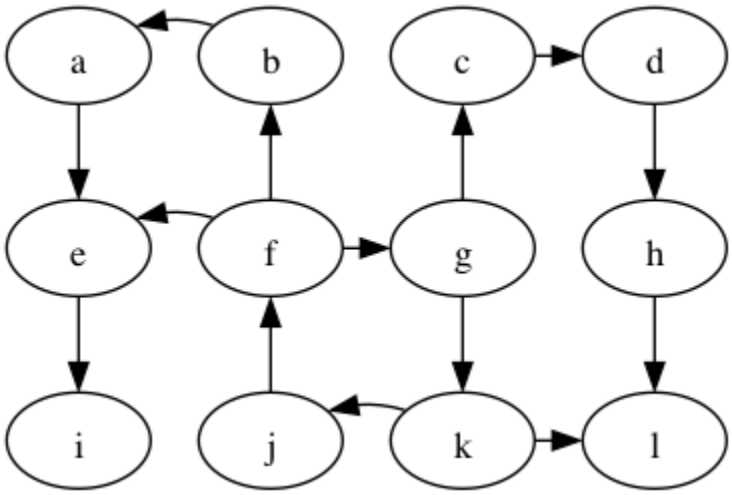
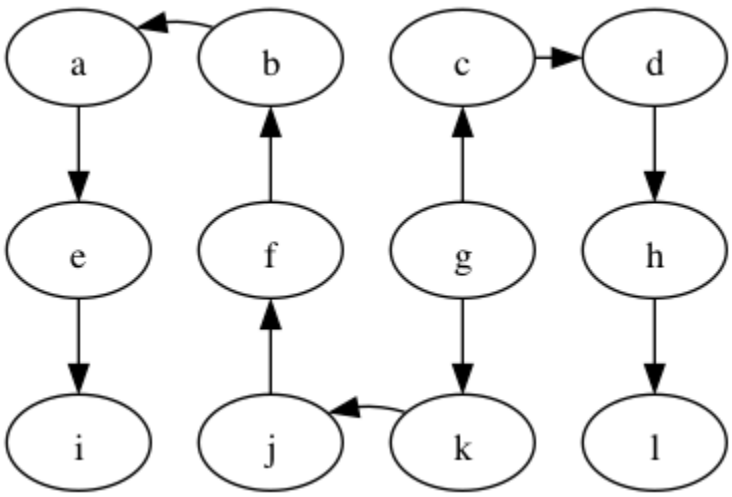
Depth-first Search

Depth-first search chooses to go deeper at each step, following an out-edge from the current vertex to a never-before-seen vertex. If there are no out-edges to never-before-seen vertices, then the search backtracks to the last visited vertex with out-edges to never-before-seen vertices and continues from there.

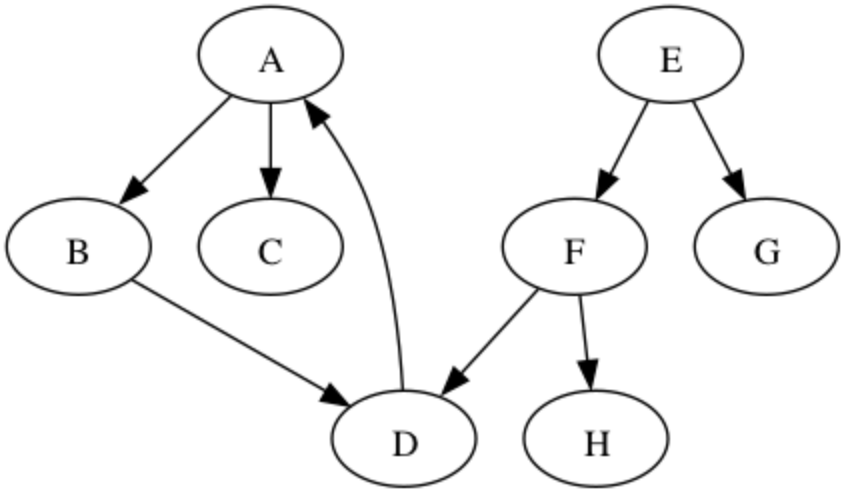
Show DFS on the following graph, starting at g:



Similar to BFS, DFS traverses a tree, a **depth-first tree**:



Sometimes all the nodes in a graph are not reachable from the starting node. Consider the following graph, starting at vertex A.



