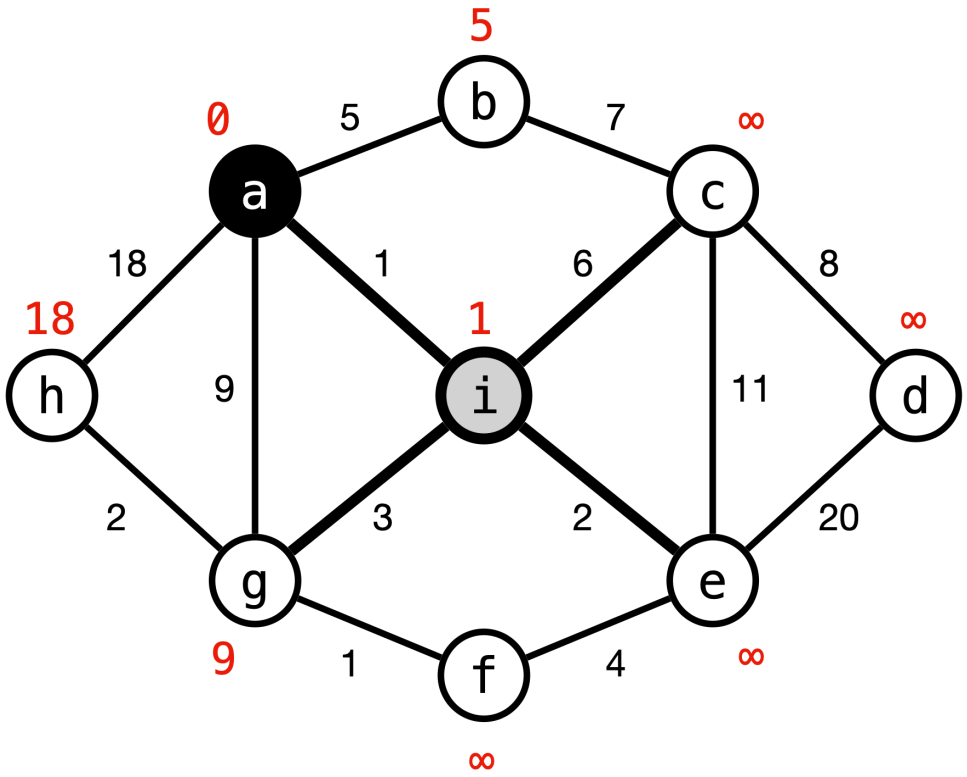
Lecture 15 - Bellman-Ford Algorithm and Dynamic Programming

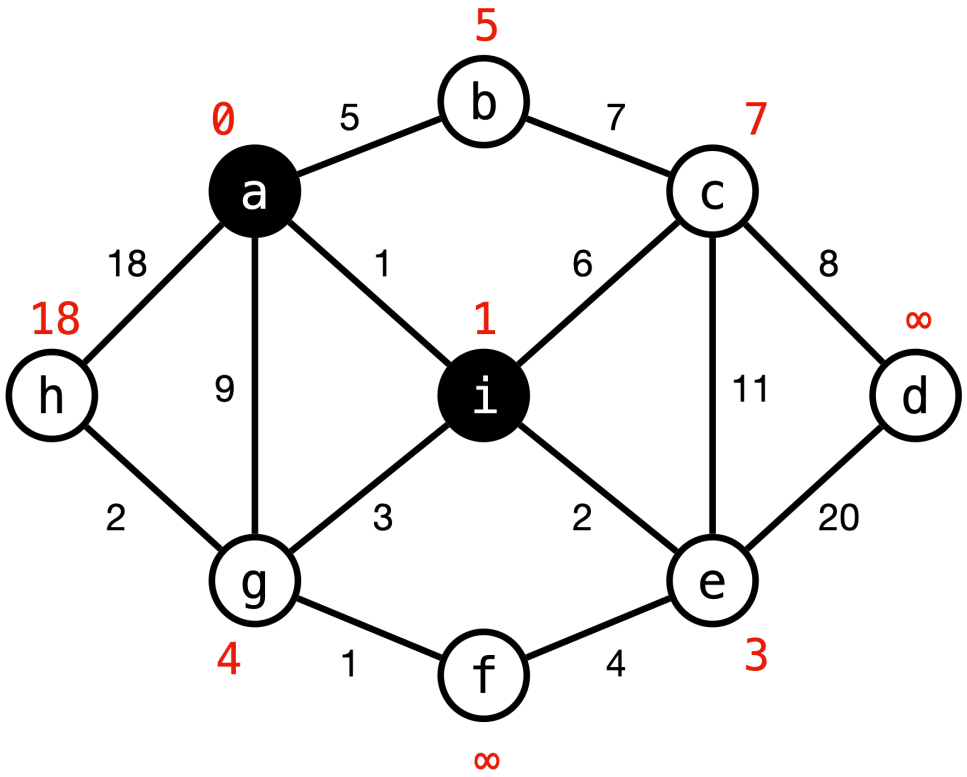
Fall 2025, Korea University

Instructor: Gabin An (gabin_an@korea.ac.kr)

