Design and Analysis of Algorithms

- ***Course Code***: BCSE204L
- ***Course Type***: Theory (ETH)
- ***Slot***: A1+TA1 & & A2+TA2
- ***Class ID***: VL2023240500901
  VL2023240500902

A1+TA1

Day	Start	End
Monday	08:00	08:50
Wednesday	09:00	09:50
Friday	10:00	10:50

A2+TA2

Day	Start	End
Monday	14:00	14:50
Wednesday	15:00	15:50
Friday	16:00	16:50

Syllabus- Module 4

Module:4	Graph Algorithms	6 hours

All pair shortest path: Bellman Ford Algorithm, Floyd-Warshall Algorithm Network Flows: Flow Networks, Maximum Flows: Ford-Fulkerson, Edmond-Karp, Push Re-label Algorithm – Application of Max Flow to maximum matching problem

Graph shortest path (SP) algorithms

- Graph shortest path (SP) algorithms are a class of algorithms designed to find the shortest path between two nodes (vertices) in a graph.
- The "shortest path" refers to the path with the minimum total weight or cost among all possible paths between two nodes.
- **Application**: These algorithms are widely used in various applications such as network routing, transportation planning, and social network analysis.

There are two main types of shortest path algorithms:

- **Single Source Shortest Path (SSSP) Algorithms**:
  - These algorithms find the shortest path from a single source node to all other nodes in the graph.
- **All Pair Source Shortest Path (APSP) Algorithms**:
  - These algorithms find the shortest path between all pairs of nodes in the graph.

Graph shortest path (SP) algorithms

Single Source Shortest Path (SSSP) Algorithms:

- **Purpose**: Finding the shortest path from a single source node to all other nodes in the graph.
- **Key Algorithms**:
  - **Bellman-Ford Algorithm**: Can handle negative weight edges and is based on dynamic programming principles.
  - **Dijkstra's Algorithm**: Suitable for graphs with non-negative edge weights and is based on a greedy method approach. It significantly improves the runtime compared to Bellman-Ford for non-negative weights.

All Pair Source Shortest Path (APSP) Algorithm:

- **Purpose**: Finding the shortest path between all pairs of nodes in the graph.
- **Common Algorithm**:
  - Floyd-Warshall Algorithm is a popular choice for solving APSP problem. It's based on dynamic programming and works efficiently for dense graphs.

Syllabus- Module 4

Module:4	Graph Algorithms	6 hours

All pair shortest path: **Bellman Ford Algorithm**, Floyd-Warshall Algorithm Network Flows: Flow Networks, Maximum Flows: Ford-Fulkerson, Edmond-Karp, Push Re-label Algorithm – Application of Max Flow to maximum matching problem

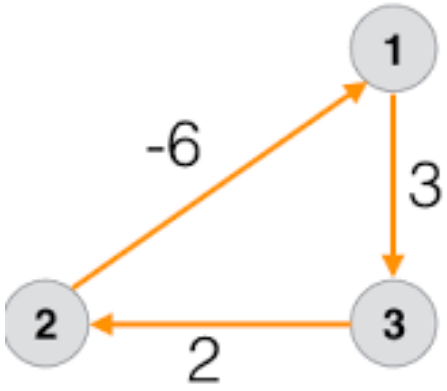
Single-source shortest path algorithms: Bellman-Ford algorithm

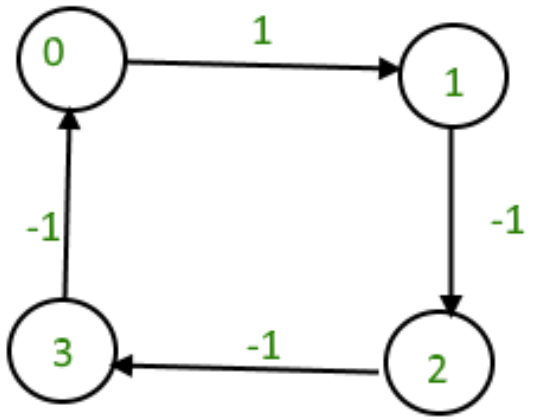
- The Bellman-Ford algorithm is a single-source shortest path algorithm that accommodates both positive and negative edge weights, although unable to handle graphs containing negative weight cycles. It was introduced by Richard Bellman and Lester Ford in the 1950s.

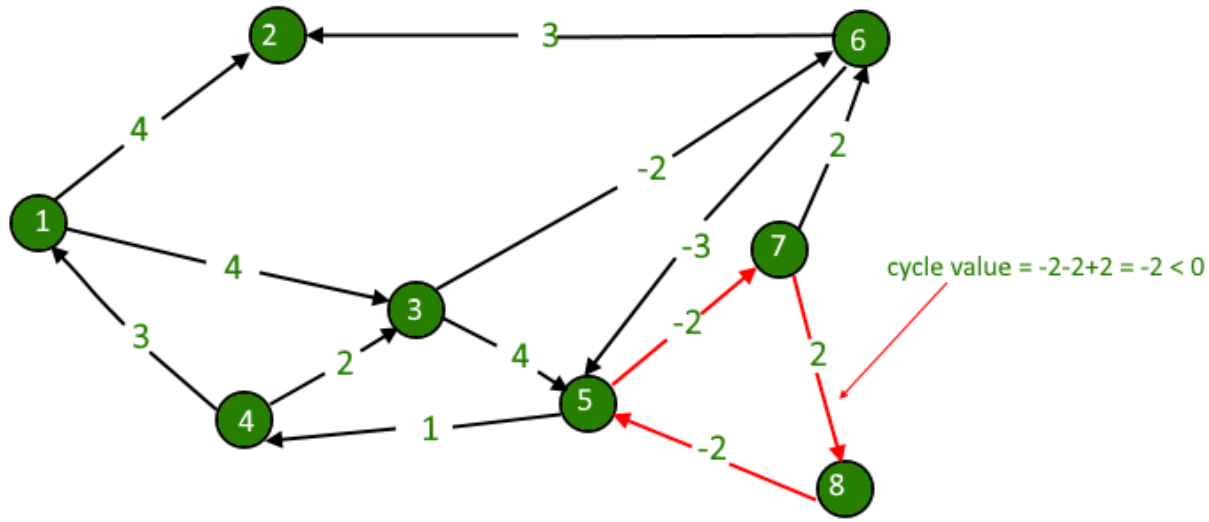
The description and implementation of Bellman and Ford's algorithm for the single-source shortest-path problem, featuring general edge weights, can be found in "Dynamic Programming" by R. E. Bellman, Princeton University Press, 1957.