The search from A will reach B, C, and D, but not E, F, G and H. In such situations, one can restart the depth-first search at another vertex. For example, we could restart at the first vertex (alphabetically) that has not yet been visited, which would be E. The depth-first search from E would then reach the remaining vertices, including F, G, and H. But what happens with vertex D? It is reachable from both A and E. The usual thing is to not revisit any nodes, even after restarting. So vertex D would be visited in the search that started from A but not in the search that started from E.

There are two ways to implement DFS. The first is like the algorithm for BFS, but replaces the queue with a stack.

```
stack.push(start)
while not stack.empty()
  u = stack.pop()
  if not visited[u]
        visited[u] = true
        for v in G.adjacent(u)
          if not visited[v]
                parent_map[v] = u
                stack.push(v)
```

The second algorithm for DFS is recursive:

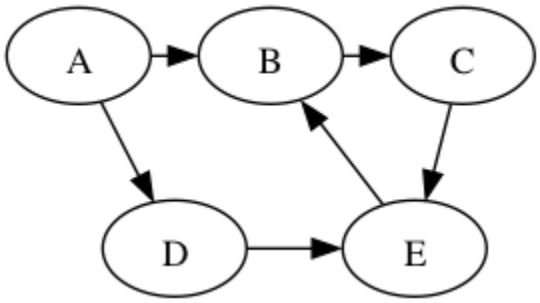
```
DFS(u, G, parent_map, visited) =
  if not visited[u]
        visited[u] = true
        for v in G.adjacent(u)
          if not visited[v]
                parent_map[v] = u
                DFS(v, G, parent_map, visited)
```

DFS and BFS both are good choices for generic search problems, that is, when you're searching for a vertex.

- BFS may require more storage if the graph has many high degree vertices because the queue gets big.
- On the other hand, if the graph contains long paths, then DFS will use a lot of storage for it's stack.
- If there are infinitely long paths in the graph, then DFS may not be a good choice.

Student group work

Perform DFS on the following graph

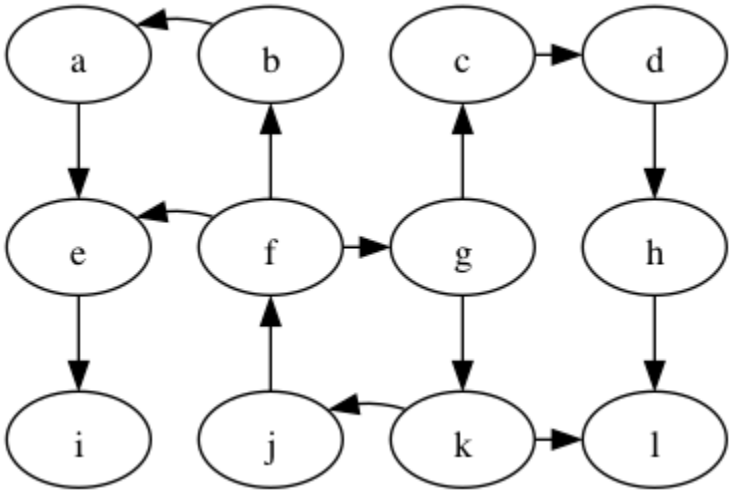


Are any of the DFS trees the same as a BFS tree for this graph? Why not?

(Solution at the bottom of the lecture.)

Edge Categories

In the following we'll continue to use this example graph:



We can categorize the edges of the graph with respect to the depth-first tree in the following way:

- **tree edge**: an edge on the tree, e.g., g → c in the graph above.
- **back edge**: an edge that connects a descendant to an ancestor with respect to the tree, e.g., f → g.
- **forward edge**: an edge that connects an ancestor to a descendant wrt. the tree, e.g., f → e
- **cross edge**: all other edges, e.g., k → l.

A graph has a cycle if and only if there is a back edge.

The DFS algorithm can compute the categories if we use a color scheme to mark the vertices instead of just using a `done` flag. The colors are:

- **white**: the vertex has not been discovered
- **gray**: the vertex has been discovered but some of its descendents have not yet been discovered
- **black**: the vertex and all of its descendents have been discovered

During DFS, when considering an out-edge of the current vertex, the edge can be categorized as follows:

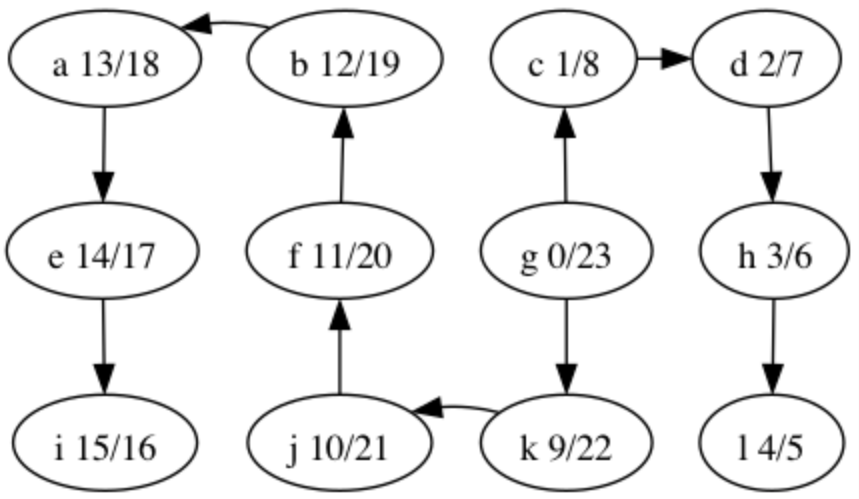
- the target vertex is white: this is a tree edge
- the target vertex is gray: this is a back edge
- the target vertex is black: this could either be a forward or cross edge

Do the DFS example again but with the white/gray/black colors.

Discover and Finish Times

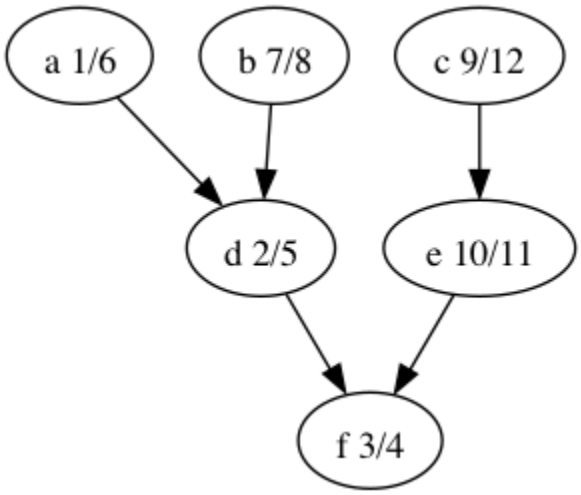
We can discover yet more structure of the graph if we record timestamps on a vertex when it is discovered (turned from white to gray) and when it is finished (turned from gray to black).

This is useful when constructing other algorithms that use DFS, such as topological sort. Here are the discover/finish times for the DFS tree we computed above, shown again below.



What's the relationship between topological ordering and depth-first search?

Let's look at the depth-first forest and discover/finish times of the makefile graph.



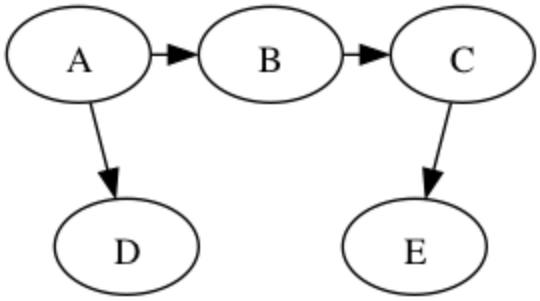
Here's the vertices ordered by finish time: f,d,a,b,e,c.

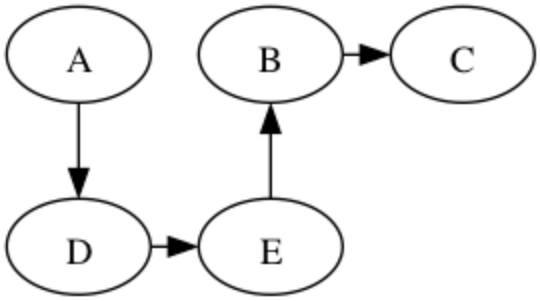
That's the reverse of one of the topological orders: c,e,b,a,d,f.

Why is that? A vertex is finished *after* every vertex that depends on it is finished. That's the same as topological ordering except we've swapped *before* for *after*.