Recap: SSSP and Dijkstra's Algorithm



Pick i



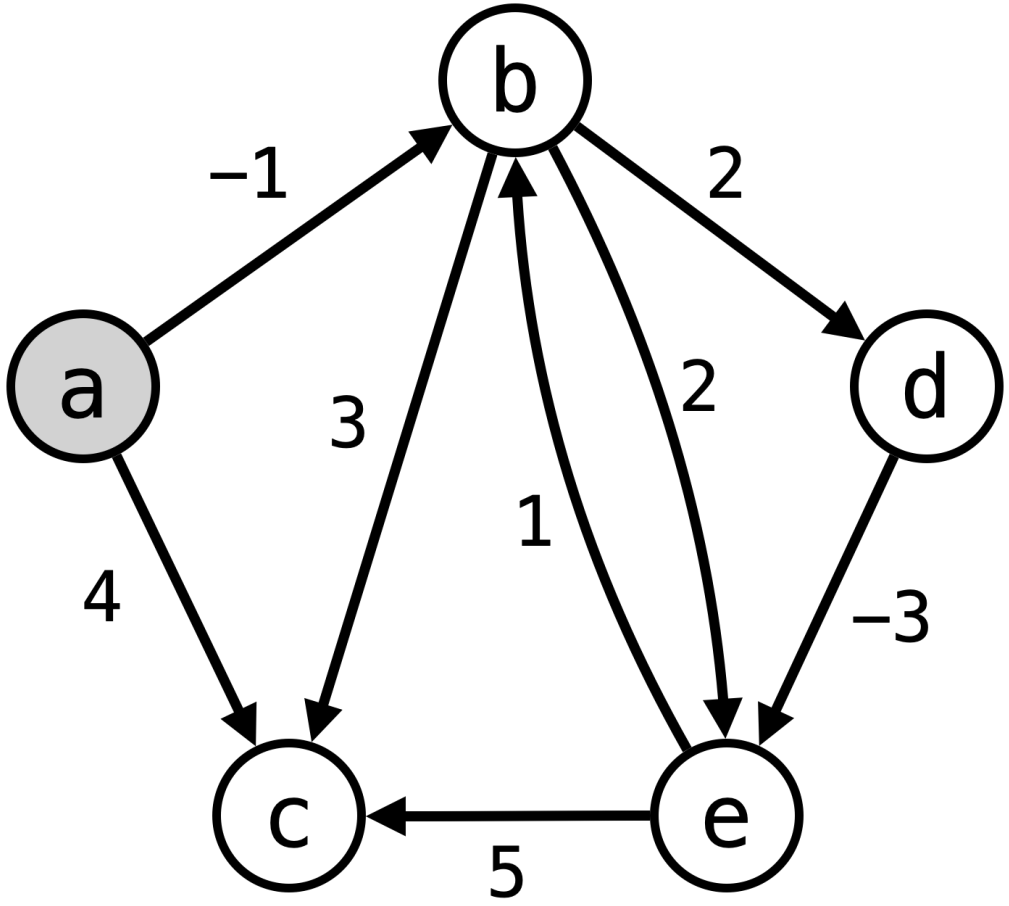
Relax its edges & move i from F to D

Recap: Limitation of Dijkstra's Algorithm

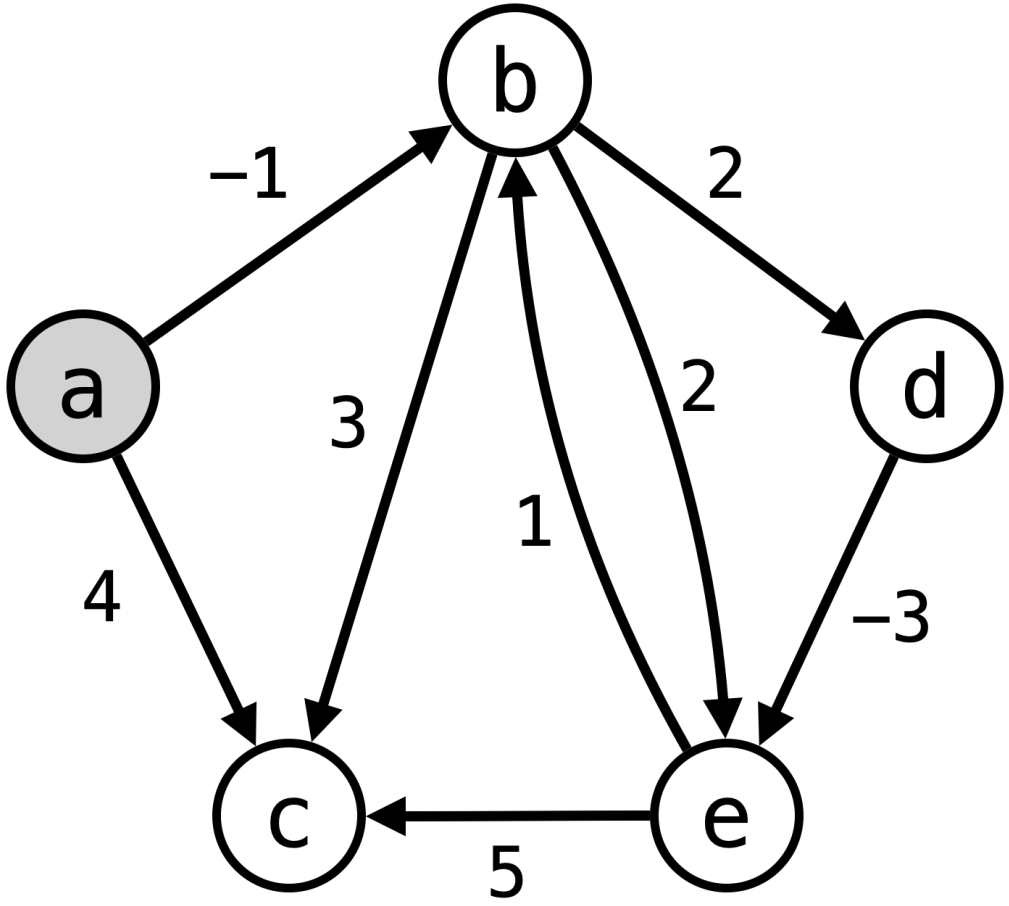
- Requires **nonnegative edge weights**
- Not robust to **graph updates** (edge weight changes)

➡️ For graphs with negative weights, we use **Bellman-Ford**, which we will learn today!