Bellman-ford algorithm (BFA)

- A negative cycle occurs when the sum of weights along a cycle becomes negative.
- The Bellman-Ford algorithm possesses the unique capability to detect the presence of such negative edge cycles.
- Whenever we give some graph as input to BFA, it has a capacity to say
  - No (can't find the SP, because of negative edge cycle) or
  - Yes (can find out the SP, because there are no negative edge cycle. And it can even give the shortest path weights alsc`







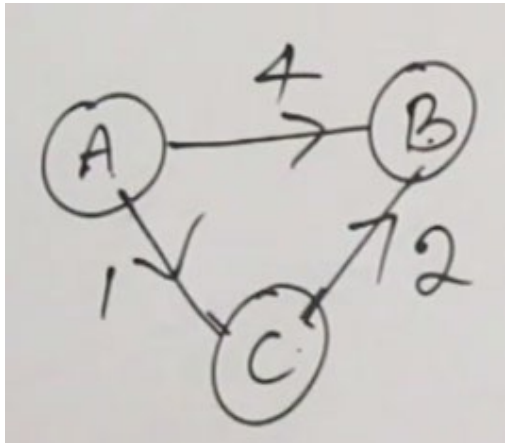
Bellman-ford algorithm
(How to check Negative edge cycle)

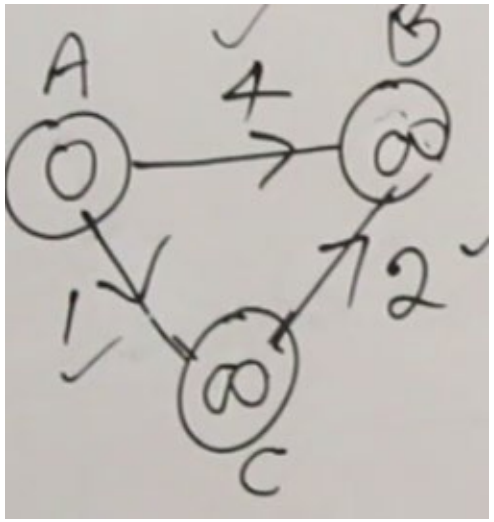
- A significant feature of the Bellman-Ford algorithm is its ability to identify negative edge cycles.
- Notably, in a graph with N nodes, the shortest path will never contain more than N-1 edges.
- Therefore, if a graph comprises N nodes and N edges, a cycle is inevitable (exapted). The Bellman-Ford algorithm operates based on this principle.

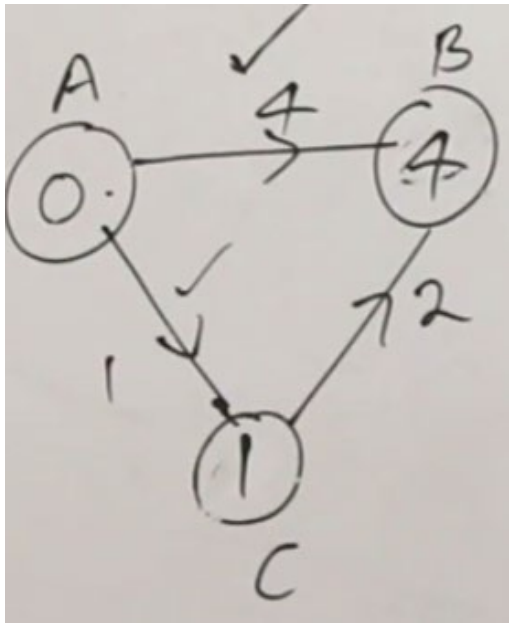
Bellman-ford algorithm

(How to check Negative edge cycle)

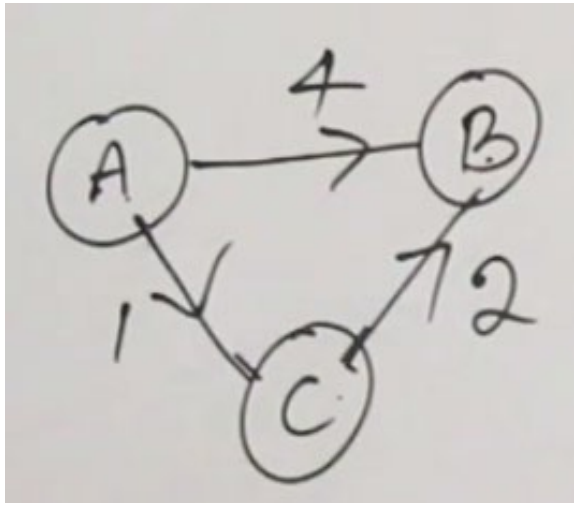
- When you relax an edge for the first time, the algorithm computes the shortest paths with a maximum of one edge in that path.
- Similarly, after relaxing edges twice, the shortest path length may contain a maximum of two edges, and so forth.
- Following N-1 relaxations, where the length of the shortest path can be at most N-1 edges, additional relaxation may be necessary.
- If, after N-1 relaxations, further relaxation results in improved values for any vertices, it indicates the presence of a negative edge cycle.
- This occurs because after considering N-1 edges to determine the shortest path, any subsequent relaxation may introduce loops into the path. If these loops offer superior values, it suggests the existence of a negative edge cycle.

