CS 2500: Algorithms

Lecture 27: Dynamic Programming: DP for Graphs

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

November 19, 2024

All-Pairs Shortest Paths: Floyd-Warshall Algorithm

Problem Definition:

- Given a directed graph $G = (V, E)$ with n vertices.
- Let cost(i, j) be the cost adjacency matrix where:
  - cost$(i, i) = 0$ for $1 \leq i \leq n$.
  - cost$(i, j) = \infty$ if $\langle i, j \rangle \notin E(G)$.
  - cost(i, j) is the length of edge $\langle i, j \rangle$ if $\langle i, j \rangle \in E(G)$ and $i \neq j$.
- The goal is to determine a matrix A such that $A(i, j)$ represents the shortest path length from i to j.

All-Pairs Shortest Paths: Floyd-Warshall Algorithm

Key Points:

- Compute A using $O(n^3)$ time with the principle of optimality.
- The graph G must not contain a cycle of negative length.

Dynamic Programming Principle:

- Consider a shortest path $i \to j$ passing through vertex k.
- The subpaths $i \to k$ and $k \to j$ must also be shortest paths.
- Recursive definition:

$$A^k(i,j) = \min(A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j))$$

- Base case:

$$A^0(i,j) = \mathrm{cost}(i,j)$$

Floyd-Warshall Be Calculated In-Place

- The computation of $A^k(i,j)$ only depends on values from A^{k-1}, specifically:

$$A^k(i,j) = \min\left(A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\right).$$

- During the calculation of $A^k(i,j)$:
  - $A^k(i,k) = A^{k-1}(i,k)$,
  - $A^k(k,j) = A^{k-1}(k,j)$.
- Since $A^k(i,j)$ is calculated row by row and column by column:
  - The current row and column values for k do not change during the iteration.
  - This ensures the correctness of the updates for other entries.
- The algorithm reuses the same matrix A without needing an additional data structure:
- This optimization ensures a space complexity of $O(n^2)$, which is crucial for large graphs.

All-Pairs Shortest Paths: Floyd-Warshall Algorithm

Algorithm 1 All-Pairs Shortest Paths (Floyd-Warshall Algorithm)

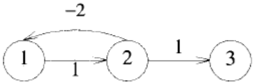
Require: $\text{cost}[1 \ldots n, 1 \ldots n]$: cost adjacency matrix of the graph
Ensure: $\mathrm{A}[1 \ldots n, 1 \ldots n]$: shortest path lengths between all pairs of vertices

Initialize $\mathrm{A}[i, j] \leftarrow \text{cost}[i, j]$ for all $1 \leq i, j \leq n$
for $k = 1$ to n **do**
    for $i = 1$ to n **do**
        for $j = 1$ to n **do**
            $\mathrm{A}[i, j] \leftarrow \min(\mathrm{A}[i, j], \mathrm{A}[i, k] + \mathrm{A}[k, j])$
        end for
    end for
end for
return A $\triangleright$ Matrix containing shortest path lengths between all vertex pairs

Time Complexity: $O(n^3)$.

All-Pairs Shortest Paths: Floyd-Warshall Algorithm

Example: Graph with Negative Cycle



- Adjacency matrix:

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

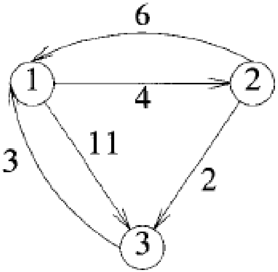
- Negative cycle: $1 \to 2 \to 1$.

Consequence:

- Length of path $1, 2, 1, 2, 1, 2, \ldots 1, 2, 3$ can become arbitrarily small.
- Shortest path computation fails.

All-Pairs Shortest Paths: Floyd-Warshall Algorithm

Example: Directed Graph



Initial Cost Matrix (A^0):

$$A^0 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

All-Pairs Shortest Paths: First Iteration ($k = 1$)

Update Rule:

$$A^1(i,j) = \min(A^0(i,j), A^0(i,1) + A^0(1,j))$$

Matrix After Update (A^1):

$$A^1 = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Steps:

- For $i = 3$, $j = 2$:

  $$A^1(3,2) = \min(A^0(3,2), A^0(3,1)+A^0(1,2)) = \min(\infty, 3+4) = 7$$

- All other values remain the same as no shorter paths are found through vertex 1.