Example: Negative Edge Weights



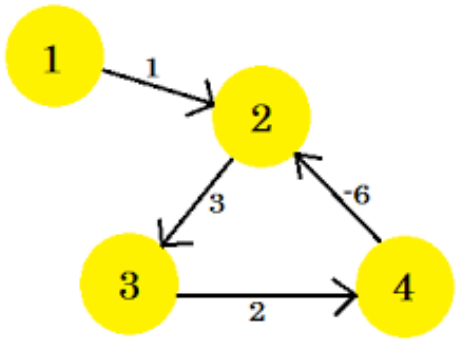
Example: Negative Edge Weights



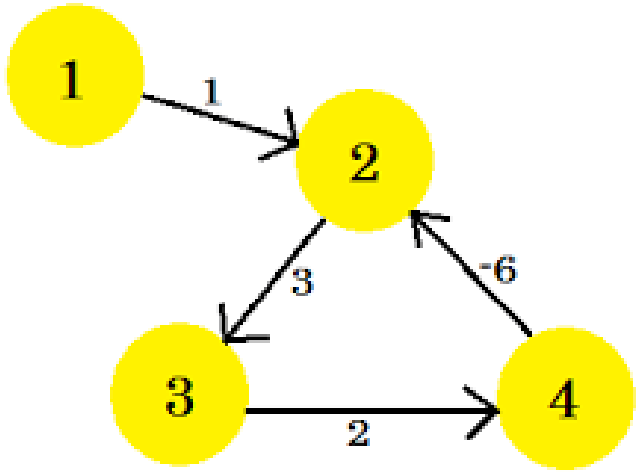
target	shortest path	total cost
b	a → b	-1
c	a → b → c	2
d	a → b → d	1
e	a → b → d → e	-2

Negative Edge Weights and Negative Cycles

- Negative weights are fine for Bellman-Ford.
- However, the presence of a **negative cycle**, a cycle whose total edge weight is negative, causes a problem:
  - Each time we traverse the cycle, the path cost decreases.
  - As a result, the "shortest path" to any node on such a cycle is not well defined (it tends toward negative infinity). *E.g., What is the shortest path from 1 to 2?*



Example of Negative Cycle



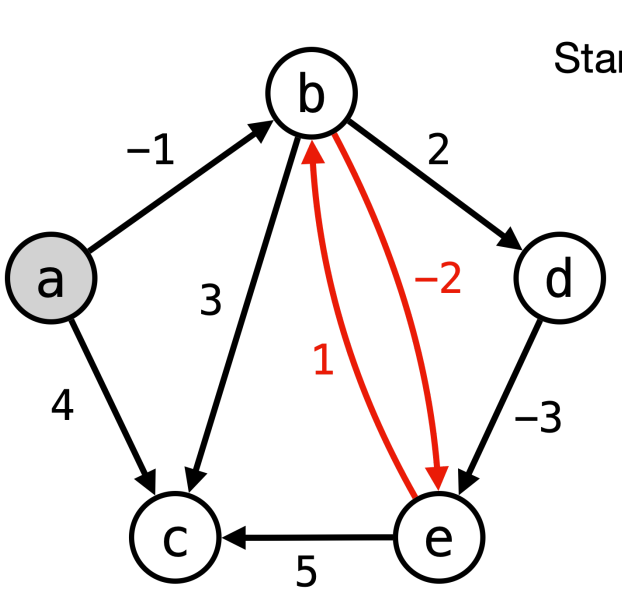
- `1 → 2` : `1`
- `1 → 2 → 3 → 4 → 2` : `0` 😟
- `1 → 2 → 3 → 4 → 2 → 3 → 4 → 2` : `−1` 😱

> The shortest path would be of infinite length and is not well-defined!

Bellman-Ford Algorithm

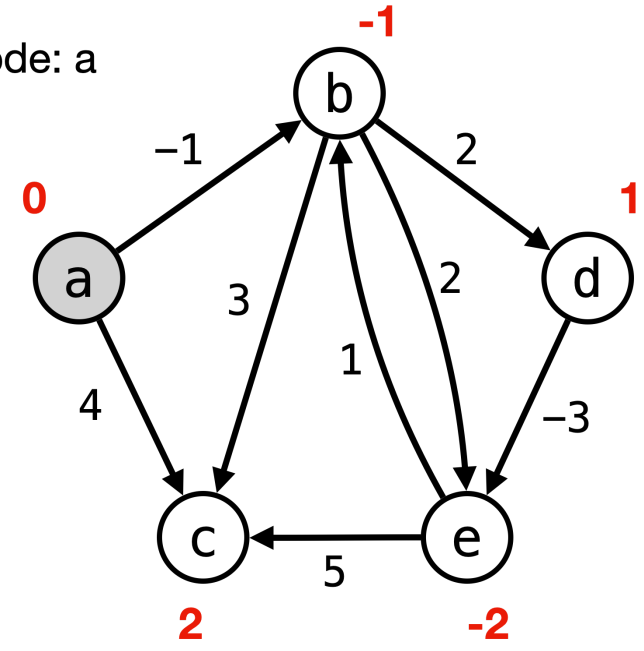
Input: Graph $G = (V, E)$ with possibly **negative weights**, and a source vervex s

Output: ① Detects **negative cycles** (if reachable from s), or ② computes shortest paths $d(s, v)$ for all $v \in V$ (**SSSP**)



Negative Cycle Detected!

Starting node: a



Bellman-Ford Algorithm

The algorithm computes the shortest paths **in a bottom-up manner**:

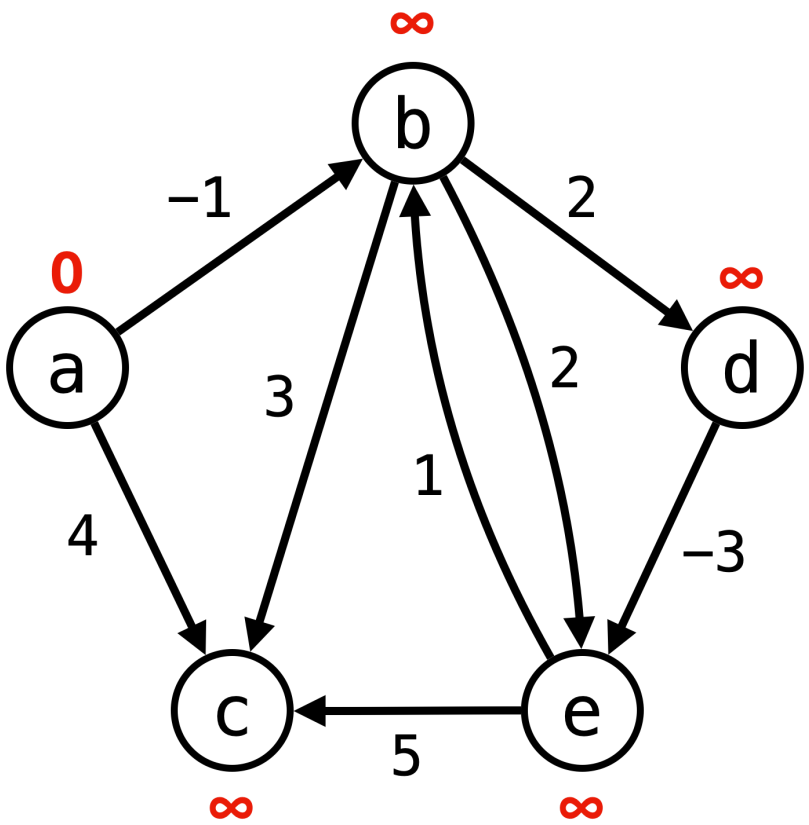
1. It first computes the shortest paths that use **at most 0 edges**.
2. Then it computes the shortest paths that use **at most 1 edge**.
3. Then **at most 2 edges**, and so on.
4. After the i**-th iteration**, we have the shortest paths that use **at most** i **edges**.

💡 Since any **simple path** can have **at most** $|V| - 1$ **edges**, after performing $|V| - 1$ **iterations**, all shortest paths are guaranteed to be found.

Bellman-Ford Algorithm

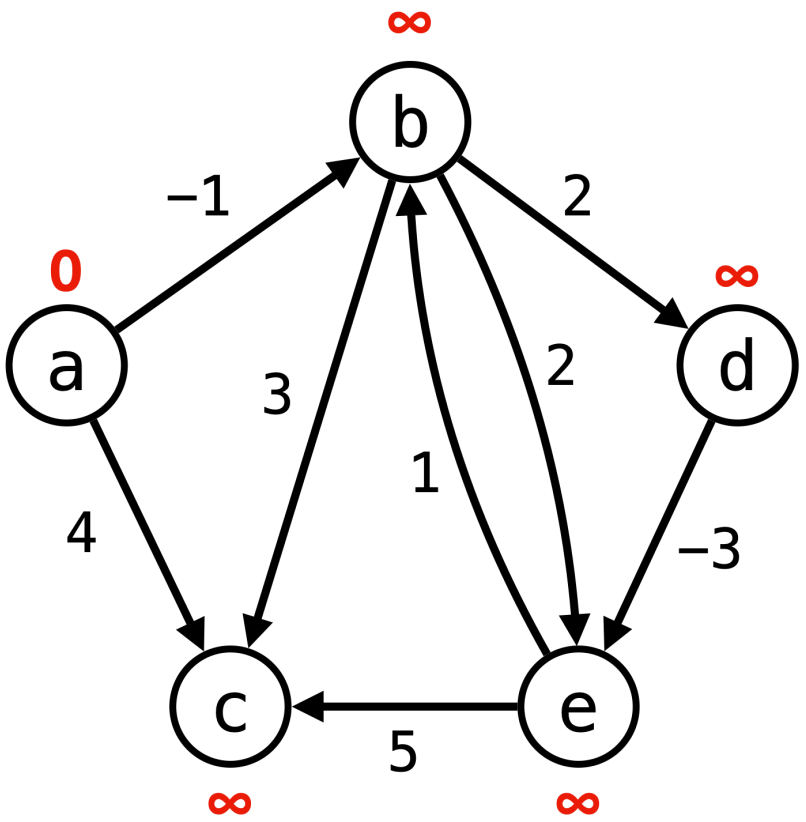
1. Initialize $d[s] = 0$, others ∞.
2. **Repeat $|V| - 1$ times:**
   Relax **every edge** $(u, v) \in E$: $d[v] \leftarrow \min(d[v], d[u] + w(u, v))$
3. Extra pass:
   If any edge (u, v) can still be relaxed, i.e., $d[v] > d[u] + w(u, v)$
   → **negative cycle detected** 🔁.

Example 1: Negative Cycle Does Not Exist



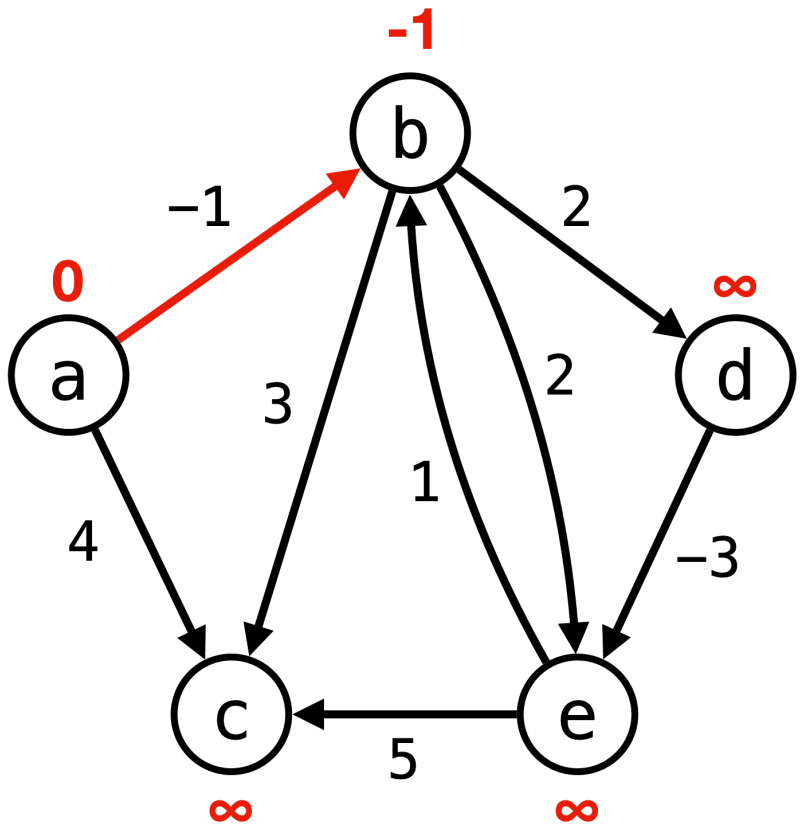
1. Initialize d[s] = 0, others ∞

Example 1: Negative Cycle Does Not Exist



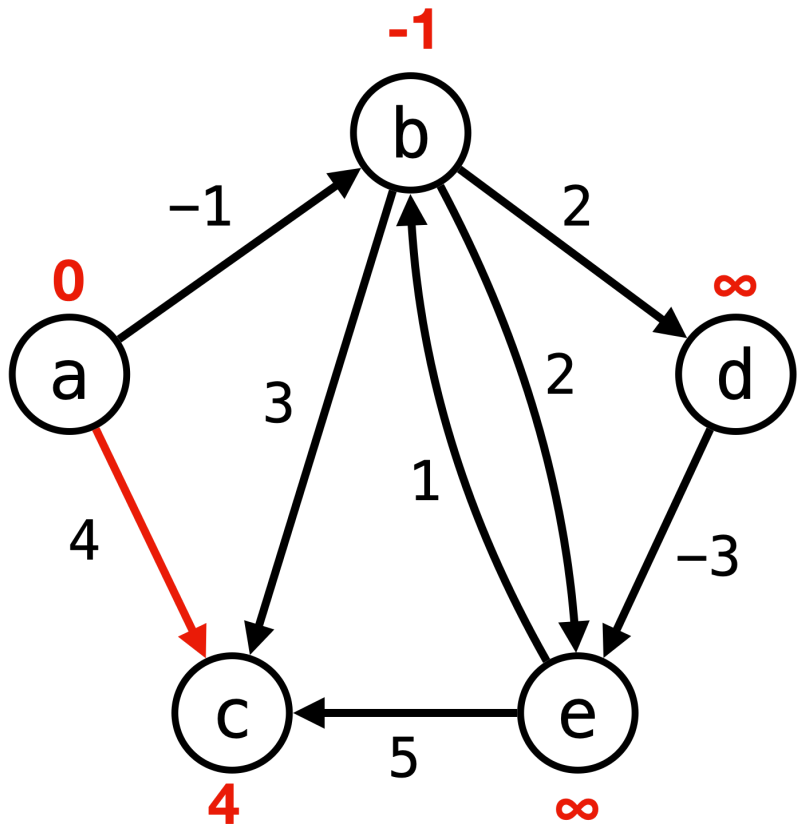
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



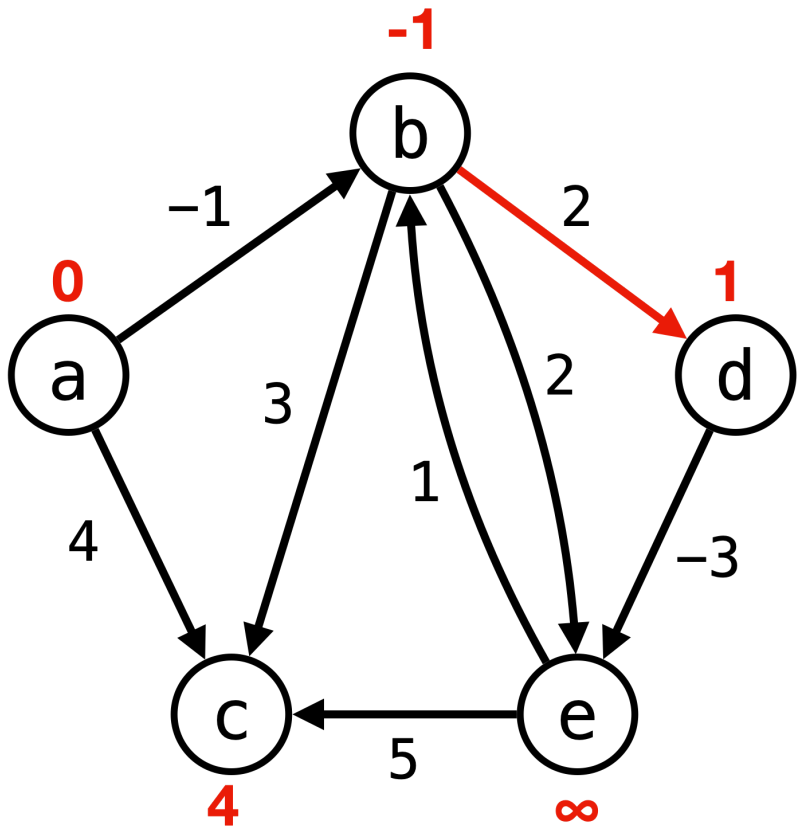
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