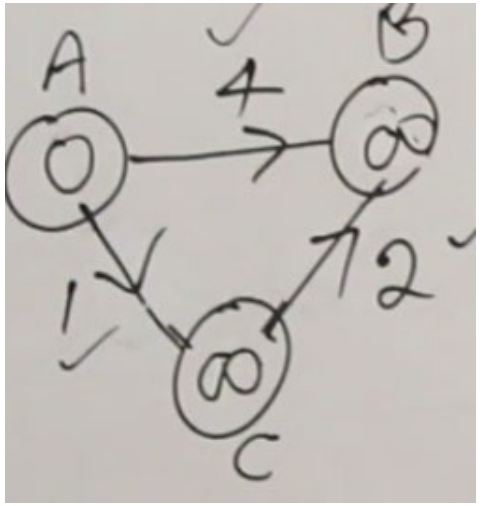


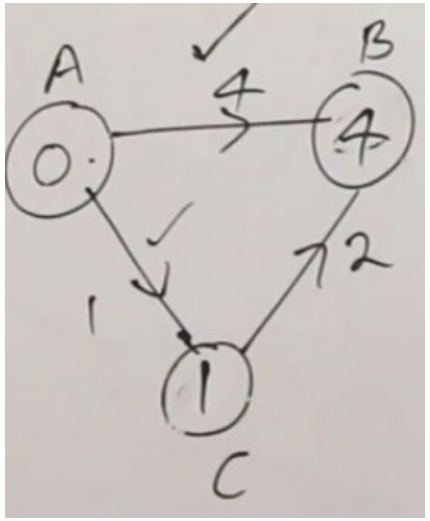


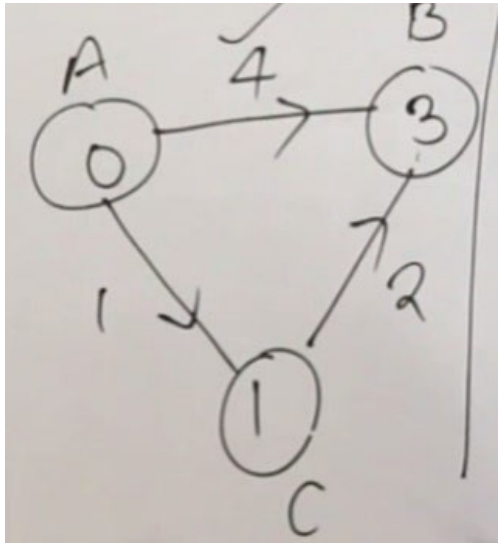


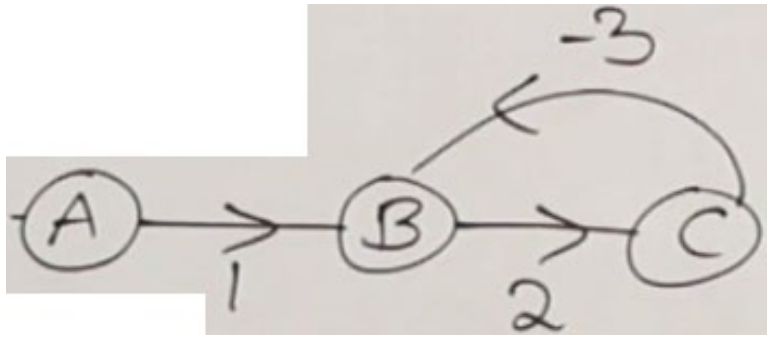
When you relax the edge one time (i.e., the first time), it finds the length of the shortest paths, with a maximum of one edge in that path.


A
B
C
4
1
2

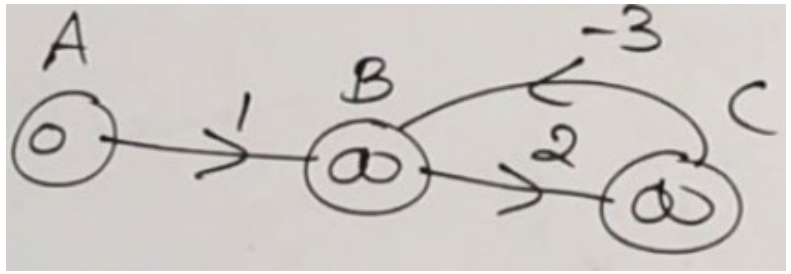

A
B
C
0
∞
∞
4
1
2

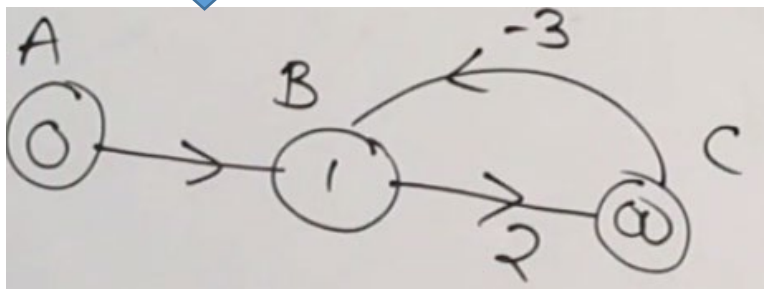

A
B
C
0
4
1
4
1
2

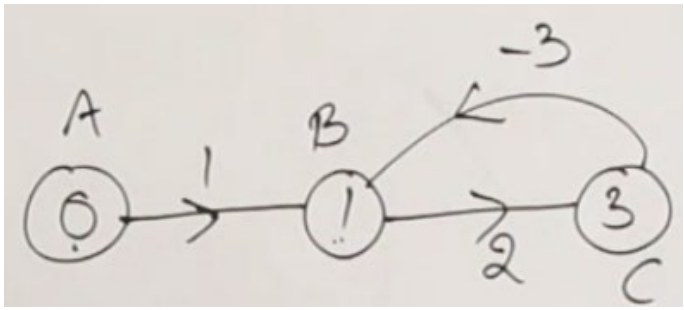

A
B
C
0
3
1
4
1
2

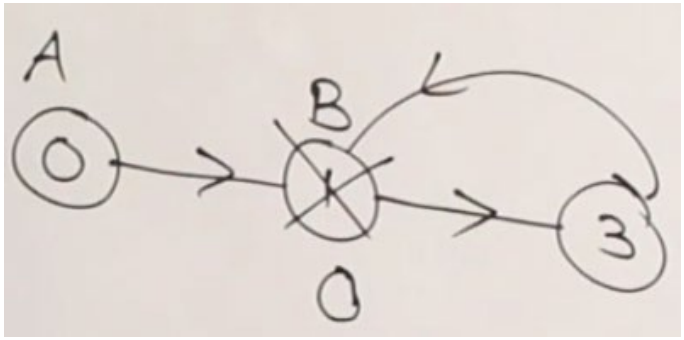


The length of the shortest path to the vertices, in which the path contained only one edge.



