

Bellman Ford Algorithm

● MEDIUM

■ Introduction

Bellman-Ford is the answer to Dijkstra's limitation: it handles **NEGATIVE** edge weights. It works by relaxing **ALL** edges $V-1$ times. If a shorter path is still found after $V-1$ iterations, a negative cycle exists.

■ Core Intuition

Why $V-1$ Iterations?

- Any shortest path in a graph has at most $V-1$ edges (visiting each vertex at most once).
- After k iterations, `dist[]` contains shortest paths using at most k edges.
- After $V-1$ iterations: shortest paths using at most $V-1$ edges = all shortest paths.
- If **ANY** edge still relaxes on the V th iteration $\rightarrow$ negative cycle detected.

■ Visual

Bellman-Ford Trace

Graph with negative edge:

```
0 --5-- 1 ---4-- 2
```

```
\ /
```

```
-6 3
```

```
\ /
```

```
3
```

Iter 1: relax all edges $\rightarrow$ dist updates

Iter 2: further relaxation via negative edge

...after $V-1$ iters: stable shortest distances

Negative Cycle Detection:

If `dist[v] > dist[u] + w(u,v)` still holds on V th iteration $\rightarrow$ **NEGATIVE CYCLE**

■ Java Implementation

Java

```
class Solution {
    public int[] bellmanFord(int V, int[][] edges, int src) {
        // edges[i] = {u, v, weight}
```

```

int[] dist = new int[V];

Arrays.fill(dist, Integer.MAX_VALUE);

dist[src] = 0;

// Relax all edges V-1 times
for (int i=1; i < V; i++) {
    for (int[] edge : edges) {
        int u = edge[0], v = edge[1], w = edge[2];
        if (dist[u] != Integer.MAX_VALUE && dist[u]+w < dist[v]) {
            dist[v] = dist[u] + w;
        }
    }
}

// Vth relaxation: detect negative cycle
for (int[] edge : edges) {
    int u = edge[0], v = edge[1], w = edge[2];
    if (dist[u] != Integer.MAX_VALUE && dist[u]+w < dist[v]) {
        System.out.println("Negative cycle detected!");
        return new int[0];
    }
}

return dist;
}

```

■ Time & Space Complexity

Case	Time	Space	Why
Time	$O(V * E)$	$O(V)$	$V-1$ iterations, each relaxing E edges
Space	$O(V)$	$O(V)$	dist[] array only; no heap needed

Dijkstra vs Bellman-Ford Summary

- Dijkstra: $O((V+E)\log V)$, NO negative weights, greedy with priority queue.
- Bellman-Ford: $O(V * E)$, handles negative weights, detects negative cycles.
- Use Dijkstra when all weights non-negative (faster). Use Bellman-Ford with negative weights.

Common Mistakes

Avoid These

- Only doing V-2 iterations — need exactly V-1 to guarantee all shortest paths.
- Not checking `dist[u] != MAX_VALUE` before adding weight — causes integer overflow.
- Confusing with Dijkstra — Bellman-Ford does NOT use a priority queue.

Practice Problems

Case	Time	Space	Why
Medium	LC 787	Cheapest Flights K Stops	Bellman-Ford with K constraint
Medium	LC 743	Network Delay Time	Compare Dijkstra vs Bellman
Hard	LC 1334	Find the City Min Neighbors	All-pairs shortest path

Graphs — Overall Quick Revision

- BFS: Queue, level-by-level, shortest path (unweighted), `visited[]` before enqueue
- DFS: Recursion/Stack, deep-first, `visited[]` before recursion
- Connected Components: run BFS/DFS from each unvisited node
- Cycle (undirected): BFS with parent tracking
- Cycle (directed): DFS with `visited[]` + `inStack[]`
- Topo Sort DFS: post-order push to stack
- Topo Sort Kahn's: `in-degree` → `queue` → `reduce` → BFS
- Dijkstra: Min-Heap, non-negative weights, $O((V+E)\log V)$
- Bellman-Ford: V-1 relaxations, handles negatives, $O(V \cdot E)$
- Grid problems: treat as graph, 4-directional DFS/BFS

Final Mental Model

The Graph Cheat Sheet

- Unweighted shortest path → BFS
- Weighted, non-negative → Dijkstra
- Weighted, with negatives → Bellman-Ford
- Task ordering → Topological Sort
- Connected components → DFS/BFS from each unvisited
- Grid problems → 2D BFS/DFS with direction arrays