Shortest Path Algorithms: Bellman-Ford and Floyd-Warshall

Varun Kumar

July 9, 2025

Contents

List of Algorithms

Python Programs

List of Tables

1. Introduction: Shortest Path Problem

The **Shortest Path Problem** involves finding the path between two nodes in a **weighted graph** such that the sum of the edge weights is minimized.

This problem has many real-world applications:

- Network routing
- Road navigation
- Flight booking systems
- Dependency resolution

Depending on the type of graph and requirements, different algorithms are used:

- **Single-source shortest path:** e.g., Dijkstra, Bellman-Ford
- **All-pairs shortest path:** e.g., Floyd-Warshall
- **Graphs with negative weights:** Bellman-Ford, Floyd-Warshall

2. Bellman-Ford Algorithm

2.1 Purpose

The Bellman-Ford algorithm finds the shortest path from a single source to all other vertices in a weighted graph. Unlike Dijkstra's algorithm, Bellman-Ford works correctly even when the graph contains **negative edge weights**.

2.2 Problem Context

Given a directed graph $G = (V, E)$ with edge weights $w(u, v)$ (which may be negative), and a source vertex $s \in V$, compute the shortest path distance $d(s, v)$ for every $v \in V$. Also detect whether a **negative-weight cycle** is reachable from the source.

2.3 Time Complexity

$$O(V \cdot E)$$

where V is the number of vertices and E is the number of edges.

2.4 Space Complexity

$$O(V)$$

for storing the distance array.

2.5 Key Features

- Works for directed and undirected graphs.
- Handles graphs with negative edge weights.
- Detects and reports negative weight cycles.
- Slower than Dijkstra's algorithm, but more general.

2.6 Step-by-Step Explanation

1. Initialize the distance to all vertices as ∞, except the source which is 0.

2. Repeat $V - 1$ times:
    - For every edge (u, v) with weight w, update:

$$\text{if } dist[u] + w < dist[v], \text{ then set } dist[v] = dist[u] + w$$

3. Check for negative-weight cycles by repeating the edge-relaxation step once more.
4. If any edge can still be relaxed, report a negative-weight cycle.

Algorithm 1 Bellman-Ford Algorithm

```
Input: Graph G = (V, E) with edge weights (possibly negative), source vertex s
Output: Shortest distances from s to all vertices (or detect negative cycle)
function BellmanFord(G, s)
    for each vertex v in V do
        dist[v] ← ∞
    end for
    dist[s] ← 0
    for i ← 1 to |V| − 1 do
        for each edge (u, v) with weight w in E do
            if dist[u] + w < dist[v] then
                dist[v] ← dist[u] + w
            end if
        end for
    end for
    for each edge (u, v) with weight w in E do
        if dist[u] + w < dist[v] then
            return Negative weight cycle detected
        end if
    end for
    return dist
end function
```

2.7 Comparison with Dijkstra's Algorithm

Feature	**Bellman-Ford**	**Dijkstra**
Negative weights	Supported	Not supported
Time complexity	$O(V \cdot E)$	$O((V + E) \log V)$ (with min-heap)
Cycle detection	Can detect negative weight cycles	Cannot detect cycles
Approach	Edge relaxation ($V - 1$ times)	Greedy approach using Min-Heap

3. Dry Run: Bellman-Ford Algorithm

3.1 Example Graph: With Negative Edge Weight

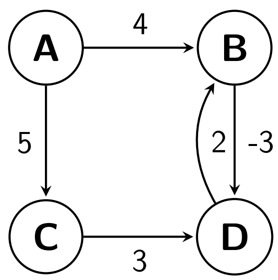

Figure 1: Directed Graph with Negative Edge Weight

Vertices: A, B, C, D
Edges: (A, B, 4), (A, C, 5), (B, D, -3), (C, D, 3), (D, B, 2)

Initialization

Source Vertex: A

Vertex	**Distance from A**
A	0
B	∞
C	∞
D	∞

Relaxation Steps

Bellman-Ford runs for $V - 1 = 3$ iterations.