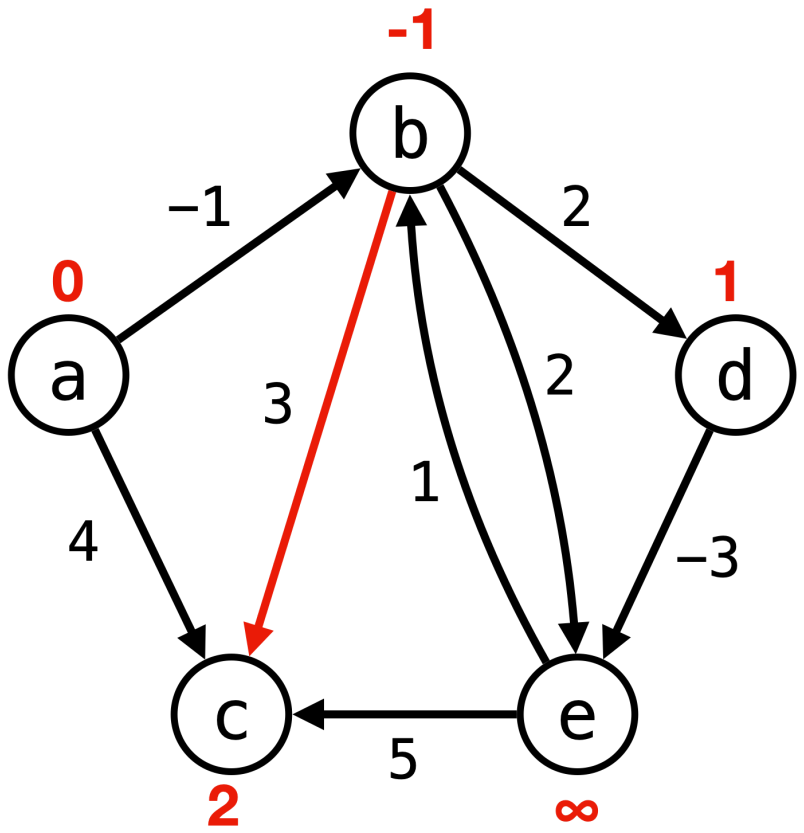
2. Relax every edge - **1 out of 4 reps**
(a, b), **(a, c)**, (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



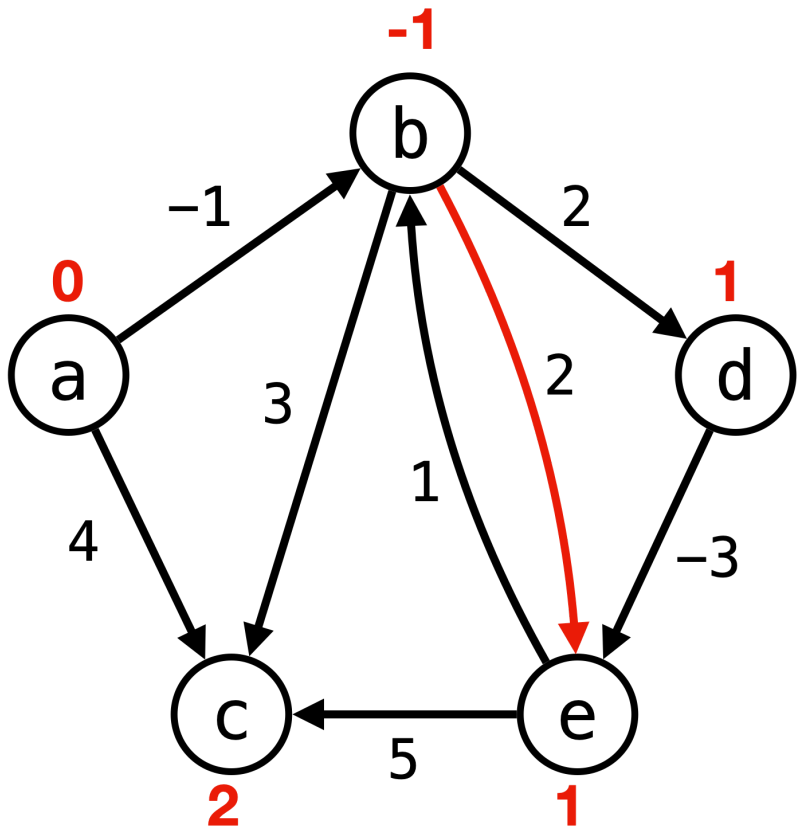
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), **(b, d)**, (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



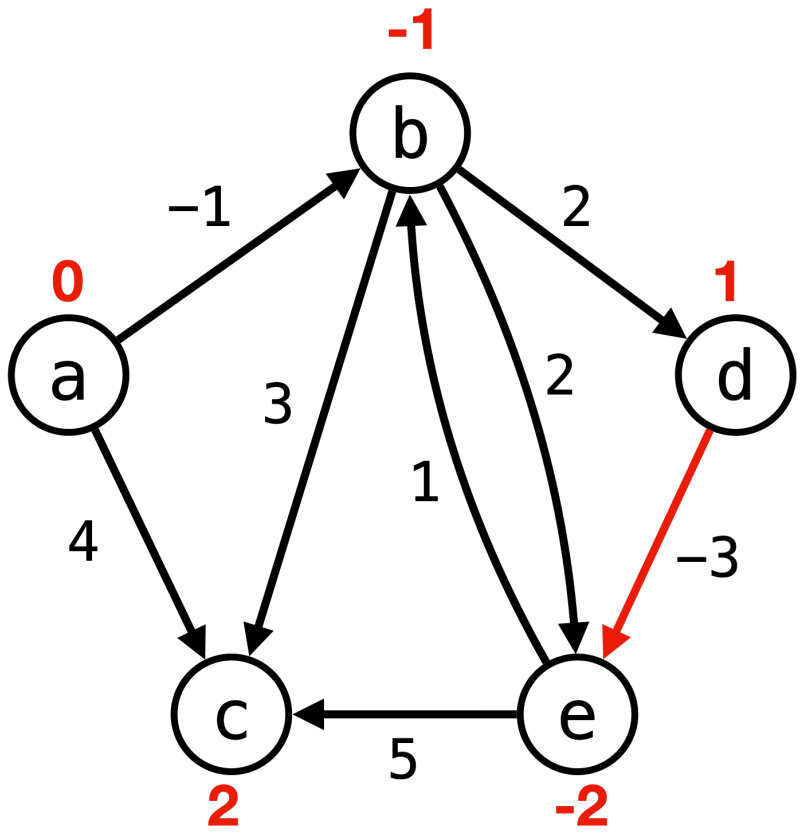
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), **(b, c)**, (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