All-Pairs Shortest Paths: Second Iteration ($k = 2$)

Update Rule:

$$A^2(i,j) = \min(A^1(i,j), A^1(i,2) + A^1(2,j))$$

Matrix After Update (A^2):

$$A^2 = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Steps:

- For $i = 1$, $j = 3$:

  $$A^2(1,3) = \min(A^1(1,3), A^1(1,2)+A^1(2,3)) = \min(11, 4+2) = 6$$

- All other values remain the same as no shorter paths are found through vertex 2.

All-Pairs Shortest Paths: Third Iteration ($k = 3$)

Update Rule:

$$A^3(i,j) = \min(A^2(i,j), A^2(i,3) + A^2(3,j))$$

Matrix After Update (A^3):

$$A^3 = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Steps:

- For $i = 2$, $j = 1$:

  $$A^3(2,1) = \min(A^2(2,1), A^2(2,3) + A^2(3,1)) = \min(6, 2+3) = 5$$

- All other values remain the same as no shorter paths are found through vertex 3.

All-Pairs Shortest Paths: Final Result

Shortest Path Matrix (A^3):

$$A^3 = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Interpretation:

- $A^3(1,3) = 6$: Shortest path from vertex 1 to 3.
- $A^3(2,1) = 5$: Shortest path from vertex 2 to 1.
- $A^3(3,2) = 7$: Shortest path from vertex 3 to 2.

Key Observations:

- The algorithm successfully computes all-pairs shortest paths.
- Negative weight cycles would make the result invalid, but none exist in this graph.

Single-Source Shortest Path

Goal:

- Compute the shortest paths from a source vertex s to all other vertices in a graph $G = (V, E)$.
- Algorithm: Dijkstra

Problem:

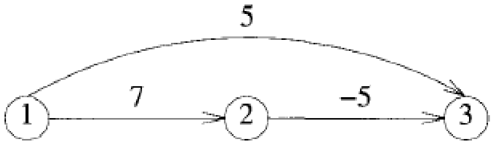
- Graph G may have negative edge weights.
- Shortest path algorithms like Dijkstra's may not compute correct results.

Example:

- Algorithm terminates with:

$$path[2] = 7, \quad path[3] = 5$$

- Actual shortest path: $1 \to 2 \to 3$, with length 2.



Single-Source Shortest Path: Bellman-Ford Algorithm

Key Features:

- Handles graphs with negative edge weights.
- Detects negative weight cycles.

Input:

- Graph G (as an adjacency matrix or edge list), source node s.

Output:

- Shortest distance array $path[]$: Distance from s to every other vertex.
- Predecessor array $pred[]$: Helps reconstruct shortest paths.

Single-Source Shortest Path: Bellman-Ford Algorithm

Recursive Definition:

- Define $path^k[v]$ as the shortest distance from the source s to vertex v using at most k edges.

Base Case:

$$path^0[v] = \begin{cases} 0, & \text{if } v = s, \\ +\infty, & \text{if } v \neq s. \end{cases}$$

Recursive Case:

$$path^k[v] = \min\left(path^{k-1}[v], \min_{u \in V}\{path^{k-1}[u] + adj[u][v]\}\right)$$

Explanation:

- $path^k[v]$ is the minimum of:
  - The shortest distance to v using at most $k-1$ edges ($path^{k-1}[v]$).
  - The distance to v via some neighbor u, considering one additional edge.

Single-Source Shortest Path: Bellman-Ford Algorithm

Initialization:

1. Initialize *path*[] to $+\infty$ for all nodes, except $path[s] = 0$ for the source node s.
2. Initialize *pred*[] to **NIL** for all nodes.

Relaxation:

1. For each of the $n - 1$ iterations:
   - For each edge (u, v), update:

     If $path[u] + adj[u][v] < path[v]$ then: $\begin{array}{l} path[v] \leftarrow path[u] + adj[u][v], \\ pred[v] \leftarrow u. \end{array}$

Negative Weight Cycle Detection:

1. Check all edges (u, v) for further relaxation:

   If $path[u] + adj[u][v] < path[v]$, then report a negative weight cycle.

Output:

- Return *path*[] and *pred*[].

Single-Source Shortest Path: Bellman-Ford Algorithm

Algorithm 2 Bellman-Ford (Graph G, Adjacency Matrix adj, Source Node s)

Require: G: Graph, adj: Adjacency matrix representing edge weights, s: Source node

Ensure: $path[]$: Shortest distances from s to all other nodes

Ensure: $pred[]$: Predecessor array for shortest paths

```
Initialize path[] with +∞ for all nodes except the source
path[s] ← 0                                  ▷ Distance from source to itself is zero
Initialize pred[] with NIL for all nodes
for k := 1 to n − 1 do                       ▷ Relax edges up to n − 1 times
    for each node u ∈ V do
        for each neighbor v of u such that adj[u][v] ≠ 0 do
            if path[v] > path[u] + adj[u][v] then
                path[v] ← path[u] + adj[u][v]
                pred[v] ← u
            end if
        end for
    end for
end for
                                             ▷ Check for negative weight cycles
for each edge (u, v) in G do
    if path[v] > path[u] + adj[u][v] then
        Report: "Graph contains a negative weight cycle"
        return
    end if
end for
return path[], pred[]
```

Single-Source Shortest Path: Bellman-Ford Algorithm

Time Complexity:

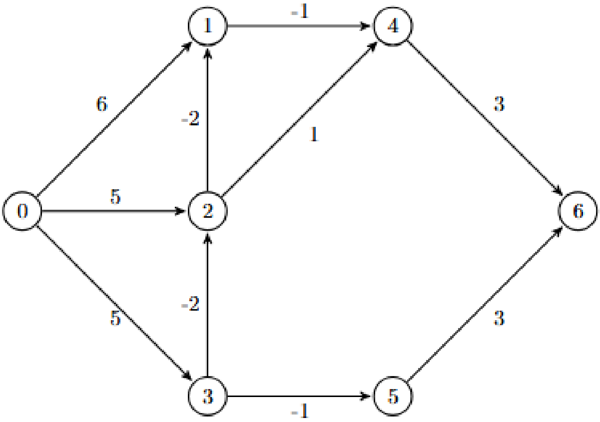
- $O(n^3)$ using adjacency matrices.
- $O(ne)$ using adjacency lists (e: number of edges).

Space Complexity:

- $O(n)$ for *path*[] and *pred*[].

Example: Bellman-Ford Algorithm

Compute shortest paths from source vertex 1 to all other vertices using the Bellman-Ford algorithm.



Example: Bellman-Ford Algorithm

Edge	Weight
(0 → 0)	0
(0 → 1)	6
(0 → 2)	5
(0 → 3)	5
(1 → 1)	0
(1 → 4)	-1
(2 → 1)	-2
(2 → 2)	0
(2 → 4)	1
(3 → 2)	-2
(3 → 3)	0
(3 → 5)	-1
(4 → 4)	0
(4 → 6)	3
(5 → 5)	0
(5 → 6)	3
(6 → 6)	0

Example: Bellman-Ford Algorithm–Initialization

Initial distances: [0, ∞, ∞, ∞, ∞, ∞, ∞]
Initial predecessors: [None, None, None, None, None, None, None]