Iteration 1

Edge	Relaxation	Updated Distances
$(A, B, 4)$	$0 + 4 < \infty$	$B = 4$
$(A, C, 5)$	$0 + 5 < \infty$	$C = 5$
$(B, D, -3)$	$4 - 3 < \infty$	$D = 1$
$(C, D, 3)$	$5 + 3 > 1$	No change
$(D, B, 2)$	$1 + 2 < 4$	$B = 3$

Distances after Iteration 1:

$$A = 0, \quad B = 3, \quad C = 5, \quad D = 1$$

Iteration 2

Edge	Relaxation	Updated Distances
$(A, B, 4)$	$0 + 4 > 3$	$No change$
$(A, C, 5)$	$0 + 5 = 5$	$No change$
$(B, D, -3)$	$3 - 3 = 0 < 1$	$D = 0$
$(C, D, 3)$	$5 + 3 > 0$	$No change$
$(D, B, 2)$	$0 + 2 = 2 < 3$	$B = 2$

Distances after Iteration 2:

$$A = 0, \quad B = 2, \quad C = 5, \quad D = 0$$

Iteration 3

Edge	Relaxation	Updated Distances
$(A, B, 4)$	$No change$	$-$
$(A, C, 5)$	$No change$	$-$
$(B, D, -3)$	$2 - 3 = -1 < 0$	$D = -1$
$(C, D, 3)$	$No change$	$-$
$(D, B, 2)$	$-1 + 2 = 1 < 2$	$B = 1$

Distances after Iteration 3:

$$A = 0, \quad B = 1, \quad C = 5, \quad D = -1$$

Negative Cycle Check

Run one more iteration to check if further relaxation is possible.
- Edge (B, D, -3): $1 - 3 = -2 < -1 \rightarrow$ **Relaxation possible!**

Negative weight cycle detected

—

Output Summary

Result

The graph contains a **negative weight cycle** reachable from the source. The Bellman-Ford algorithm terminates and reports failure.

3.2 Understanding the Dry Run Output

The output of the dry run simulates how the Bellman-Ford algorithm works internally. Here's what each part of the dry run means:

1. Step-by-Step Edge Relaxations

Each iteration updates the shortest known distances from the source vertex (in our case, A) to all other vertices using edge relaxation.

- If $dist[u] + w < dist[v]$, then $dist[v]$ is updated.
- This simulates checking if going through an intermediate node gives a shorter path.

  For example, in Iteration 1:

  - Distance to B becomes 4 via A $\rightarrow$ B
  - Distance to C becomes 5 via A $\rightarrow$ C
  - Distance to D becomes 1 via A $\rightarrow$ B $\rightarrow$ D

2. Multiple Iterations Improve Paths

Bellman-Ford runs the edge relaxation process $V - 1$ times (where V is the number of vertices). This ensures all shortest paths (which can be at most $V - 1$ edges long) are found.

- Distances improve progressively as better paths are discovered.
- For example: D goes from $\infty \to 1 \to 0 \to -1$

3. Final Iteration: Negative Cycle Detection

After the $V - 1$ iterations, Bellman-Ford runs one extra iteration to check if any edge can still be relaxed.

- If yes, then the graph contains a **negative weight cycle**.
- Such cycles allow endlessly reducing path costs, making shortest path meaningless.

In our example:

$$\text{Edge (B, D, -3): } 1 - 3 = -2 < -1$$

$\Rightarrow$ **Negative weight cycle detected**

4. Final Interpretation

What it shows	**Meaning in Algorithm**
Distance updates in each iteration	Edge relaxations updating shortest known paths
Improving distances over rounds	Intermediate vertices offering shorter routes
Negative cycle detection step	Unique ability of Bellman-Ford to identify cycles
Final result box output	Indicates if shortest paths are valid or undefined

5. Why It Matters

Understanding the dry run helps solve problems that ask:

- How many iterations are required?
- Will Bellman-Ford detect a negative cycle?
- Which algorithm (Bellman-Ford or Dijkstra) is more suitable for a graph?