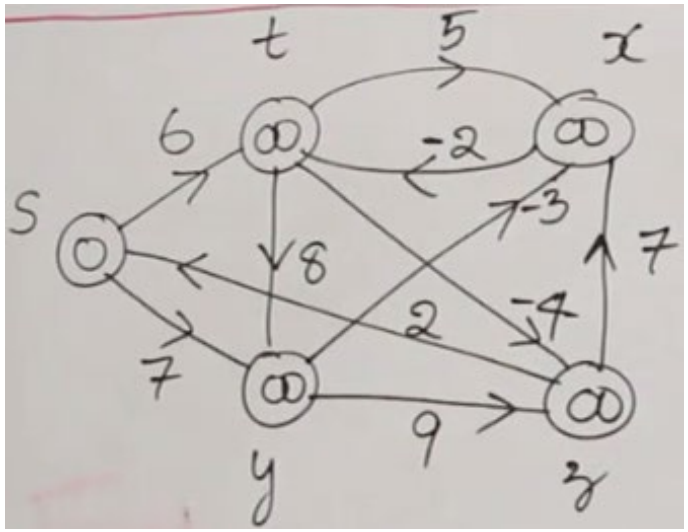


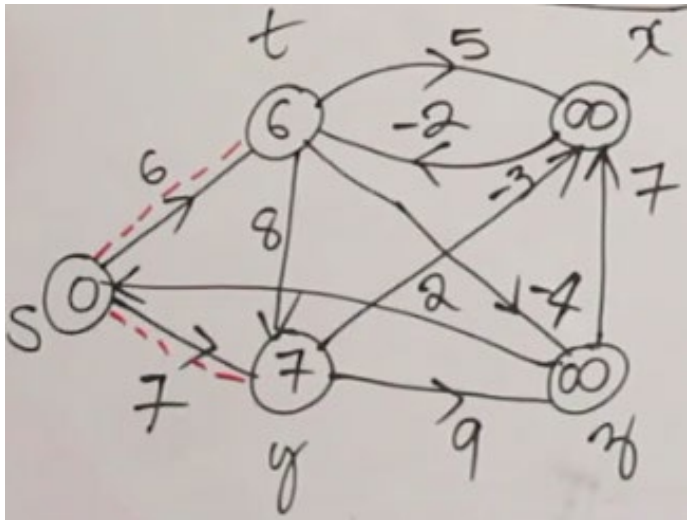


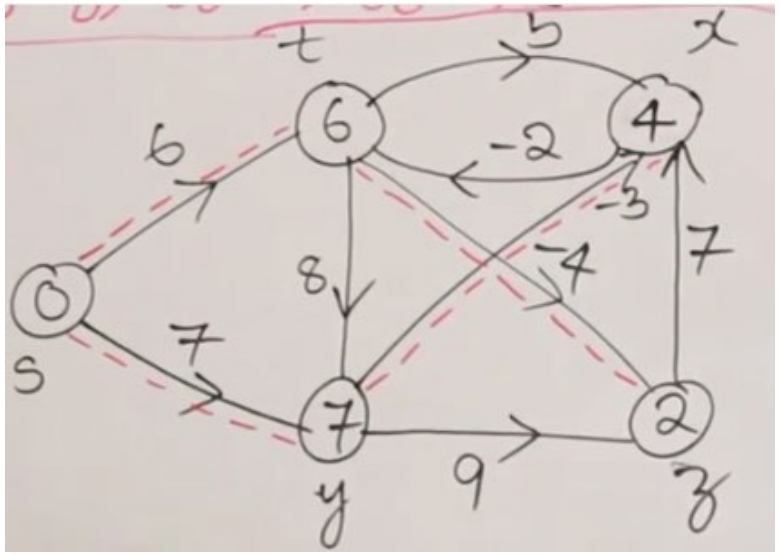


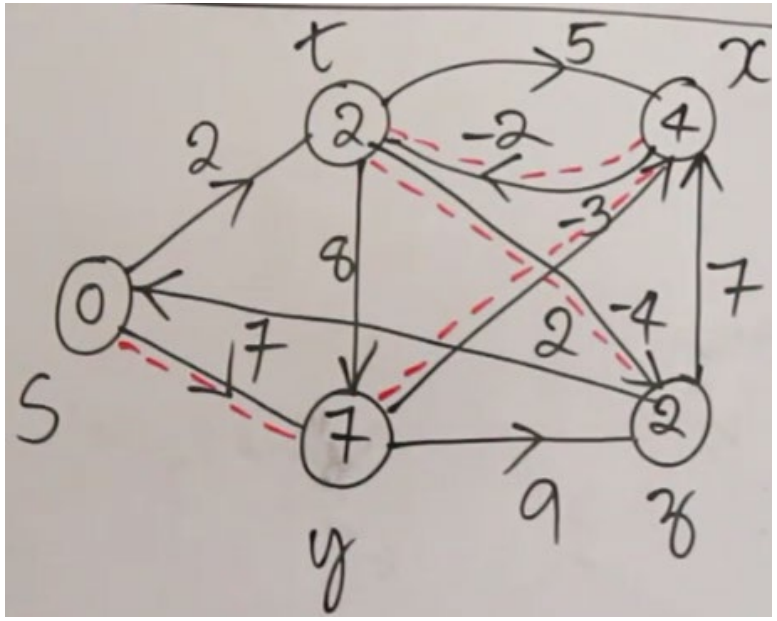
The Bellman-Ford algorithm typically runs relaxation steps for $n-1$ iterations, where n is the number of vertices in the graph. However, it may perform an additional relaxation step to check for the presence of negative weight cycles. Which are crucial to consider for accurately determining shortest paths in the presence of such cycles.

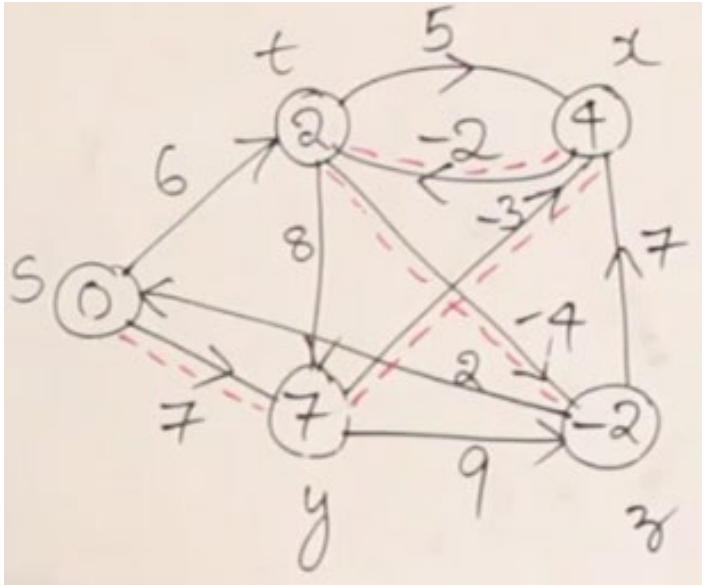
If, after $n-1$ iterations, further relaxation results in improved values for any vertices, it implies that there is a negative edge cycle.



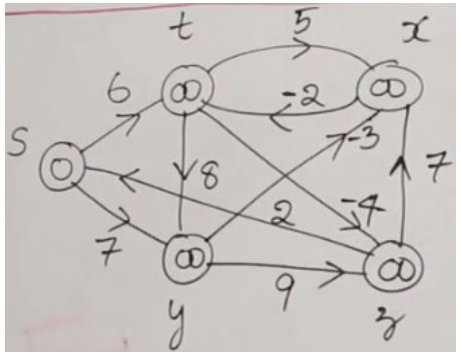


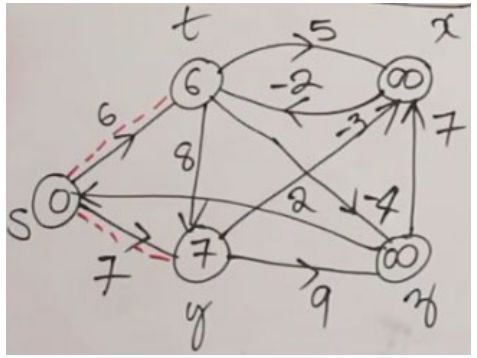


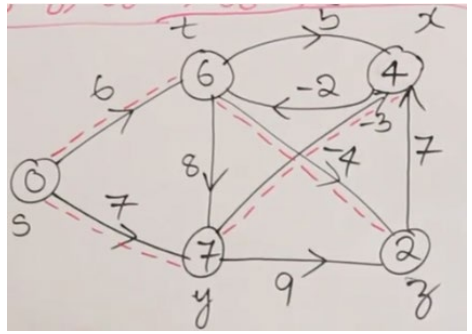


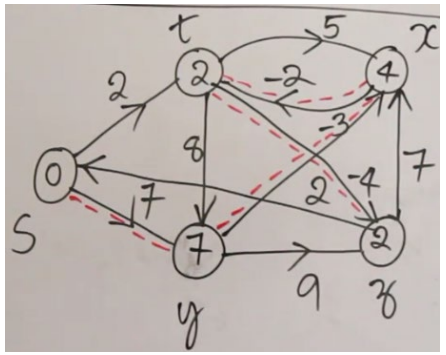


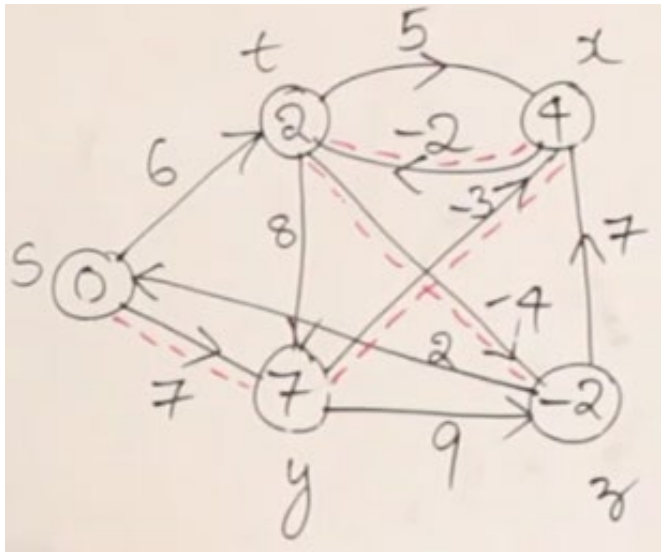
	s	t	x	y	z
s	0	6	inf	7	inf
t	inf	0	5	8	-4
x	inf	-2	0	inf	inf
y	inf	inf	2	0	9
z	2	inf	7	inf	0











	s	t	x	y	z
s	0	6	inf	7	inf
t	inf	0	5	8	-4
x	inf	-2	0	inf	inf
y	inf	inf	2	0	9
z	2	inf	7	inf	0

```
Enter the number of vertices: 5
Enter the number of edges: 9
Enter edge 1 (source, destination, weight): s t 6
Enter edge 2 (source, destination, weight): s y 7
Enter edge 3 (source, destination, weight): t x 5
Enter edge 4 (source, destination, weight): t z -4
Enter edge 5 (source, destination, weight): x t -2
Enter edge 6 (source, destination, weight): y x 2
Enter edge 7 (source, destination, weight): y z 9
Enter edge 8 (source, destination, weight): z s 2
Enter edge 9 (source, destination, weight): z x 7
Enter the starting node: s
Vertex Distance from Source   Path
s              0              s
t              2              s -> y -> x -> t
x              4              s -> y -> x
y              7              s -> y
z              2              s -> z
```

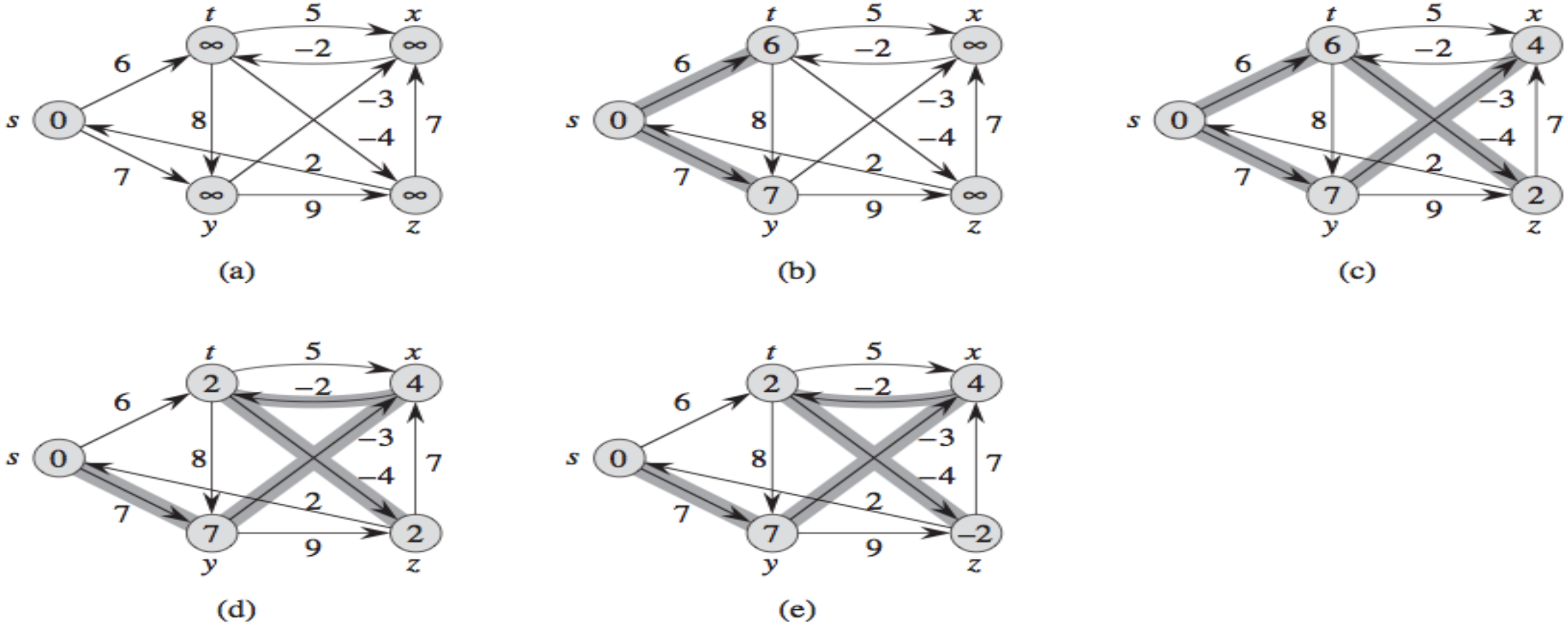
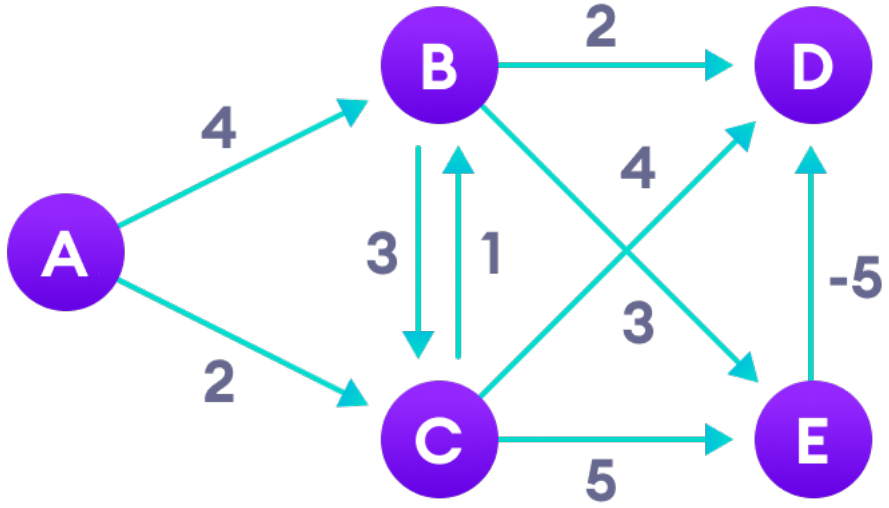