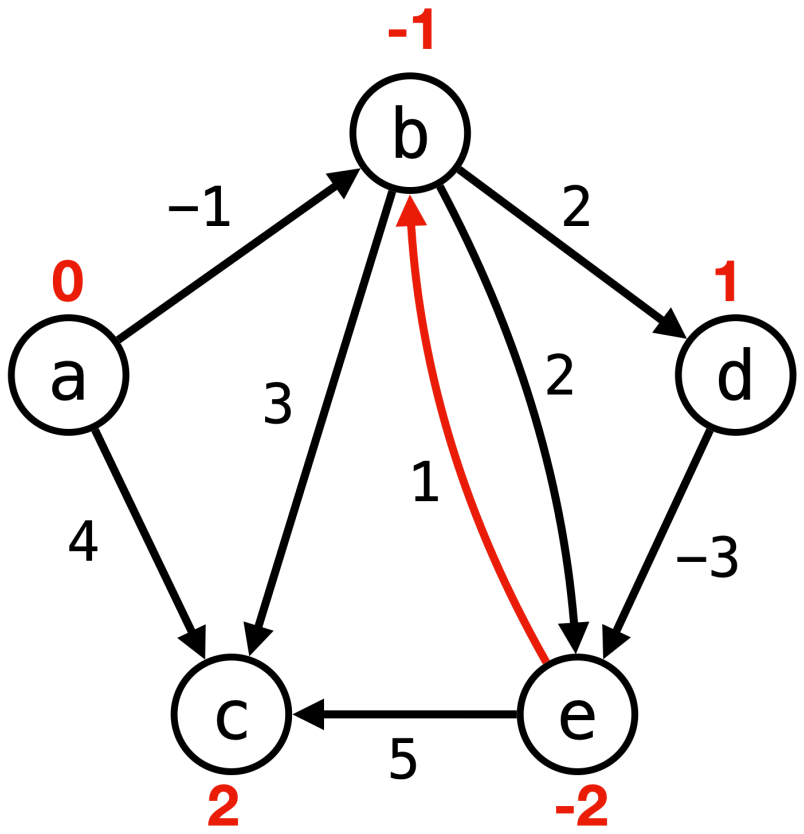
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), **(b, e)**, (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



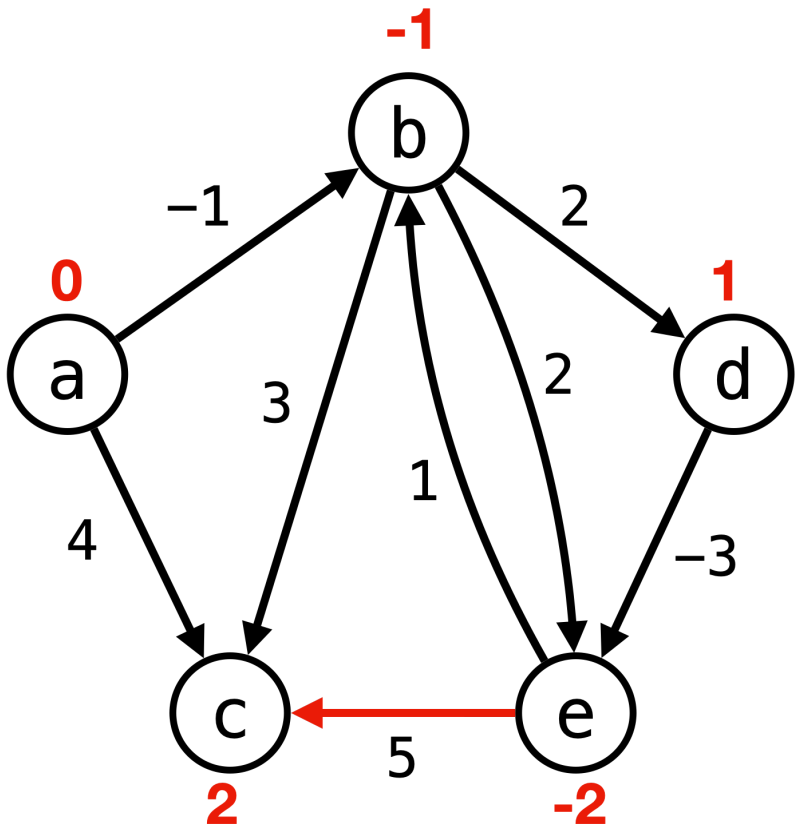
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), **(d, e)**, (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