3.3 Example Graph: With Positive Edge Weight

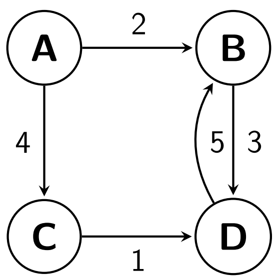

Figure 2: Graph with Positive Edge Weight Cycle

Vertices: A, B, C, D
Edges: (A, B, 2), (A, C, 4), (B, D, 3), (C, D, 1), (D, B, 5)

Initialization

Source Vertex: A

Vertex	Distance from A
A	0
B	∞
C	∞
D	∞

Relaxation Steps (3 Iterations)

Iteration 1

Edge	Relaxation Condition	Updated Distances
$(A, B, 2)$	$0 + 2 < \infty$	$B = 2$
$(A, C, 4)$	$0 + 4 < \infty$	$C = 4$
$(B, D, 3)$	$2 + 3 < \infty$	$D = 5$
$(C, D, 1)$	$4 + 1 = 5 \geq 5$	$No change$
$(D, B, 5)$	$5 + 5 = 10 > 2$	$No change$

After Iteration 1:

$$A = 0, \quad B = 2, \quad C = 4, \quad D = 5$$

Iteration 2

No edge relaxes further.
All conditions fail: distances unchanged.

Iteration 3

Still no changes.

$$\Rightarrow \text{All distances are stable.}$$

Negative Cycle Check

Perform one more iteration:

- No edge (u, v) satisfies $dist[u] + w < dist[v]$
- So, **no negative weight cycle exists**

Final Result

The graph does not contain a negative weight cycle.
Shortest distances from A are:

$$\boxed{A = 0, \quad B = 2, \quad C = 4, \quad D = 5}$$

3.4 Understanding the Output of the Dry Run

The dry run demonstrates how the Bellman-Ford algorithm processes graphs with only positive edge weights. Below is the interpretation of the output:

1. Edge Relaxation Steps

In the first iteration, all reachable vertices from the source (A) have their distances updated because the initial distances are ∞. Each edge is examined and relaxed if a shorter path is found:

- $(A, B, 2)$ sets $dist[B] = 2$
- $(A, C, 4)$ sets $dist[C] = 4$
- $(B, D, 3)$ sets $dist[D] = 5$

No further updates are made in subsequent iterations, which means that the shortest paths have already been found.

2. Stable Distances After Iteration 1

After the first iteration, the shortest distances from the source vertex A to all other vertices are finalized:

$$\boxed{A = 0, \quad B = 2, \quad C = 4, \quad D = 5}$$

- No edge caused a relaxation in iteration 2 or 3.
- This indicates that the shortest paths have been found before completing all $V - 1$ iterations.

3. Cycle Handling with Positive Weights

The graph contains a cycle: $B \to D \to B$, with total weight $3 + 5 = 8$.

- Since the total weight is positive, it does not cause any endless relaxation.
- Bellman-Ford correctly ignores such cycles when computing shortest paths.

4. Final Negative Cycle Check

After the $V - 1$ iterations, Bellman-Ford checks all edges once more to see if any further relaxation is possible.

- No edge satisfies $dist[u] + w < dist[v]$
- Therefore, **no negative weight cycle exists**

Conclusion

The algorithm successfully terminates. All shortest distances from source vertex A have been computed correctly, and no negative weight cycle is present in the graph.