Exercise Solutions

There are multiple depth-first trees:



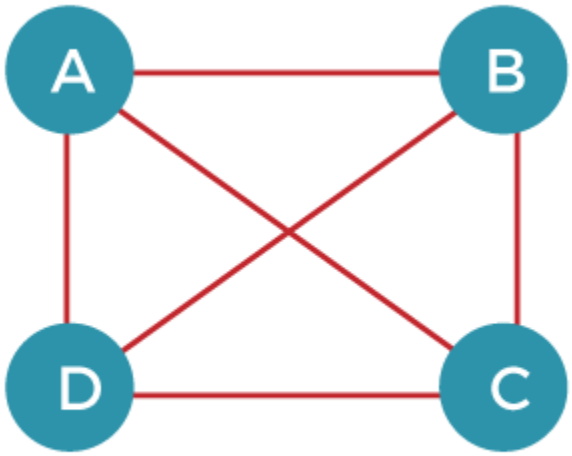


Shortest Path, Minimum Spanning Tree

Minimum Spanning Tree

Before knowing about the minimum spanning tree, we should know about the spanning tree.

To understand the concept of spanning tree, consider the below graph:

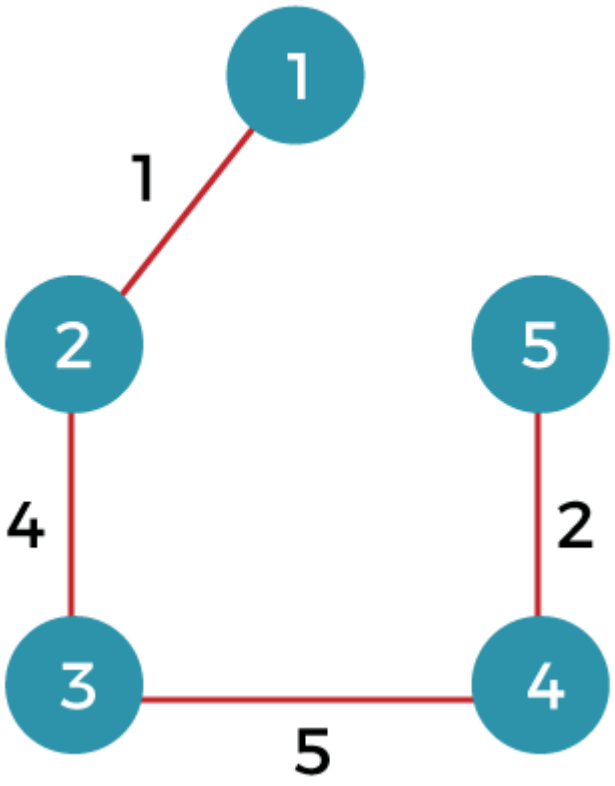


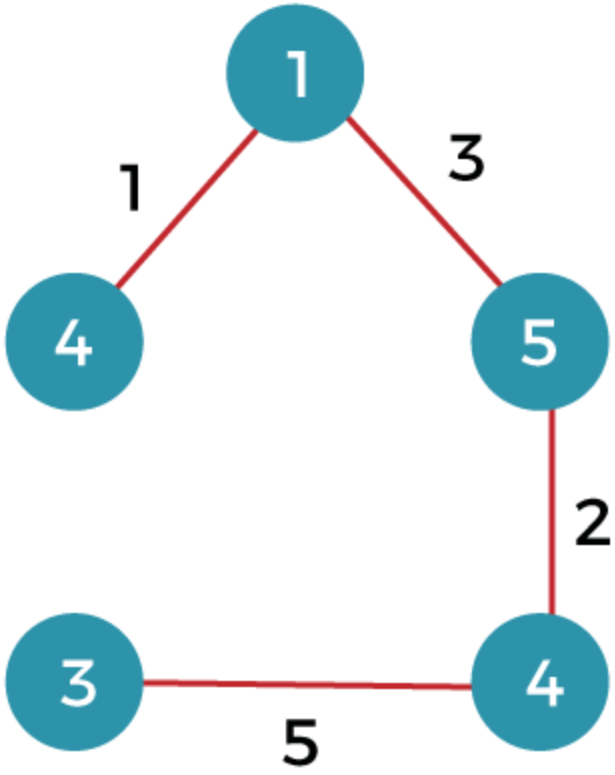
The above graph can be represented as G(V, E), where 'V' is the number of vertices, and 'E' is the number of edges. The spanning tree of the above graph would be represented as G`(V`, E`). In this case, V` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different. The number of edges in the spanning tree is the subset of the number of edges in the original graph. Therefore, the number of edges can be written as:
It can also be written as:

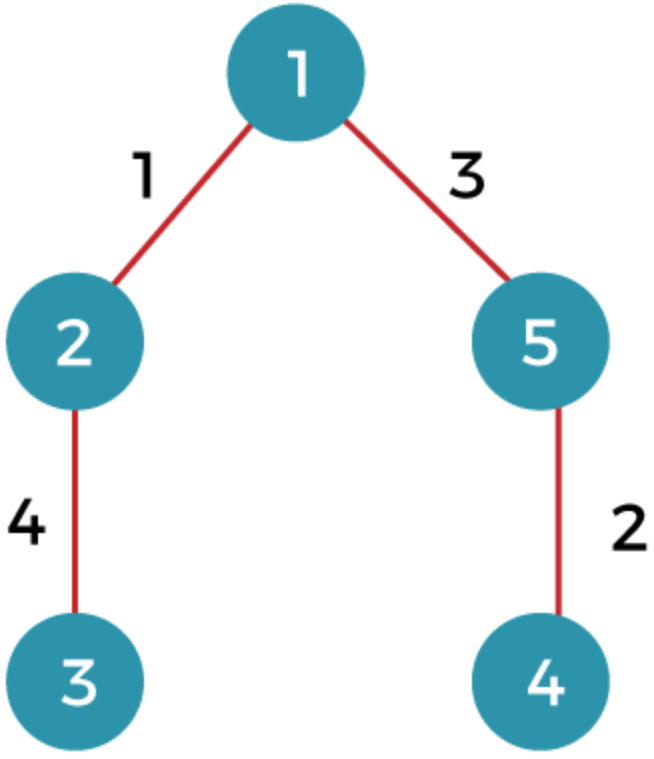
E` = |V| - 1

Two conditions exist in the spanning tree, which is as follows:

- The number of vertices in the spanning tree would be the same as the number of vertices in the original graph.
  V` = V
- The number of edges in the spanning tree would be equal to the number of edges minus 1.
  E` = |V| - 1
- The spanning tree should not contain any cycle.
- The spanning tree should not be disconnected.
- Consider the below graph:
- The above graph contains 5 vertices. As we know, the vertices in the spanning tree would be the same as the graph; therefore, V` is equal 5. The number of edges in the spanning tree would be equal to (5 - 1), i.e., 4. The following are the possible spanning trees:




1
1
3
4
5
2
3
4
5


1
1
3
2
5
4
2
3
4

What is a minimum spanning tree?

The minimum spanning tree is a spanning tree whose sum of the edges is minimum. Consider the below graph that contains the edge weight:

The following are the spanning trees that we can make from the above graph.

- The first spanning tree is a tree in which we have removed the edge between the vertices 1 and 5 shown as below:
  The sum of the edges of the above tree is (1 + 4 + 5 + 2): 12
- The second spanning tree is a tree in which we have removed the edge between the vertices 1 and 2 shown as below:
  The sum of the edges of the above tree is (3 + 2 + 5 + 4) : 14
- The third spanning tree is a tree in which we have removed the edge between the vertices 2 and 3 shown as below:
  The sum of the edges of the above tree is (1 + 3 + 2 + 5) : 11
- The fourth spanning tree is a tree in which we have removed the edge between the vertices 3 and 4 shown as below:
  The sum of the edges of the above tree is (1 + 3 + 2 + 4) : 10. The edge cost 10 is minimum so it is a minimum spanning tree.

General properties of minimum spanning tree:

- If we remove any edge from the spanning tree, then it becomes disconnected. Therefore, we cannot remove any edge from the spanning tree.
- If we add an edge to the spanning tree then it creates a loop. Therefore, we cannot add any edge to the spanning tree.
- In a graph, each edge has a distinct weight, then there exists only a single and unique minimum spanning tree. If the edge weight is not distinct, then there can be more than one minimum spanning tree.
- A complete undirected graph can have an n^{n-2} number of spanning trees.
- Every connected and undirected graph contains atleast one spanning tree.
- The disconnected graph does not have any spanning tree.
- In a complete graph, we can remove maximum (e-n+1) edges to construct a spanning tree.

Dijkstra's Algorithm

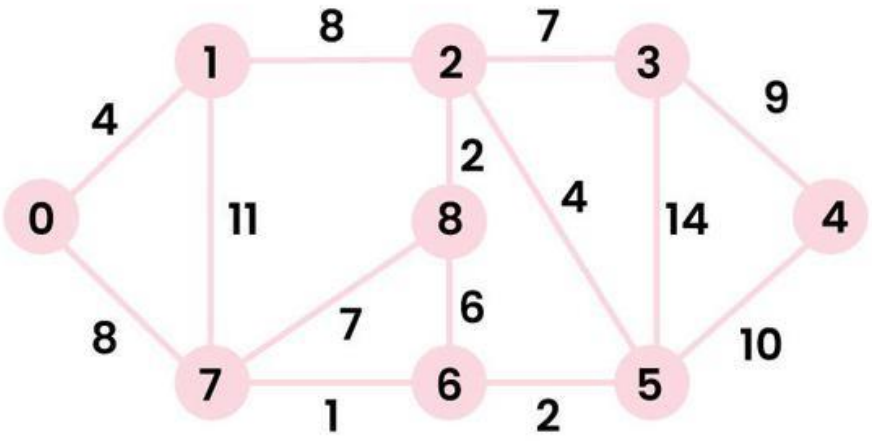
How to find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm

Given a weighted graph and a source vertex in the graph, find the **shortest paths** from the source to all the other vertices in the given graph.
Note: The given graph does not contain any negative edge.
Algorithm :

- Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE** . Assign the distance value as 0 for the source vertex so that it is picked first.
- While **sptSet** doesn't include all vertices
    - Pick a vertex **u** that is not there in **sptSet** and has a minimum distance value.
    - Include u to **sptSet** .
    - Then update the distance value of all adjacent vertices of **u** .
        - To update the distance values, iterate through all adjacent vertices.
        - For every adjacent vertex **v,** if the sum of the distance value of **u** (from source) and weight of edge **u-v** , is less than the distance value of **v** , then update the distance value of **v** .

Note: We use a boolean array **sptSet[]** to represent the set of vertices included in **SPT** . If a value **sptSet[v]** is true, then vertex v is included in **SPT** , otherwise not. Array **dist[]** is used to store the shortest distance values of all vertices.

Examples:
Input: *src = 0, the graph is shown below.*



Working of Dijkstra's Algorithm

Output: *0 4 12 19 21 11 9 8 14*
Explanation: *The distance from 0 to 1 = 4.*
The minimum distance from 0 to 2 = 12. 0->1->2
The minimum distance from 0 to 3 = 19. 0->1->2->3
The minimum distance from 0 to 4 = 21. 0->7->6->5->4
The minimum distance from 0 to 5 = 11. 0->7->6->5
The minimum distance from 0 to 6 = 9. 0->7->6
The minimum distance from 0 to 7 = 8. 0->7
The minimum distance from 0 to 8 = 14. 0->1->2->8

Bellman–Ford Algorithm

Given a **weighted** graph with **V** vertices and **E** edges, and a source vertex **src**, find the **shortest path** from the source vertex to all vertices in the given graph. If a vertex cannot be reached from source vertex, mark its distance as 10^8.
Note: If a graph contains negative weight cycle, return -1.
Negative weight cycle:
A negative weight cycle is a cycle in a graph, whose sum of edge weights is negative. If you traverse the cycle, the total weight accumulated would be less than zero.

In the presence of negative weight cycle in the graph, the shortest path **doesn't exist** because with each traversal of the cycle shortest path keeps **decreasing**.

Limitation of Dijkstra's Algorithm:

Since, we need to find the **single source shortest path**, we might initially think of using Dijkstra Algo. However, Dijkstra is not suitable when the graph consists of **negative edges**. The reason is, it **doesn't revisit** those nodes which have already been marked as visited. If a shorter path exists through a longer route with negative edges, Dijkstra's algorithm will fail to handle it.
Bellman-Ford Algorithm – O(V*E) Time and O(V) Space
Bellman-Ford *is a* ***single source*** *shortest path algorithm. It effectively works in the cases of negative edges and is able to detect negative cycles as well. It works on the principle of* ***relaxation of the edges****.*

Principle of Relaxation of Edges

- Relaxation means **updating** the shortest distance to a node if a shorter path is found through another node. For an edge **(u, v)** with weight **w**:

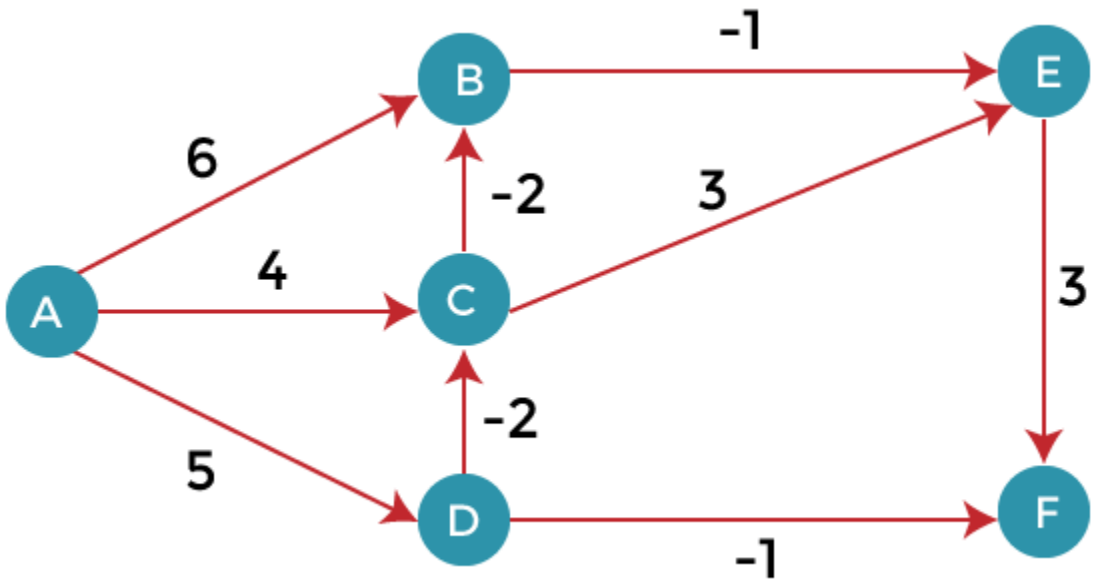
- If going through u gives a shorter path to v from the source node (i.e., **distance[v] > distance[u] + w**), we update the distance[v] as distance[u] + w.

- In the bellman-ford algorithm, this process is repeated **(V – 1)** times for all the edges.

Rule of this algorithm

1. We will go on relaxing all the edges (n - 1) times where,
2. n = number of vertices

Consider the below graph:



As we can observe in the above graph that some of the weights are negative. The above graph contains 6 vertices so we will go on relaxing till the 5 vertices. Here, we will relax all the edges 5 times. The loop will iterate 5 times to get the correct answer. If the loop is iterated more than 5 times then also the answer will be the same, i.e., there would be no change in the distance between the vertices.

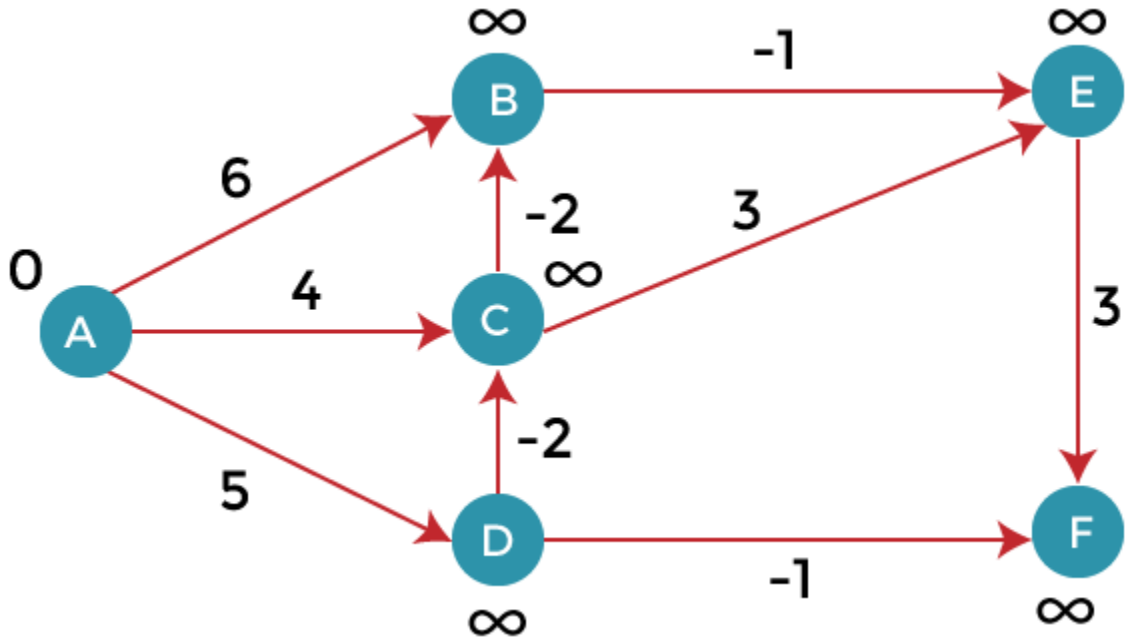
Relaxing means:

