Bellman Ford

➢ What is Bellman Ford?

Bellman–Ford is a shortest path algorithm that finds the minimum distance from a single source node to all other nodes in a weighted graph.
It works even when:

- Edges have negative weights
- The graph contains cycles

However, it cannot handle graphs with negative weight cycles that are reachable from the source — but it can detect them.

➢ When Should You Use Bellman–Ford?

You choose Bellman–Ford over other algorithms when:

- The graph has negative edge weights
- You need to detect negative cycles
- You don't need extremely fast performance (Bellman–Ford is slower than Dijkstra)

Common applications:

- Currency arbitrage detection (negative cycles represent profit opportunities)
- Routing protocols (e.g., RIP)
- Optimizing paths in networks where weights may decrease

➢ Key Concepts You Need to Know

1. **Weighted Graph**

   Bellman–Ford works on a weighted graph where each edge has a cost (positive or negative).

2. **Relaxation**

   This is the heart of the algorithm. To "relax" an edge (u → v with weight w) means:

   - If the shortest known path to v can be improved by going through u, update it.

3. **Negative Weight Cycle**

   A cycle whose edges sum to a negative value.

Bellman–Ford can detect this by trying one more relaxation step after finishing the main loop.

➢ How Bellman–Ford Works

1. Initialize all distances to infinity, except the source which is set to 0.
2. Repeat the relaxation process (V − 1) times (where V = number of vertices):
   - For every edge, try to update the shortest path.
3. Perform one more relaxation loop:
   - If any distance updates again, it means there is a negative cycle.

- **Pseudocode:**

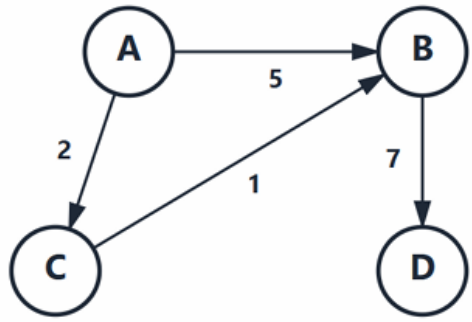
```
BELLMAN-FORD (G,s)

    INITIALIZE SINGLE – SOURCE (G,s)

     for i ← 1 to |V|-1 do
        for each edge (u,v) ∈ G.E do
          RELAX (U, V)
     for each edge (u,v) ∈ E do
           if d[V] > d[U] + W(U, V) then

return FALSE
return TRUE
```

- **Example Graph:**



Bellman-Ford Algorithm from node A:

- ✓ **Initial:**
  Distance: A=0, B=∞, C=∞, D=∞
- ✓ **Iteration 1:**
  A→B: dist[B] = min(∞, 0+5) = 5
  A→C: dist[C] = min(∞, 0+2) = 2
  C→B: dist[B] = min(5, 2+1) = 3
  B→D: dist[D] = min(∞, 3+7) = 10

After Iteration 1: A=0, B=3, C=2, D=10

- ✓ **Iteration 2:**
  A→B: dist[B] = min(3, 0+5) = 3
  A→C: dist[C] = min(2, 0+2) = 2
  C→B: dist[B] = min(3, 2+1) = 3
  B→D: dist[D] = min(10, 3+7) = 10

After Iteration 2: A=0, B=3, C=2, D=10

- ✓ **Iteration 3:**
  No changes (distances stabilized)

Final shortest distances from A:

A→A: 0
A→C: 2
A→B: 3 (via C)
A→D: 10 (via C→B)

No negative cycles detected!

➢ Time & Space Complexity

- Time complexity: O(V × E)
- Space complexity: O(V)

This makes Bellman–Ford slower than Dijkstra, but far more flexible.

➢ What Need to Implement Bellman–Ford?

- A list of edges (edge list is ideal)
- A distance array
- The number of vertices V

Graph representations:

- Edge list (best for Bellman–Ford)
- Adjacency list (usable, but requires converting to edges)

➢ Final Thoughts

Bellman–Ford may not be the fastest, but it is one of the most important graph algorithms due to its ability to handle negative weights and detect negative cycles.
Once you understand Bellman–Ford, you're ready to explore:

- Dijkstra's Algorithm
- Floyd–Warshall (all-pairs shortest path)
- Johnson's Algorithm