Example: Bellman-Ford Algorithm–Iteration 1

Edge	Weight	Relaxed?	Reason/Details	Updated Distances
$(0 \to 0)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, ∞, ∞, ∞, ∞, ∞, ∞]
$(0 \to 1)$	6	Yes	Updated dist[1] to 6, as $0 + 6 = 6 < \infty$. Predecessor set to 0.	[0, 6, ∞, ∞, ∞, ∞, ∞]
$(0 \to 2)$	5	Yes	Updated dist[2] to 5, as $0 + 5 = 5 < \infty$. Predecessor set to 0.	[0, 6, 5, ∞, ∞, ∞, ∞]
$(0 \to 3)$	5	Yes	Updated dist[3] to 5, as $0 + 5 = 5 < \infty$. Predecessor set to 0.	[0, 6, 5, 5, ∞, ∞, ∞]
$(1 \to 1)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 6, 5, 5, ∞, ∞, ∞]
$(1 \to 4)$	-1	Yes	Updated dist[4] to 5, as $6 + (-1) = 5 < \infty$. Predecessor set to 1.	[0, 6, 5, 5, 5, ∞, ∞]
$(2 \to 1)$	-2	Yes	Updated dist[1] to 3, as $5 + (-2) = 3 < 6$. Predecessor set to 2.	[0, 3, 5, 5, 5, ∞, ∞]
$(2 \to 2)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 5, 5, 5, ∞, ∞]
$(2 \to 4)$	1	No	New distance $5+1 = 6$, but $6 \geq$ current dist[4] $= 5$.	[0, 3, 5, 5, 5, ∞, ∞]
$(3 \to 2)$	-2	Yes	Updated dist[2] to 3, as $5 + (-2) = 3 < 5$. Predecessor set to 3.	[0, 3, 3, 5, 5, ∞, ∞]
$(3 \to 3)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 3, 5, 5, ∞, ∞]
$(3 \to 5)$	-1	Yes	Updated dist[5] to 4, as $5 + (-1) = 4 < \infty$. Predecessor set to 3.	[0, 3, 3, 5, 5, 4, ∞]
$(4 \to 4)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 3, 5, 5, 4, ∞]
$(4 \to 6)$	3	Yes	Updated dist[6] to 8, as $5 + 3 = 8 < \infty$. Predecessor set to 4.	[0, 3, 3, 5, 5, 4, 8]
$(5 \to 5)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 3, 5, 5, 4, 8]
$(5 \to 6)$	3	Yes	Updated dist[6] to 7, as $4 + 3 = 7 < 8$. Predecessor set to 5.	[0, 3, 3, 5, 5, 4, 7]
$(6 \to 6)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 3, 5, 5, 4, 7]

Table 1: Iteration 1: Edge Relaxations and Updated Distances

Distances after iteration 1: [0, 3, 3, 5, 5, 4, 7]
Predecessors after iteration 1: [None, 2, 3, 0, 1, 3, 5]

Example: Bellman-Ford Algorithm–Iteration 2

Edge	Weight	Relaxed?	Reason/Details	Updated Distances
$(0 \to 0)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 3, 5, 5, 4, 7]
$(0 \to 1)$	6	No	New distance $0+6=6$, but $6 \geq$ current dist[1] $=3$.	[0, 3, 3, 5, 5, 4, 7]
$(0 \to 2)$	5	No	New distance $0+5=5$, but $5 \geq$ current dist[2] $=3$.	[0, 3, 3, 5, 5, 4, 7]
$(0 \to 3)$	5	No	New distance $0+5=5$, but $5 \geq$ current dist[3] $=5$.	[0, 3, 3, 5, 5, 4, 7]
$(1 \to 1)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 3, 3, 5, 5, 4, 7]
$(1 \to 4)$	-1	Yes	Updated dist[4] to 2, as $3+(-1)=2<5$. Predecessor set to 1.	[0, 3, 3, 5, 2, 4, 7]
$(2 \to 1)$	-2	Yes	Updated dist[1] to 1, as $3+(-2)=1<3$. Predecessor set to 2.	[0, 1, 3, 5, 2, 4, 7]
$(2 \to 2)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 7]
$(2 \to 4)$	1	No	New distance $3+1=4$, but $4 \geq$ current dist[4] $=2$.	[0, 1, 3, 5, 2, 4, 7]
$(3 \to 2)$	-2	No	New distance $5+(-2)=3$, but $3 \geq$ current dist[2] $=3$.	[0, 1, 3, 5, 2, 4, 7]
$(3 \to 3)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 7]
$(3 \to 5)$	-1	No	New distance $5+(-1)=4$, but $4 \geq$ current dist[5] $=4$.	[0, 1, 3, 5, 2, 4, 7]
$(4 \to 4)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 7]
$(4 \to 6)$	3	Yes	Updated dist[6] to 5, as $2+3=5<7$. Predecessor set to 4.	[0, 1, 3, 5, 2, 4, 5]
$(5 \to 5)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 5]
$(5 \to 6)$	3	No	New distance $4+3=7$, but $7 \geq$ current dist[6] $=5$.	[0, 1, 3, 5, 2, 4, 5]
$(6 \to 6)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 5]

