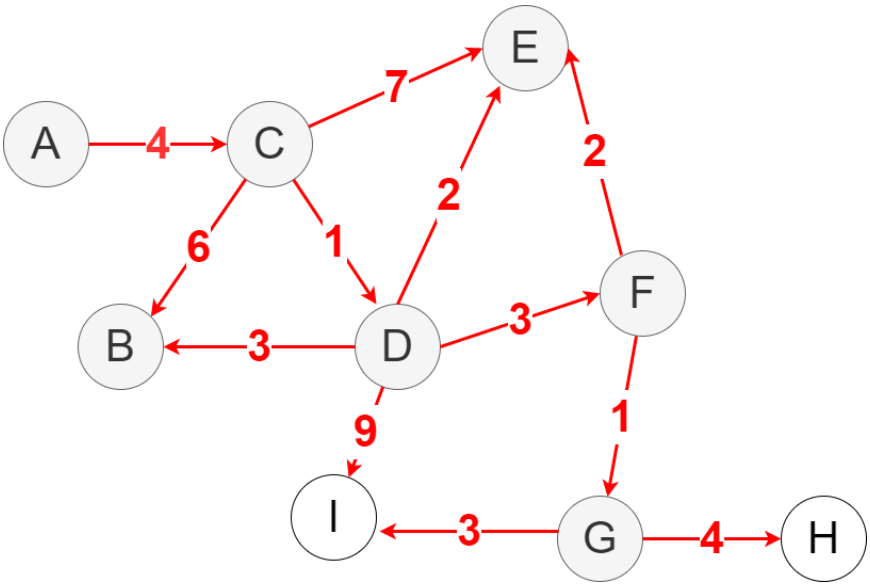
Here we need to examine E and G (since those can be immediately reached from F). The cost to get to E is 2. Combined with the smallest cost to get to F, which is 8, this gives us a total cost of 10. This is NOT smaller than the cost we already have in our table, which is 7, so we are not going to change the table for this one. To put this another way, we are basically saying that the path to get to E that goes through F (which is A->C->D->F->E) is at a total cost of 10, which is not shorter than the current shortest path to E that we know of (which is A->C->D->E) which has a total cost of 7. We don't currently have a good path to G, however, as our table still shows infinity for that. So we can update the table there:

0	8	4	5	7	8	9	∞	14
A	B	C	D	E	F	G	H	I

Now let's cross F off of the list:

Unvisited: [~~A~~, ~~**B**~~, ~~C~~, ~~D~~, ~~E~~, ~~F~~, **G, H, I**]

The vertex with the next shortest path that we have not visited yet is G. So let's visit that:

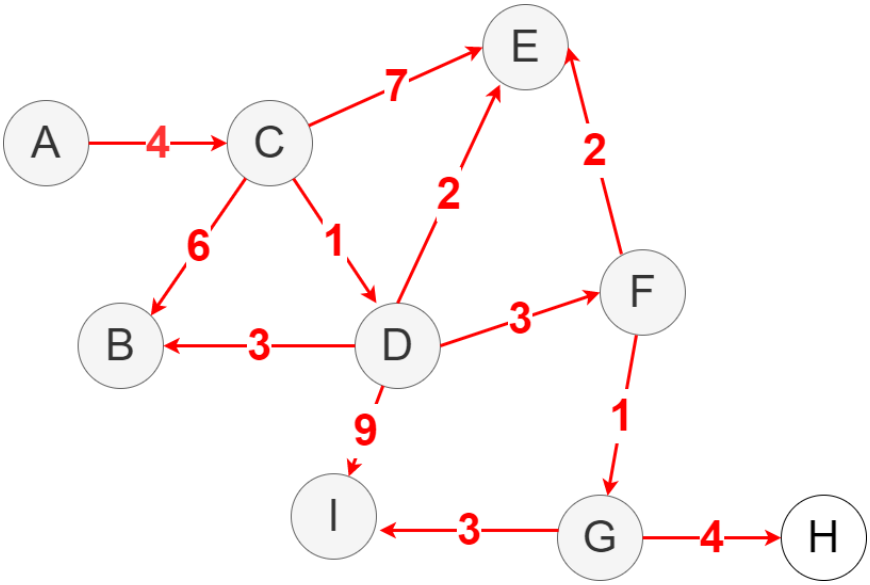


This reveals a path to H and a shorter path to I:

0	8	4	5	7	8	9	13	12
A	B	C	D	E	F	G	H	I

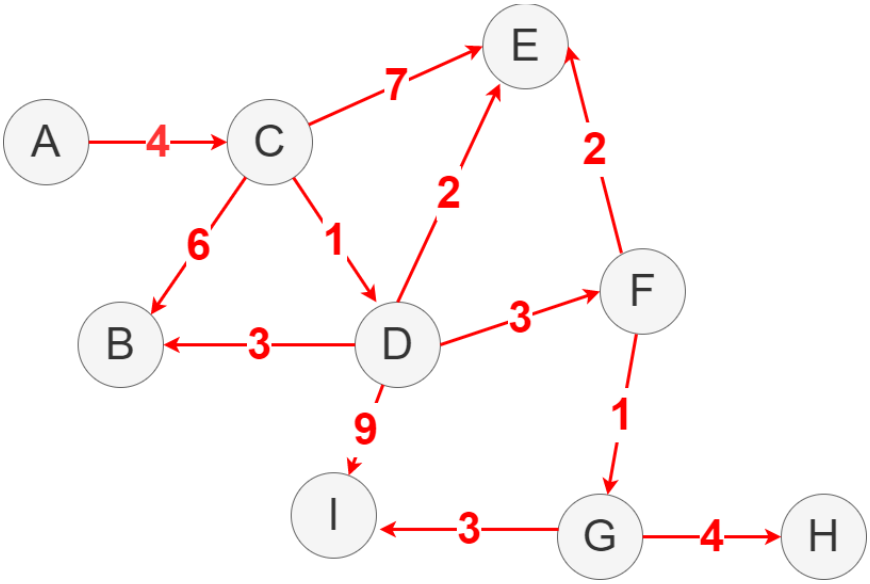
Now cross it off of the list and visit I next:

Unvisited: [~~A~~, ~~**B**~~, ~~C~~, ~~D~~, ~~E~~, ~~F~~, ~~G~~, **H, I**]



I doesn't have any outgoing arcs so we don't do anything here. Cross it off the list and visit H:

Unvisited: [~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~F~~, ~~G~~, ~~**H,**~~ ~~I~~]



No outgoing arcs, so H is done. Cross it off of our list:

Unvisited: [~~A~~, ~~B~~, ~~C~~, ~~D~~, ~~E~~, ~~F~~, ~~G~~, ~~H~~, ~~I~~]

Ok, that's all the vertices visited. Our final list of shortest distances from A is the following:

0	8	4	5	7	8	9	13	12
A	B	C	D	E	F	G	H	I

Here is another look at this showing only the arcs that contribute to the shortest distances: