Bellman-Ford Algorithm

- A **Bellman-Ford** algorithm is also guaranteed to find the shortest path in a graph
- Bellman-Ford is slower than **Dijkstra's algorithm**.
- handling graphs with **negative edge weights**.
- detecting **negative cycles**.

The idea behind Bellman Ford Algorithm:

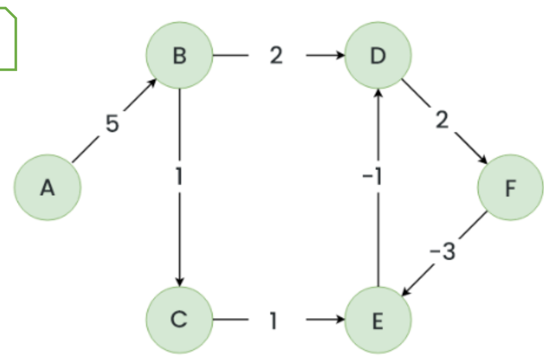
- starts with a single source.
- calculates the distance to each node.
- The distance is initially unknown and assumed to be infinite.
- the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on "Principle of Relaxation".

Principle of Relaxation of Edges for Bellman-Ford:

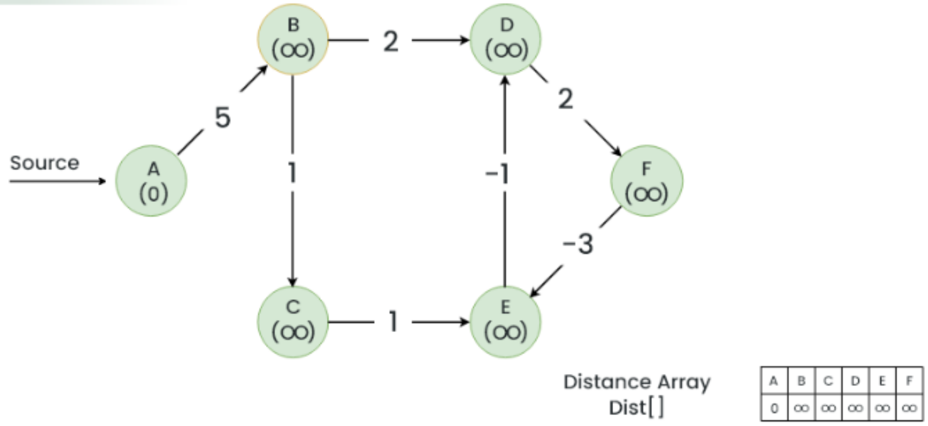
- if we relax the edges **N** times, and there is any change in the shortest distance of any node between the **N-1th** and **Nth** relaxation than a negative cycle exists, otherwise not exist.

Working of Bellman-Ford Algorithm to Detect the Negative cycle in the graph

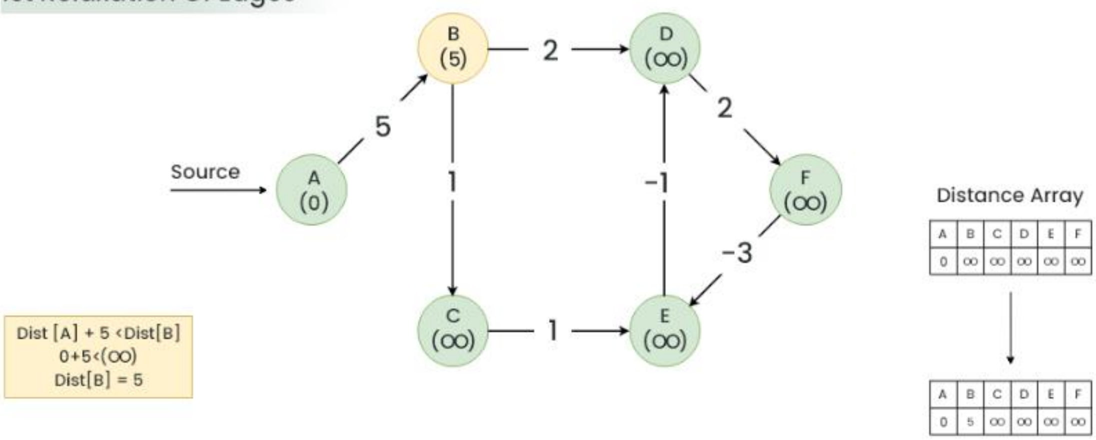
Initial Graph



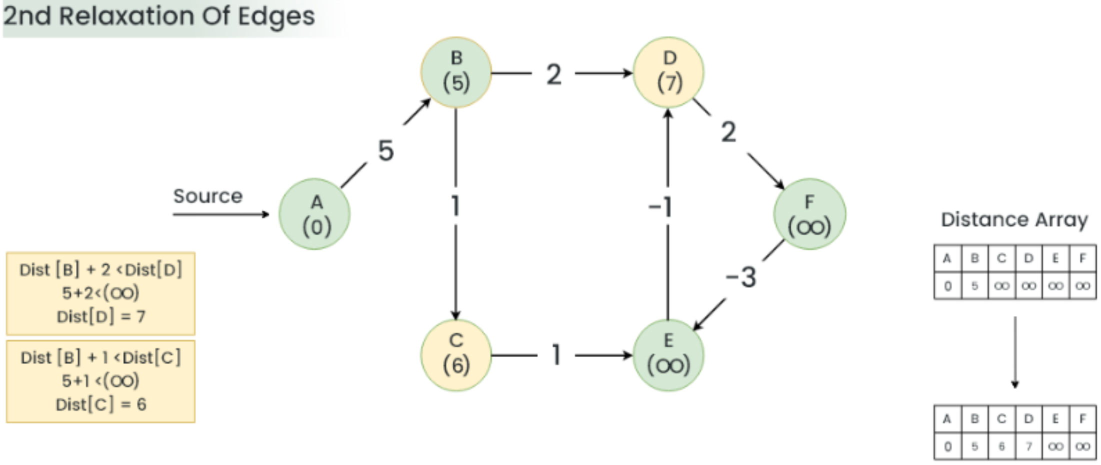
Initialize The Distance Array



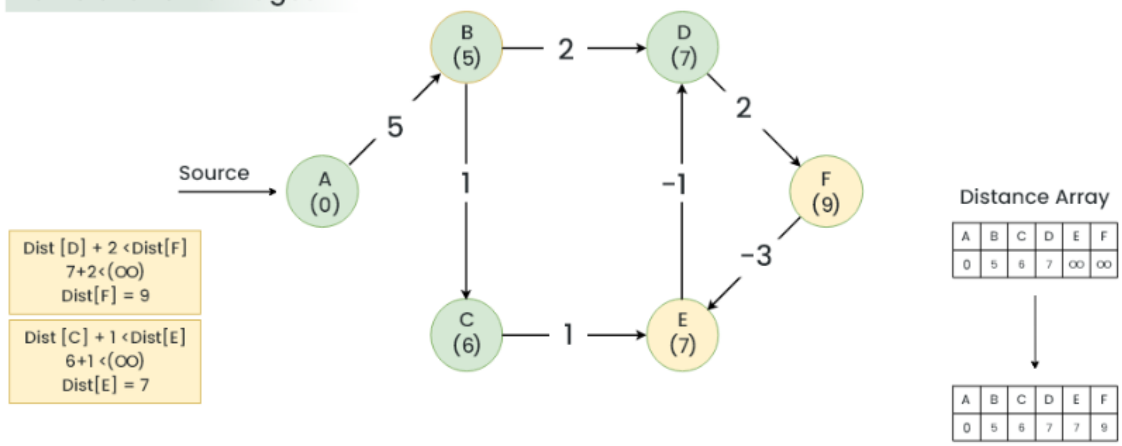
1st Relaxation Of Edges



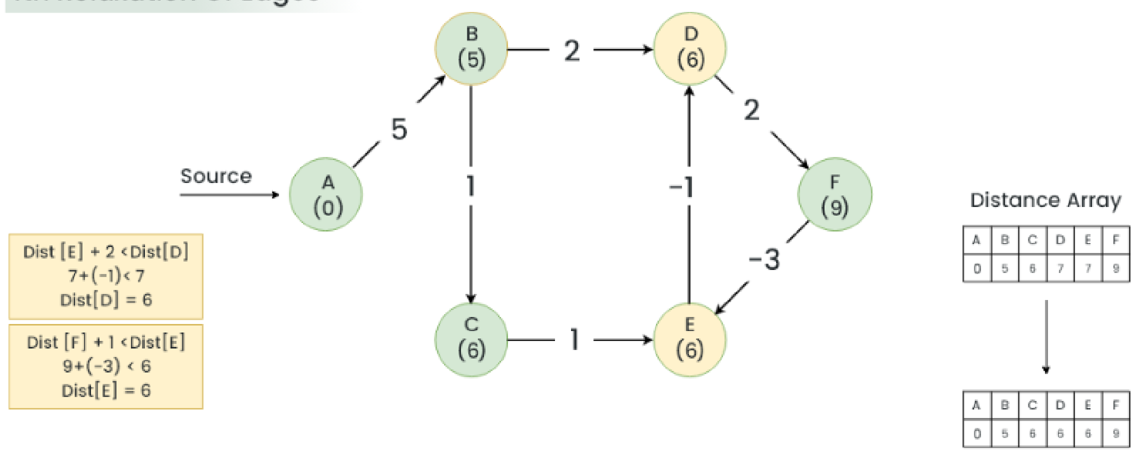
2nd Relaxation Of Edges



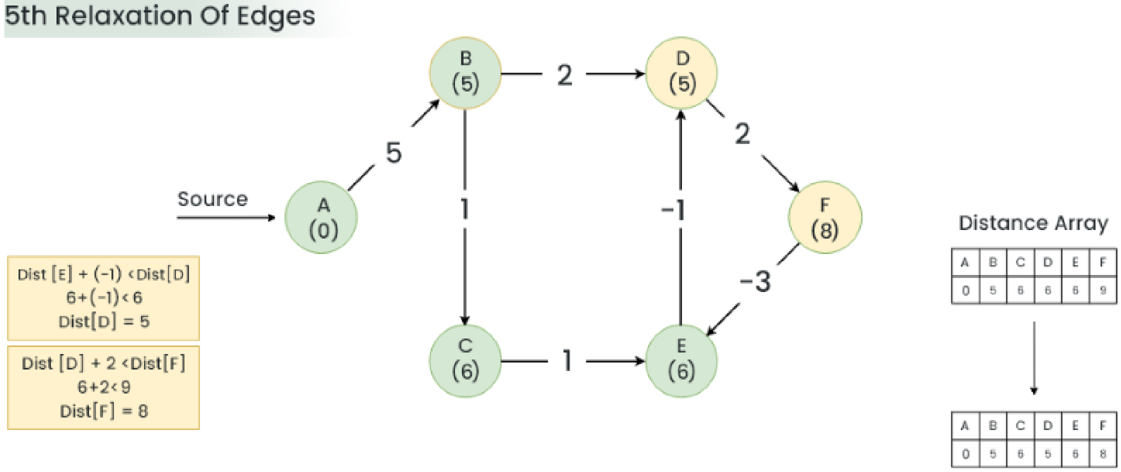
3rd Relaxation Of Edges

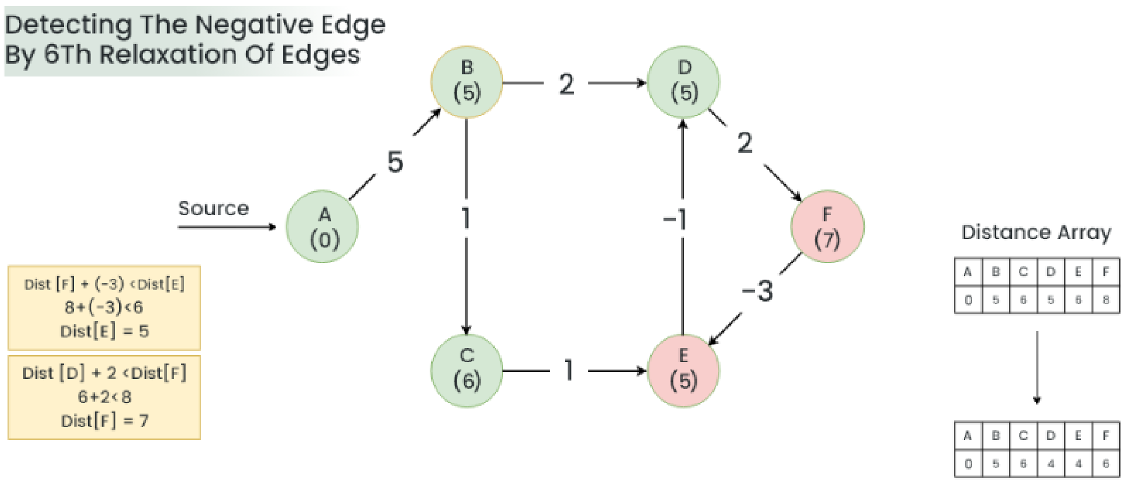


4th Relaxation Of Edges



5th Relaxation Of Edges





***Result:** A negative cycle (D->F->E) exists in the graph.*

Algorithm

- Initialize distance array **dist[]** for each vertex 'v' as **dist[v] = INFINITY**
- Assume any vertex (let's say '0') as source and assign **dist = 0.**
- Relax all the edges(u,v,weight) N-1 times as per the below condition:
  - **dist[v] = minimum(dist[v], distance[u] + weight)**
- Now, Relax all the edges one more time i.e. the Nth time and based on the below two cases we can detect the negative cycle:
  - **Case 1** (Negative cycle exists): For any edge(u, v, weight), if dist[u] + weight < dist[v]
  - **Case 2** (No Negative cycle) : case 1 fails for all the edges.