Table 2: Iteration 2: Edge Relaxations and Updated Distances

Distances after iteration 2: [0, 1, 3, 5, 2, 4, 5]
Predecessors after iteration 2: [None, 2, 3, 0, 1, 3, 4]

Example: Bellman-Ford Algorithm–Iteration 3

Edge	Weight	Relaxed?	Reason/Details	Updated Distances
$(0 \to 0)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 5]
$(0 \to 1)$	6	No	New distance $0+6=6$, but $6 \geq$ current dist[1] $=1$.	[0, 1, 3, 5, 2, 4, 5]
$(0 \to 2)$	5	No	New distance $0+5=5$, but $5 \geq$ current dist[2] $=3$.	[0, 1, 3, 5, 2, 4, 5]
$(0 \to 3)$	5	No	New distance $0+5=5$, but $5 \geq$ current dist[3] $=5$.	[0, 1, 3, 5, 2, 4, 5]
$(1 \to 1)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 2, 4, 5]
$(1 \to 4)$	-1	Yes	Updated dist[4] to 0, as $1+(-1)=0<2$. Predecessor set to 1.	[0, 1, 3, 5, 0, 4, 5]
$(2 \to 1)$	-2	No	New distance $3+(-2)=1$, but $1 \geq$ current dist[1] $=1$.	[0, 1, 3, 5, 0, 4, 5]
$(2 \to 2)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 5]
$(2 \to 4)$	1	No	New distance $3+1=4$, but $4 \geq$ current dist[4] $=0$.	[0, 1, 3, 5, 0, 4, 5]
$(3 \to 2)$	-2	No	New distance $5+(-2)=3$, but $3 \geq$ current dist[2] $=3$.	[0, 1, 3, 5, 0, 4, 5]
$(3 \to 3)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 5]
$(3 \to 5)$	-1	No	New distance $5+(-1)=4$, but $4 \geq$ current dist[5] $=4$.	[0, 1, 3, 5, 0, 4, 5]
$(4 \to 4)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 5]
$(4 \to 6)$	3	Yes	Updated dist[6] to 3, as $0+3=3<5$. Predecessor set to 4.	[0, 1, 3, 5, 0, 4, 3]
$(5 \to 5)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(5 \to 6)$	3	No	New distance $4+3=7$, but $7 \geq$ current dist[6] $=3$.	[0, 1, 3, 5, 0, 4, 3]
$(6 \to 6)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]

Table 3: Iteration 3: Edge Relaxations and Updated Distances

Distances after iteration 3: [0, 1, 3, 5, 0, 4, 3]
Predecessors after iteration 3: [None, 2, 3, 0, 1, 3, 4]

Example: Bellman-Ford Algorithm–Iteration 4

Edge	Weight	Relaxed?	Reason/Details	Updated Distances
$(0 \to 0)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(0 \to 1)$	6	No	New distance $0+6=6$, but $6 \geq$ current dist[1] $= 1$.	[0, 1, 3, 5, 0, 4, 3]
$(0 \to 2)$	5	No	New distance $0+5=5$, but $5 \geq$ current dist[2] $= 3$.	[0, 1, 3, 5, 0, 4, 3]
$(0 \to 3)$	5	No	New distance $0+5=5$, but $5 \geq$ current dist[3] $= 5$.	[0, 1, 3, 5, 0, 4, 3]
$(1 \to 1)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(1 \to 4)$	-1	No	New distance $1+(-1)=0$, but $0 \geq$ current dist[4] $= 0$.	[0, 1, 3, 5, 0, 4, 3]
$(2 \to 1)$	-2	No	New distance $3+(-2)=1$, but $1 \geq$ current dist[1] $= 1$.	[0, 1, 3, 5, 0, 4, 3]
$(2 \to 2)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(2 \to 4)$	1	No	New distance $3+1=4$, but $4 \geq$ current dist[4] $= 0$.	[0, 1, 3, 5, 0, 4, 3]
$(3 \to 2)$	-2	No	New distance $5+(-2)=3$, but $3 \geq$ current dist[2] $= 3$.	[0, 1, 3, 5, 0, 4, 3]
$(3 \to 3)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(3 \to 5)$	-1	No	New distance $5+(-1)=4$, but $4 \geq$ current dist[5] $= 4$.	[0, 1, 3, 5, 0, 4, 3]
$(4 \to 4)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(4 \to 6)$	3	No	New distance $0+3=3$, but $3 \geq$ current dist[6] $= 3$.	[0, 1, 3, 5, 0, 4, 3]
$(5 \to 5)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]
$(5 \to 6)$	3	No	New distance $4+3=7$, but $7 \geq$ current dist[6] $= 3$.	[0, 1, 3, 5, 0, 4, 3]
$(6 \to 6)$	0	No	Edge connects the node to itself, distance remains unchanged.	[0, 1, 3, 5, 0, 4, 3]

Table 4: Iteration 4: Edge Relaxations and Updated Distances

Distances after iteration 4: [0, 1, 3, 5, 0, 4, 3]
Predecessors after iteration 4: [None, 2, 3, 0, 1, 3, 4]

Bellman-Ford Algorithm: Proof of Correctness

We must prove the following:

1. **Theorem 1: Bellman-Ford detects negative cycles:**
   - **This means:** If there is a negative cycle reachable from the source s, then for some edge (u, v), $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.
2. **Theorem 2: If the graph has no negative cycles then, the distance estimates on the last iteration are equal to the true shortest distances**
   - **This means:** $d_{n-1}(v) = \delta(s, v)$ for all vertices v.

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v),
  $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Why do we want to prove this?

- The Bellman-Ford algorithm performs $n - 1$ iterations of edge relaxation, where n is the number of vertices in the graph.
- **Why** $n - 1$ **iterations?**
  - The longest simple path in a graph (i.e., a path with no repeated vertices) can have at most $n - 1$ edges.
  - Therefore, after $n - 1$ iterations, the shortest-path information for all vertices should propagate fully if there are no negative cycles.

Bellman-Ford Algorithm: Proof of Correctness

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v), $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

What happens in the presence of a negative cycle?

- If a negative cycle is reachable, the distance estimates $d(v)$ for vertices in or reachable from the cycle will decrease indefinitely with each traversal of the cycle.
- Even after $n - 1$ iterations, some edge (u, v) will satisfy:

$$d(v) > d(u) + w(u, v).$$

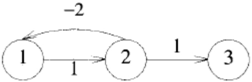
- This indicates that $d(v)$ can still decrease further. Such behavior is only possible if the graph contains a negative-weight cycle.

Bellman-Ford Algorithm: Proof of Correctness

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v),
  $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Example:



- Consider the following graph with a negative cycle $1 \to 2 \to 1$ and an additional path $1 \to 2 \to 3$.
- The total weight of the cycle $1 \to 2 \to 1$ is:

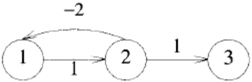
$$w(1, 2) + w(2, 1) = 1 + (-2) = -1.$$

Bellman-Ford Algorithm: Proof of Correctness

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v), $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Example:



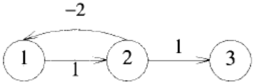
- **Behavior of Bellman-Ford:**
  - Start from 1 with initial distances:

$$d(1) = 0,\ d(2) = \infty,\ d(3) = \infty.$$

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v),
  $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Example:



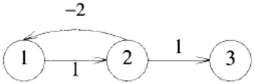
- **Behavior of Bellman-Ford:**
  - After relaxing edges once:

$$d(2) = 0 + 1 = 1,\ d(1) = 1 + (-2) = -1,\ d(3) = 1 + 1 = 2.$$

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v),
  $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Example:



- **Behavior of Bellman-Ford:**
  - After a second relaxation:

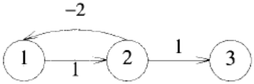
    $d(2) = -1 + 1 = 0,\ d(1) = 0 + (-2) = -2,\ d(3) = 0 + 1 = 1.$

Bellman-Ford Algorithm: Proof of Correctness

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v), $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Example:



- **Behavior of Bellman-Ford:**
  - Repeating the cycle further decreases distances:

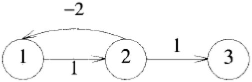
$$d(2) = -2+1 = -1,\ d(1) = -1+(-2) = -3,\ d(3) = -1+1 = 0.$$

Bellman-Ford Algorithm: Proof of Correctness

Theorem 1: Bellman-Ford detects negative cycles:

- If there is a negative cycle reachable from the source s, then for some edge (u, v), $d_{n-1}(v) > d_{n-1}(u) + w(u, v)$.

Example:



- **Conclusion:**
  - Each traversal of the cycle $1 \to 2 \to 1$ reduces distances $d(1)$ and $d(2)$, causing $d(v) > d(u) + w(u, v)$ to persist indefinitely.
  - Thus, the condition $d(v) > d(u) + w(u, v)$ demonstrates the presence of a negative cycle.

Proof: Bellman-Ford Detects Negative Cycles

- Suppose a negative cycle $v_0 \to v_1 \to \ldots \to v_k$ exists and is reachable from the source s, where $v_0 = v_k$ (i.e., the cycle starts and ends at the same vertex).
- **What does a negative cycle mean?**
  - A negative cycle is a cycle for which the sum of edge weights is less than zero:
  $$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$
  - If a negative cycle is reachable from the source, the distance estimates $d(v)$ for vertices in or reachable from this cycle should keep decreasing indefinitely as the cycle is traversed repeatedly.

Proof: Bellman-Ford Detects Negative Cycles

- **Assumption for Proof by Contradiction:**
  - Assume Bellman-Ford's distance estimates after $n-1$ iterations satisfy:

$$d_{n-1}(v_i) \leq d_{n-1}(v_{i-1}) + w(v_{i-1}, v_i) \quad \text{for all } i = 1, \ldots, k.$$

  - This assumption means that Bellman-Ford would not further decrease any distances in the cycle after $n-1$ iterations.
- **What happens if we sum up these inequalities?**
  - Summing over all edges in the cycle gives:

$$\sum_{i=1}^{k} d_{n-1}(v_i) \leq \sum_{i=1}^{k} d_{n-1}(v_{i-1}) + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

  - Rearrange terms:

$$\sum_{i=1}^{k} d_{n-1}(v_i) - \sum_{i=1}^{k} d_{n-1}(v_{i-1}) \leq \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Detects Negative Cycles

- **Observe the left-hand side of the summed inequality:**
  - Rearranged sum:

$$\sum_{i=1}^{k} d_{n-1}(v_i) - \sum_{i=1}^{k} d_{n-1}(v_{i-1}) \leq \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

  - The terms $\sum_{i=1}^{k} d_{n-1}(v_i)$ and $\sum_{i=1}^{k} d_{n-1}(v_{i-1})$ are identical because the cycle starts and ends at the same vertex $(v_0 = v_k)$.
    - Each vertex in the cycle is visited once in $\sum_{i=1}^{k} d_{n-1}(v_i)$ and once in $\sum_{i=1}^{k} d_{n-1}(v_{i-1})$.
    - Since $v_0 = v_k$, the summation over all v_i is circularly shifted, but the total remains the same.
  - This simplifies the inequality to:

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Proof: Bellman-Ford Detects Negative Cycles

- **Contradiction:**
  - By definition of a negative cycle, $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.
  - This contradicts the inequality $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$, which means our assumption that Bellman-Ford stops updating distances is false.
  - Hence, Bellman-Ford correctly detects negative cycles if any edge (u, v) satisfies:

$$d_{n-1}(v) > d_{n-1}(u) + w(u, v).$$

Bellman-Ford Algorithm: Proof for Correctness

Theorem 2: If the graph has no negative cycles then The distance estimates on the last iteration are equal to the true shortest distances

- $d_{n-1}(v) = \delta(s, v)$ for all vertices v.

Why this holds:

- Bellman-Ford guarantees that each relaxation step improves the shortest-path estimate for a vertex.
- After $n - 1$ iterations, any path between two vertices will have been fully considered because the longest simple path in a graph can have at most $n - 1$ edges.
- If there are no negative-weight cycles, these $n - 1$ iterations suffice to propagate the shortest-path distances throughout the graph.
- Since distances no longer decrease after $n - 1$ iterations, they represent the true shortest distances from s to all reachable

Proof: Bellman-Ford Correctly Computes Distances

- We want to show that if the graph has no negative cycles, then $d_{n-1}(v) = \delta(s, v)$ for all vertices v.
- We will prove by induction on k (the number of iterations of relaxation), that $d_k(v)$ is the minimum weight of a path from s to v that uses $\leq k$ edges.
- This shows that $d_{n-1}(v)$ is the minimum weight of a path from s to v that uses $\leq n-1$ edges.

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Correctly Computes Distances

- **Base Case ($k = 0$):**
  - At the start of the algorithm:

    $$d_0(s) = 0, \quad \text{since the distance from the source } s \text{ to itself is 0.}$$

  - For all $v \neq s$:

    $$d_0(v) = \infty, \quad \text{since no paths have been explored yet.}$$

  - This satisfies the definition: no path exists from s to v using 0 edges unless $v = s$.

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Correctly Computes Distances
Inductive Hypothesis:

- Suppose that after $k-1$ iterations: $d_{k-1}(u)$ is the minimum weight of any path from the source s to u that uses at most $k-1$ edges.
- This means that all shortest paths from s to any vertex u, which can be traversed with $k-1$ or fewer edges, have already been computed correctly by iteration $k-1$.

Goal for the k-th Iteration:

- Show that after k-th iteration, $d_k(v)$ is the minimum weight of any path from s to v that uses at most k edges.

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Correctly Computes Distances
Key Argument Using Path Structure:

1. Let P be the shortest simple path from s to v that uses at most k edges.
2. Since P is a simple path (no repeated vertices), it has at most $n-1$ edges, where n is the total number of vertices in the graph.
3. Break P into two parts:
   - Q: The path from s to u, which uses at most $k-1$ edges.
   - (u, v): The last edge on the path.
4. By the inductive hypothesis, the shortest path weight from s to u using at most $k-1$ edges is already stored in $d_{k-1}(u)$.
5. Therefore, the weight of Q is $d_{k-1}(u)$.

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Correctly Computes Distances
Relaxation in the k-th Iteration:

- In iteration k, the algorithm considers all edges (u, v) in the graph.
- When processing (u, v), the distance to v is updated using the relaxation formula:

$$d_k(v) = \min(d_{k-1}(v), d_{k-1}(u) + w(u, v)),$$

  where $w(u, v)$ is the weight of the edge from u to v.
- This formula ensures:
  - Either the shortest path to v remains the same as in iteration $k - 1$ (if adding (u, v) doesn't improve it).
  - Or the shortest path to v is updated to include the edge (u, v).

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Correctly Computes Distances
Why the Update Works:

- **Case 1:** The shortest path to v uses at most $k-1$ edges.
  - In this case, $d_{k-1}(v)$ is already correct because the shortest path from s to v doesn't need more than $k-1$ edges.
  - The relaxation step won't change $d_k(v)$ because:

$$d_{k-1}(v) \leq d_{k-1}(u) + w(u,v).$$

- **Case 2:** The shortest path to v uses exactly k edges.
  - In this case, the shortest path to v must include an edge (u,v), where the subpath to u is the shortest path using at most $k-1$ edges.
  - The relaxation step updates $d_k(v)$ to reflect this new shortest path:

$$d_k(v) = d_{k-1}(u) + w(u,v).$$

- Thus, $d_k(v)$ becomes the minimum weight of any path from s to v that uses at most k edges.

Bellman-Ford Algorithm: Proof of Correctness

Proof: Bellman-Ford Correctly Computes Distances
Conclusion of the Inductive Step:

- After the k-th iteration:
  - For all vertices v, $d_k(v)$ correctly represents the minimum weight of a path from s to v using at most k edges.
- By repeating this process up to $n - 1$ iterations (where n is the number of vertices), the algorithm computes the shortest paths for all vertices, as the longest simple path can have at most $n - 1$ edges.