Figure 24.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

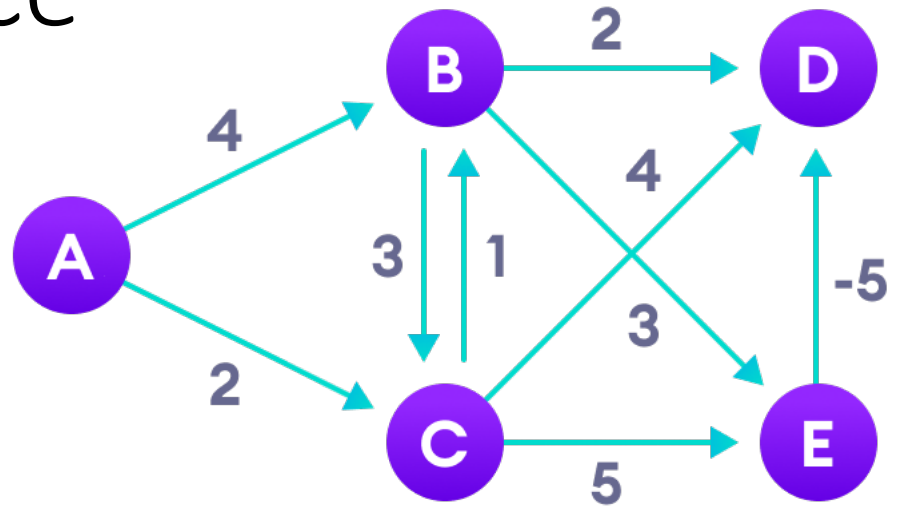
```
BELLMAN-FORD(G, w, s)
{
  Initialize single source(G, s)
  for i := 1 to |G.V| - 1
      for each edge (u,v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u,v) ∈ G.E
      if (v.d > u.d + w(u,v))
          return FALSE
  return TRUE
}
```

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n - 1 do
8           for each u such that u ≠ v and u has
9                       at least one incoming edge do
10                  for each ⟨i, u⟩ in the graph do
11                      if dist[u] > dist[i] + cost[i, u] then
12                          dist[u] := dist[i] + cost[i, u];
13  }
```

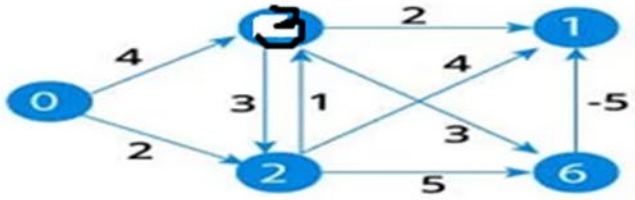
Pratice



Pratice



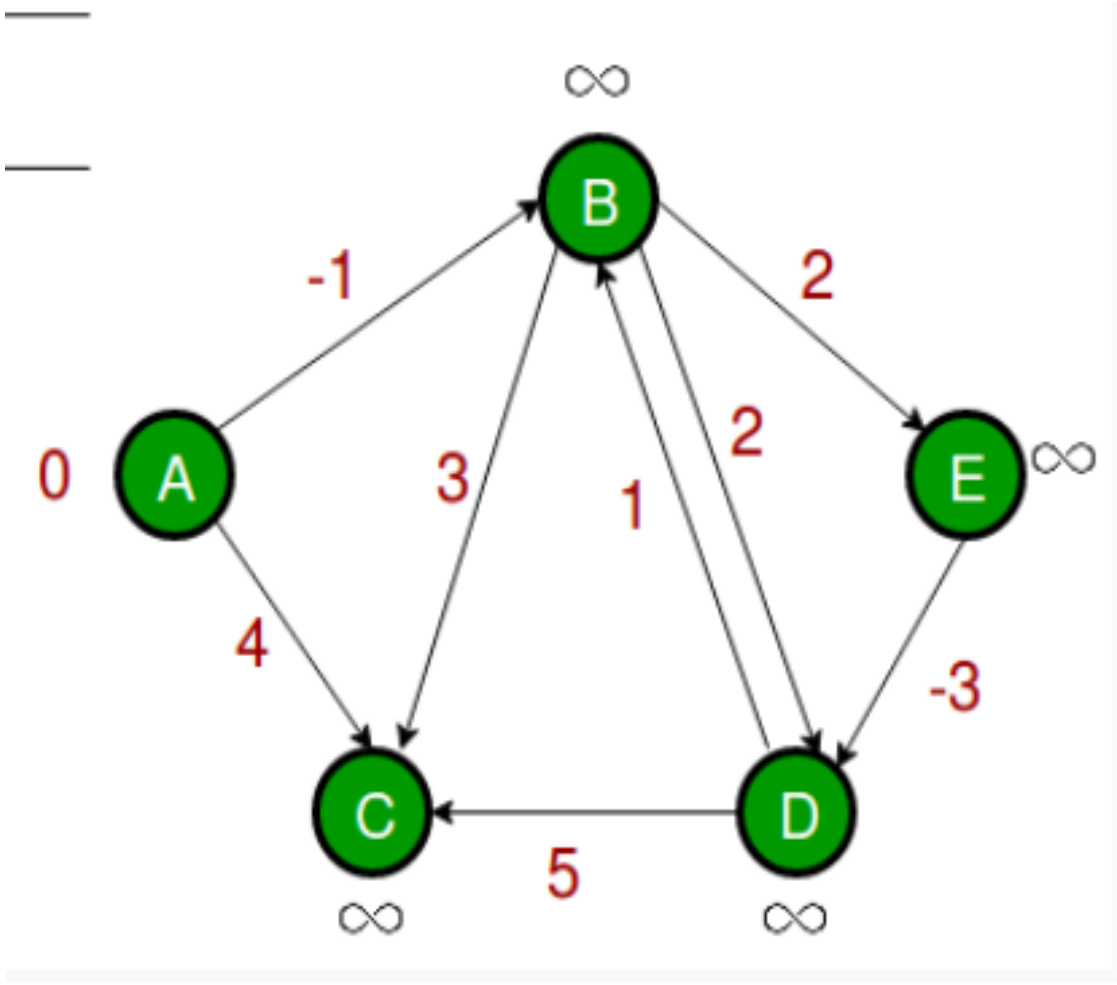
Notice how the vertex at the top right corner had its path length adjusted



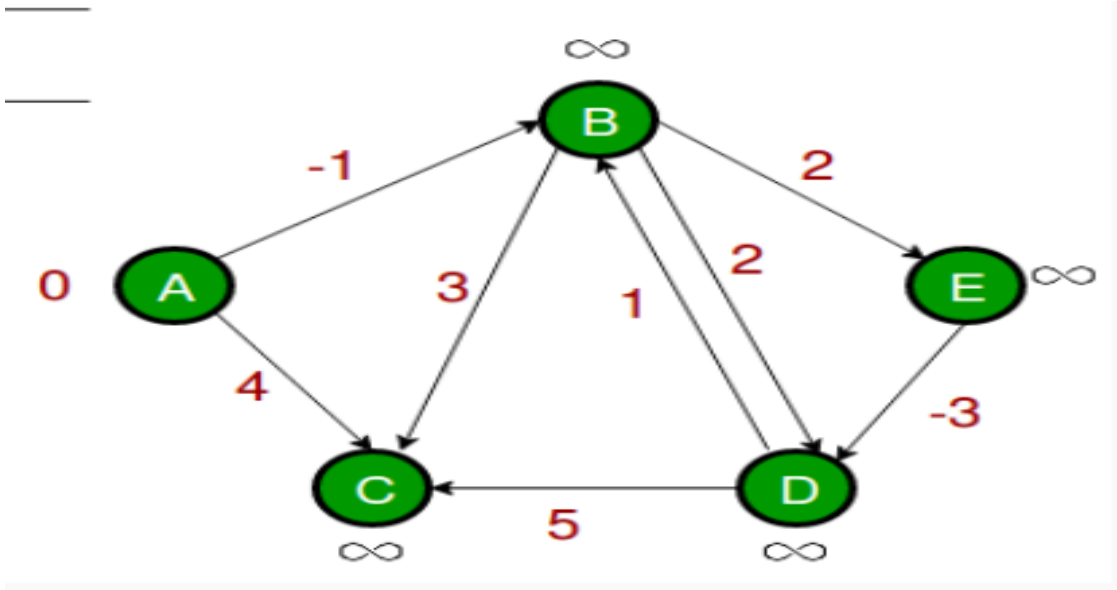
After all the vertices have their path lengths, we check if a negative cycle is present.

A	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

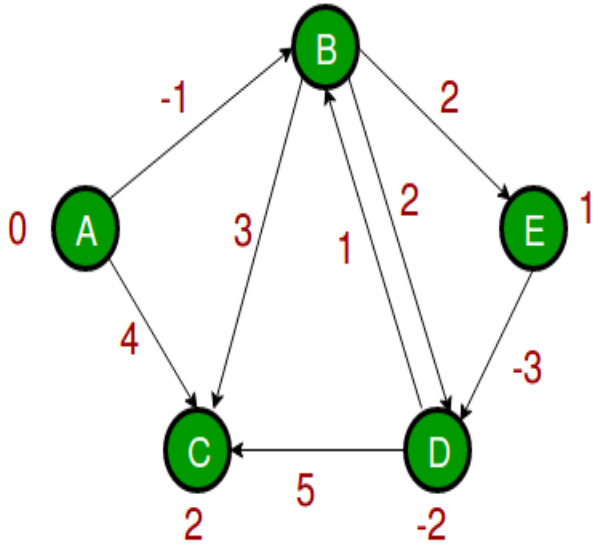
Pratice



Pratice



A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	1	1
0	-1	2	-2	1



Applications

- Used for distance-routing protocol helping in routing the data packets on the network
- Used in internet gateway routing protocol
- Used in routing information protocol

Any Question?

Thank You!

FOR YOUR ATTENTION