Detect Cycle in a Directed Graph

- It is based on the idea that there is a cycle in a graph only if there is a back edge present in the graph.
- To find cycle in a directed graph we can use the Depth First Traversal (**DFS**) technique.
- If during recursion, we reach a node that is already in the **recursion stack**, there is a cycle present in the graph.

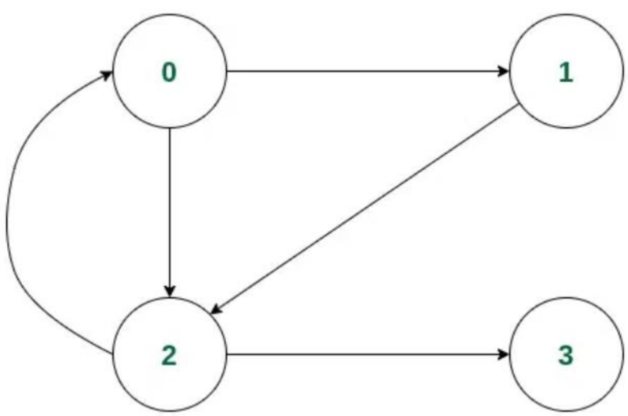
steps to Implement the idea

Create a recursive **DFS** function that has the following parameters – **current vertex**, **visited array**, and **recursion stack**.

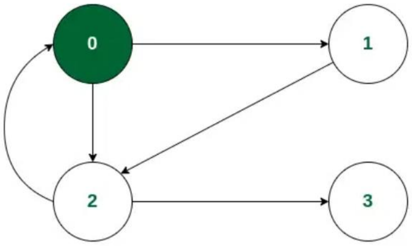
Mark the **current node as visited** and also mark the index in the recursion stack.

Iterate a loop for all the vertices and for each vertex, call the recursive function if it is not yet visited

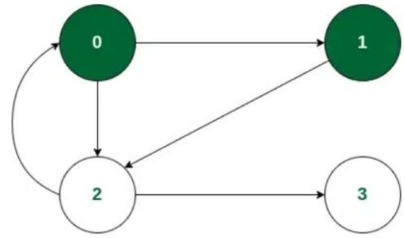
- In each recursion call, Find all the adjacent vertices of the current vertex which are not visited:
  - If an adjacent vertex is already marked in the recursion stack then return true.
  - Otherwise, call the recursive function for that adjacent vertex.



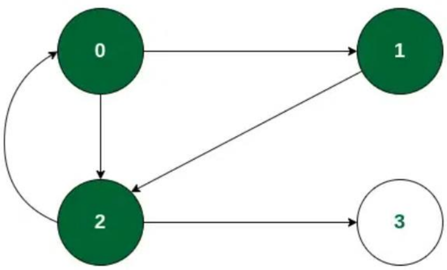
Example of a Directed Graph



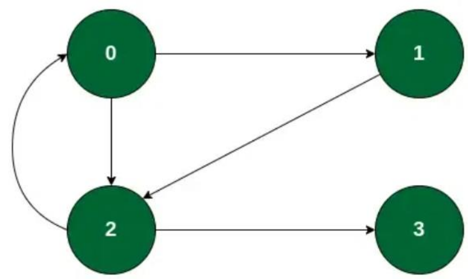
	0	1	2	3
visited	true	false	false	false
recStack	true	false	false	false



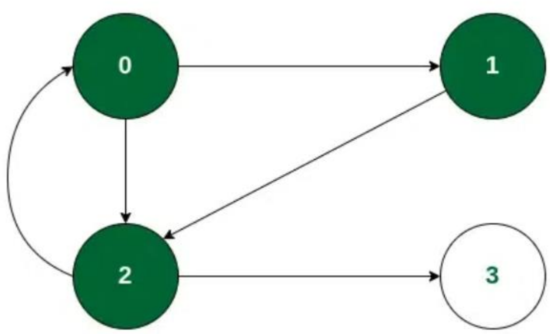
	0	1	2	3
visited	true	true	false	false
recStack	true	true	false	false



	0	1	2	3
visited	true	true	true	false
recStack	true	true	true	false



	0	1	2	3
visited	true	true	true	true
recStack	true	true	true	true



	0	1	2	3
visited	true	true	true	true
recStack	true	true	true	false