3.5 Python Code

```python
def bellman_ford(V, edges, source):
    # Initialize distances
    dist = [float('inf')] * V
    dist[source] = 0

    # Relax all edges (V - 1) times
    for _ in range(V - 1):
        for u, v, w in edges:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # Check for negative weight cycles
    for u, v, w in edges:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            print("Negative weight cycle detected.")
            return None

    return dist

# Example usage
V = 4
edges = [
    (0, 1, 2),   # A -> B
    (0, 2, 4),   # A -> C
    (1, 3, 3),   # B -> D
    (2, 3, 1),   # C -> D
    (3, 1, 5)    # D -> B (positive cycle)
]
source = 0  # Vertex A
distances = bellman_ford(V, edges, source)

if distances:
    print("Shortest distances from source A:")
    for i, d in enumerate(distances):
        print(f"Vertex {chr(ord('A') + i)}: {d}")
```

Listing 1: Bellman-Ford Algorithm in Python

3.6 Explanation of Bellman-Ford Algorithm in Python

Line 1: Define the function `bellman_ford` with parameters: number of vertices V, list of edges, and the source vertex.

Line 2: Initialize a list `dist` with ∞ for all vertices, representing unreachable distances initially.

Line 3: Set the distance of the source vertex to 0, since the shortest path to itself is 0.

Line 4: Begin the relaxation phase: loop $V-1$ times (as per the Bellman-Ford algorithm).

Line 5: For each edge (u, v, w), check if the distance to v through u is shorter than the current distance.

Line 6: If so, update `dist[v]` with the shorter distance `dist[u] + w`.

Line 7: After all relaxations, check again for each edge whether any further relaxation is possible.

Line 8: If it is, this implies a negative weight cycle exists, so print a warning and return `None`.

Line 9: If no negative cycles are found, return the `dist` list containing the shortest distances.

3.7 Borderline Case Where Bellman-Ford Fails

Case: Negative Weight Cycle Reachable from the Source

The Bellman-Ford algorithm fails when a negative weight cycle is reachable from the source vertex. This is because the algorithm relies on the fact that shortest paths can be calculated in at most $V - 1$ edge relaxations. However, if a cycle with total negative weight exists, the distance to some nodes can always be reduced by going around the cycle repeatedly.

Example

Edge	From	To	Weight
1	A	B	1
2	B	C	-1
3	C	A	-1

This forms a cycle: A → B → C → A with total weight $1+(-1)+(-1) = -1$, which is negative.

If the source is A, Bellman-Ford will:

- Relax edges for $V - 1 = 2$ times.
- On the 3rd pass (for cycle detection), it will detect that further relaxation is possible:

$$\text{dist}[A] > \text{dist}[C] + w(C \to A)$$

- It prints `"Negative weight cycle detected"` and returns `None`.

Why This Happens

Because in the presence of a reachable negative weight cycle:

- The shortest path is not well-defined.
- It can be made infinitely small by repeating the cycle.

Important Notes

- If the cycle is not *reachable* from the source, Bellman-Ford works fine.
- Bellman-Ford is one of the few shortest path algorithms that can even **detect** such cycles.

4. Floyd-Warshall Algorithm

4.1 Purpose

The **Floyd-Warshall Algorithm** is used to compute the shortest paths between **all pairs of vertices** in a weighted directed graph. It works for graphs with positive and negative edge weights (but no negative weight cycles).

Common applications:

- Network routing between all routers
- Finding transitive closures
- Calculating reachability in graphs

4.2 Time Complexity

$$O(V^3)$$

where V is the number of vertices. It uses three nested loops over the vertices.

4.3 Space Complexity

$$O(V^2)$$

because it stores all-pairs shortest distances in a 2D matrix.

4.4 Key Features

- Solves the **All-Pairs Shortest Path** problem
- Works with **negative weights** (but not negative cycles)
- Uses **Dynamic Programming** to build solutions incrementally
- Simpler implementation than running Dijkstra V times

4.5 Step-by-Step Explanation

Let $dist[i][j]$ be the shortest distance from vertex i to vertex j.