1. If (d(u) + c(u , v) < d(v))
2. d(v) = d(u) + c(u , v)

To find the shortest path of the above graph, the first step is note down all the edges which are given below:

(A, B), (A, C), (A, D), (B, E), (C, E), (D, C), (D, F), (E, F), (C, B)

Let's consider the source vertex as 'A'; therefore, the distance value at vertex A is 0 and the distance value at all the other vertices as infinity shown as below:



Since the graph has six vertices so it will have five iterations.

First iteration

Consider the edge (A, B). Denote vertex 'A' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:
d(u) = 0
d(v) = ∞
c(u , v) = 6
Since (0 + 6) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 0 + 6 = 6
Therefore, the distance of vertex B is 6.
Consider the edge (A, C). Denote vertex 'A' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:
d(u) = 0
d(v) = ∞
c(u , v) = 4
Since (0 + 4) is less than ∞, so update

1. d(v) = d(u) + c(u , v)

d(v) = 0 + 4 = 4

Therefore, the distance of vertex C is 4.

Consider the edge (A, D). Denote vertex 'A' as 'u' and vertex 'D' as 'v'. Now use the relaxing formula:
d(u) = 0
d(v) = ∞
c(u , v) = 5
Since (0 + 5) is less than ∞, so update

1. d(v) = d(u) + c(u , v)
   d(v) = 0 + 5 = 5

Therefore, the distance of vertex D is 5.
Consider the edge (B, E). Denote vertex 'B' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:
d(u) = 6
d(v) = ∞
c(u , v) = -1
Since (6 - 1) is less than ∞, so update

1. d(v) = d(u) + c(u , v)
   d(v) = 6 - 1= 5

Therefore, the distance of vertex E is 5.
Consider the edge (C, E). Denote vertex 'C' as 'u' and vertex 'E' as 'v'. Now use the relaxing formula:
d(u) = 4
d(v) = 5
c(u , v) = 3
Since (4 + 3) is greater than 5, so there will be no updation. The value at vertex E is 5.
Consider the edge (D, C). Denote vertex 'D' as 'u' and vertex 'C' as 'v'. Now use the relaxing formula:
d(u) = 5
d(v) = 4
c(u , v) = -2
Since (5 -2) is less than 4, so update

1. d(v) = d(u) + c(u , v)
   d(v) = 5 - 2 = 3

Therefore, the distance of vertex C is 3.
Consider the edge (D, F). Denote vertex 'D' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:
d(u) = 5
d(v) = ∞
c(u , v) = -1
Since (5 -1) is less than ∞, so update

1. d(v) = d(u) + c(u , v)
   d(v) = 5 - 1 = 4

Therefore, the distance of vertex F is 4.

Consider the edge (E, F). Denote vertex 'E' as 'u' and vertex 'F' as 'v'. Now use the relaxing formula:

d(u) = 5

d(v) = ∞
c(u , v) = 3
Since (5 + 3) is greater than 4, so there would be no updation on the distance value of vertex F.
Consider the edge (C, B). Denote vertex 'C' as 'u' and vertex 'B' as 'v'. Now use the relaxing formula:
d(u) = 3
d(v) = 6
c(u , v) = -2
Since (3 - 2) is less than 6, so update

1. d(v) = d(u) + c(u , v)
   d(v) = 3 - 2 = 1
   Therefore, the distance of vertex B is 1.
   Now the first iteration is completed. We move to the second iteration.
   Similarly we follows till n-1 times.

For the following topics:

Bionomial Heap
Binomial Heap (Data Structures) - javatpoint

Fibonacci Heap
Fibonacci Heap - javatpoint

Maximum Flow networks
Ford-Fulkerson Algorithm for Maximum Flow Problem - GeeksforGeeks

Topological sort

Topological sort | Practice | GeeksforGeeks