The algorithm works as follows:

1. Initialize the distance matrix:

$$dist[i][j] = \begin{cases} 0 & \text{if } i = j \\ \text{weight}(i, j) & \text{if } (i, j) \in E \\ \infty & \text{otherwise} \end{cases}$$

2. For each vertex k, update:

   If $dist[i][k]+dist[k][j] < dist[i][j]$ then set $dist[i][j] = dist[i][k]+dist[k][j]$

3. Repeat this for all $k \in [1, V]$

Algorithm 2 Floyd-Warshall Algorithm

```
Input: Weighted graph G = (V, E) as adjacency matrix
Output: Matrix dist of shortest distances between all pairs
function FloydWarshall(G)
    dist ← adjacency matrix of G
    for each vertex k in V do
        for each vertex i in V do
            for each vertex j in V do
                if dist[i][k] + dist[k][j] < dist[i][j] then
                    dist[i][j] ← dist[i][k] + dist[k][j]
                end if
            end for
        end for
    end for
    return dist
end function
```

4.6 Comparison with Dijkstra's Algorithm

Feature	**Floyd-Warshall**	**Dijkstra**
Problem Type	All-pairs shortest path	Single-source shortest path
Time Complexity	$O(V^3)$	$O((V + E)\log V)$ (with min-heap)
Handles Negative Weights	Yes	No (fails for negative weights)
Negative Cycle Detection	No (but can be checked manually)	No
Graph Type	Dense	Sparse
Approach	Dynamic Programming	Greedy + Min-Heap
Ease of Implementation	Very simple	More complex with heap

5. Dry Run: Floyd-Warshall Algorithm

5.1 Example Graph: With Negative Edge Weight

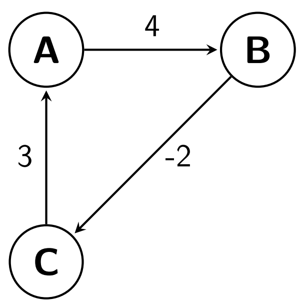

Figure 3: Graph with Negative Edge Weight (No negative cycle)

Vertices: A = 0, B = 1, C = 2
Edges:

$$(0, 1, 4), \quad (1, 2, -2), \quad (2, 0, 3)$$

Initial Distance Matrix:

$$\begin{bmatrix} 0 & 4 & \infty \\ \infty & 0 & -2 \\ 3 & \infty & 0 \end{bmatrix}$$

After k=0:

No changes (only self-loops)

After k=1:

$$dist[0][2] = dist[0][1] + dist[1][2] = 4 + (-2) = 2$$

After k=2:

$$dist[1][0] = dist[1][2] + dist[2][0] = -2 + 3 = 1$$

$$dist[1][1] = dist[1][0] + dist[0][1] = 1 + 4 = 5$$

5.2 Understanding the Dry Run Output

- Floyd-Warshall computes the shortest path between all node pairs.
- It progressively improves the solution by considering each vertex as an intermediate step.
- Negative edge weights are handled properly, as long as there are no negative cycles.
- If any $dist[i][i] < 0$ at the end, a **negative cycle** is detected.

Final Result (Negative Edge)
Shortest path matrix is successfully computed with negative weights (no cycle).

5.3 Example Graph: With Positive Edge Weight

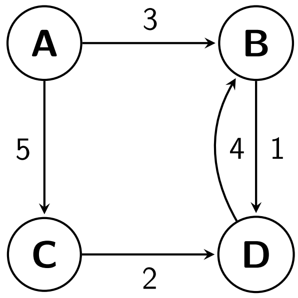

Figure 4: Graph with Positive Edge Weights

Vertices: A = 0, B = 1, C = 2, D = 3

Initial Distance Matrix:

$$\begin{bmatrix} 0 & 3 & 5 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & \infty & 0 & 2 \\ \infty & 4 & \infty & 0 \end{bmatrix}$$

5.4 Understanding the Output of the Dry Run

- After multiple updates, all indirect shortest paths are computed.
- The algorithm detects no negative cycles.
- Final matrix contains the shortest distances between all vertex pairs.

Final Result (Positive Edges)

All-pairs shortest paths successfully computed with only positive weights.

5.5 Python Code

```
def floyd_warshall(V, edges):
    dist = [[float('inf')] * V for _ in range(V)]

    for i in range(V):
        dist[i][i] = 0

    for u, v, w in edges:
        dist[u][v] = w

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    # Optional: detect negative cycles
    for i in range(V):
        if dist[i][i] < 0:
            print("Negative weight cycle detected.")
            return None

    return dist

# Example usage:
edges = [
    (0, 1, 3),  # A -> B
    (0, 2, 5),  # A -> C
    (1, 3, 1),  # B -> D
    (2, 3, 2),  # C -> D
    (3, 1, 4)   # D -> B
]
dist = floyd_warshall(4, edges)
for i, row in enumerate(dist):
    print(f"From {chr(65+i)}:", row)
```

Listing 2: Floyd-Warshall Algorithm in Python

5.6 Explanation of Floyd-Warshall Algorithm (Python)

1. `def floyd_warshall(V, edges):`
   Defines a function that takes the number of vertices V and a list of edges. Each edge is a tuple (u, v, w) representing an edge from u to v with weight w.

2. `dist = [[float('inf')] * V for _ in range(V)]`
   Initializes a $V \times V$ distance matrix with ∞, meaning all distances are initially unknown.

3. `for i in range(V):`
   Loop over all vertices to set distance from a vertex to itself as 0.

4. `dist[i][i] = 0`
   The shortest distance from any vertex to itself is 0.

5. `for u, v, w in edges:`
   Iterates over each edge in the input edge list.

6. `dist[u][v] = w`
   Updates the matrix with the given edge weights.

7. `for k in range(V):`
   Outer loop over all vertices k. This represents intermediate nodes in the path.

8. `for i in range(V):`
   Inner loop for source vertices.

9. `for j in range(V):`
   Inner loop for destination vertices.

10. `if dist[i][k] + dist[k][j] < dist[i][j]:`
    If the path from i to j via k is shorter than the current known path, update it.

11. `dist[i][j] = dist[i][k] + dist[k][j]`
    Update the shortest distance from i to j using k as an intermediate.

12. `for i in range(V):`
    After all updates, check for negative weight cycles.

13. `if dist[i][i] < 0:`
    If the diagonal of the matrix is negative, it indicates a negative weight cycle involving vertex i.

14. `print("Negative weight cycle detected.")`
    Warn the user about the presence of a negative cycle.

15. `return None`
    Abort and return `None` to indicate failure.

16. `return dist`
    If no negative cycle is found, return the final distance matrix.

5.7 Borderline Case: When Floyd-Warshall Fails

Case: Graph with a Negative Weight Cycle

The Floyd-Warshall algorithm fails to compute valid shortest paths when a **negative weight cycle** exists in the graph.

Why it Fails

- In a negative cycle, you can loop through the cycle indefinitely to reduce the total path cost.
- As a result, the concept of a ”shortest path” is not well-defined.
- Floyd-Warshall relies on dynamic programming assuming that once the shortest path between any two nodes is found, it cannot get shorter — this assumption breaks in the presence of negative cycles.

How to Detect It

- After running the algorithm, check the diagonal entries of the distance matrix:

$$\text{If } dist[i][i] < 0 \text{ for any } i, \text{ a negative cycle exists.}$$

Example

A graph with the following edges:

- A → B (weight = 1)
- B → C (weight = -2)
- C → A (weight = -2)

Forms a cycle A → B → C → A with

$$\text{total weight} = 1 + (-2) + (-2) = -3.$$

This is a negative cycle.
Floyd-Warshall will detect this because:

$$dist[A][A] = -3 < 0$$

Hence, result is invalid and must be discarded.

5.8 Logic to Detect Negative Weight Cycles

The Floyd-Warshall algorithm detects negative weight cycles by analyzing the final values in the distance matrix.
Key Observations:

- Initially, the diagonal entries of the distance matrix are all set to 0:

$$\texttt{dist[i][i] = 0} \quad \text{for all } i$$

- During the algorithm's execution, the matrix is updated using:

```
dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

- If a node can reach itself with a total cost less than 0, then a negative weight cycle exists.

Detection Condition:

$$\exists\, i \in V \text{ such that } \texttt{dist[i][i] < 0}$$

Interpretation:

- This means there exists a cycle starting and ending at node i with negative total weight.
- The algorithm will detect this and report the cycle if such a condition is found.

Conclusion: If any diagonal entry of the final distance matrix is negative, the graph contains a negative weight cycle, and shortest paths are undefined.

6. 10 Key Points:

6.1 Bellman-Ford Algorithm

1. Solves **Single-Source Shortest Path** (SSSP) problem.
2. Works with **negative edge weights**.
3. Detects **negative weight cycles**.
4. Time Complexity: $O(V \cdot E)$.
5. Uses **edge relaxation** process up to $V - 1$ times.
6. Final pass checks for further relaxation to detect cycles.
7. Works for both **directed** and **undirected** graphs.
8. Distance array initialized with ∞; source $= 0$.
9. Slower than Dijkstra but more versatile.
10. Common in scenarios with possible **debt or penalties**.

6.2 Floyd-Warshall Algorithm

1. Solves **All-Pairs Shortest Path** (APSP) problem.
2. Based on **dynamic programming**.
3. Handles **negative edge weights**, not negative cycles.
4. Time Complexity: $O(V^3)$.
5. Space Complexity: $O(V^2)$ using a 2D matrix.
6. Initializes distance matrix with direct edge weights.
7. Triple nested loop updates all distances.
8. Simple to implement; good for **dense graphs**.
9. Can be used to detect negative cycles via diagonal $dist[i][i] < 0$.
10. Used in **network routing**, **transitive closure**, etc.

7. Final Summary: Bellman-Ford vs Floyd-Warshall

Bellman-Ford Algorithm Summary

- Solves **Single-Source Shortest Path (SSSP)** problem.
- Works with **negative weights** and **detects negative cycles**.
- Time Complexity: $O(V \cdot E)$
- Uses **edge relaxation** repeated $V - 1$ times.
- Ideal for graphs with negative weights and when only one source is given.

Floyd-Warshall Algorithm Summary

- Solves **All-Pairs Shortest Path (APSP)** problem.
- Handles **negative edge weights** (not cycles).
- Time Complexity: $O(V^3)$
- Based on **dynamic programming** and a distance matrix.
- Useful in dense graphs and applications like routing, transitive closure.

Aspect	**Bellman-Ford**	**Floyd-Warshall**
Problem Solved	Single-source shortest path	All-pairs shortest path
Handles Negative Weights	Yes	Yes
Negative Cycle Detection	Yes	Via diagonal check: $dist[i][i] < 0$
Time Complexity	$O(V \cdot E)$	$O(V^3)$
Approach	Edge relaxation	Dynamic programming
Graph Type	Directed / undirected	Directed (preferably dense)
Use Case	When source node is known	When all node pairs matter

Table 1: Comparison Summary: Bellman-Ford vs Floyd-Warshall

Feature	**Bellman-Ford**	**Dijkstra**	**Floyd-Warshall**
Problem Type	Single-source shortest path (SSSP)	Single-source shortest path (SSSP)	All-pairs shortest path (APSP)
Graph Type	Directed / Undirected	Directed / Undirected	Directed
Negative Weights	Supported	Not supported	Supported (no negative cycles)
Negative Cycle Detection	Yes	No	Can be checked via $dist[i][i] < 0$
Time Complexity	$O(V \cdot E)$	$O((V + E) \log V)$ (with heap)	$O(V^3)$
Space Complexity	$O(V)$	$O(V)$	$O(V^2)$
Algorithm Type	Edge Relaxation (DP-based)	Greedy + Min-Heap	Dynamic Programming Matrix
Ease of Implementation	Easy	Medium (with heap)	Very Easy
Best for	Graphs with negative weights / cycle detection	Sparse graphs	Dense graphs / APSP
Path Reconstruction	Parent array	Parent array	Predecessor matrix (optional)

Table 2: Comparison Summary: Bellman-Ford vs Dijkstra vs Floyd-Warshall