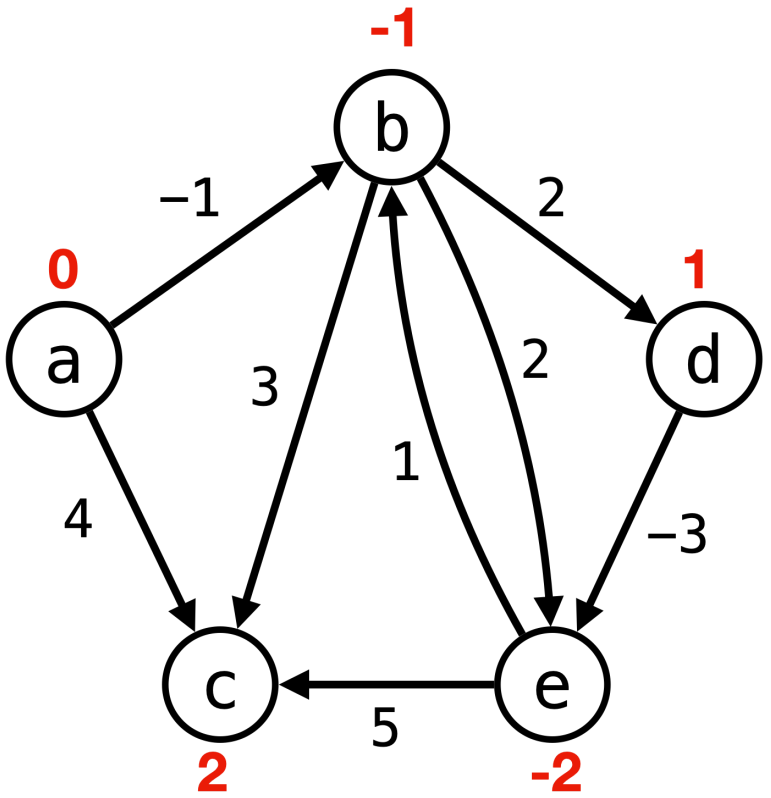
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), **(e, b)**, (e, c)

Example 1: Negative Cycle Does Not Exist



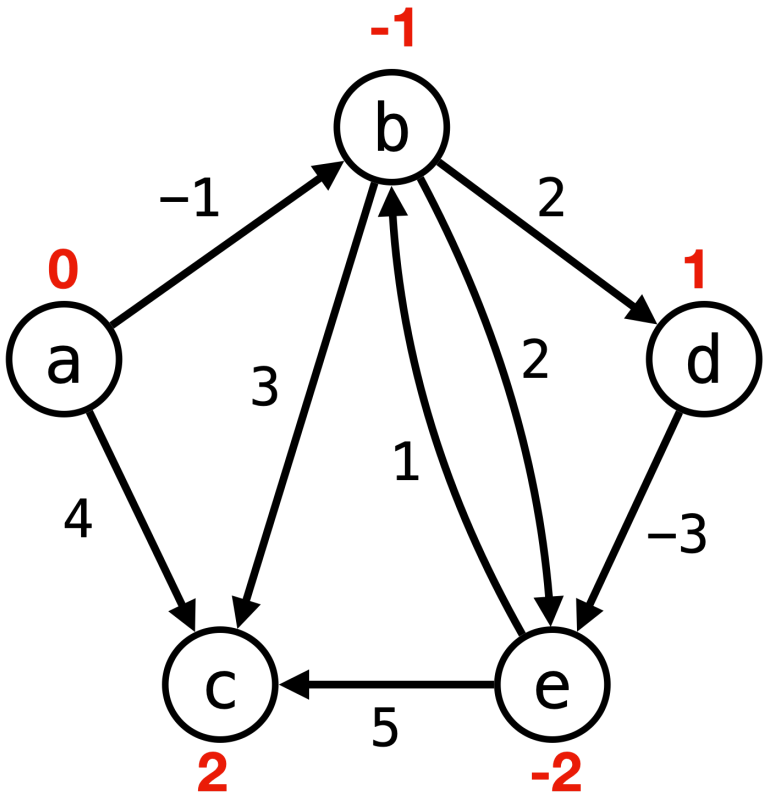
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), **(e, c)**

Example 1: Negative Cycle Does Not Exist



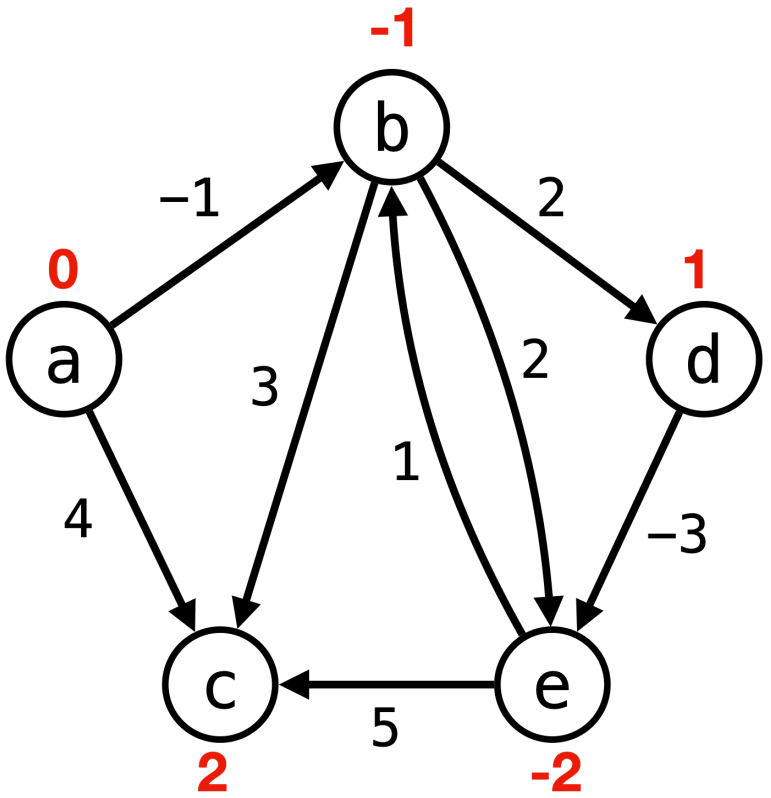
2. Relax every edge - **2 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



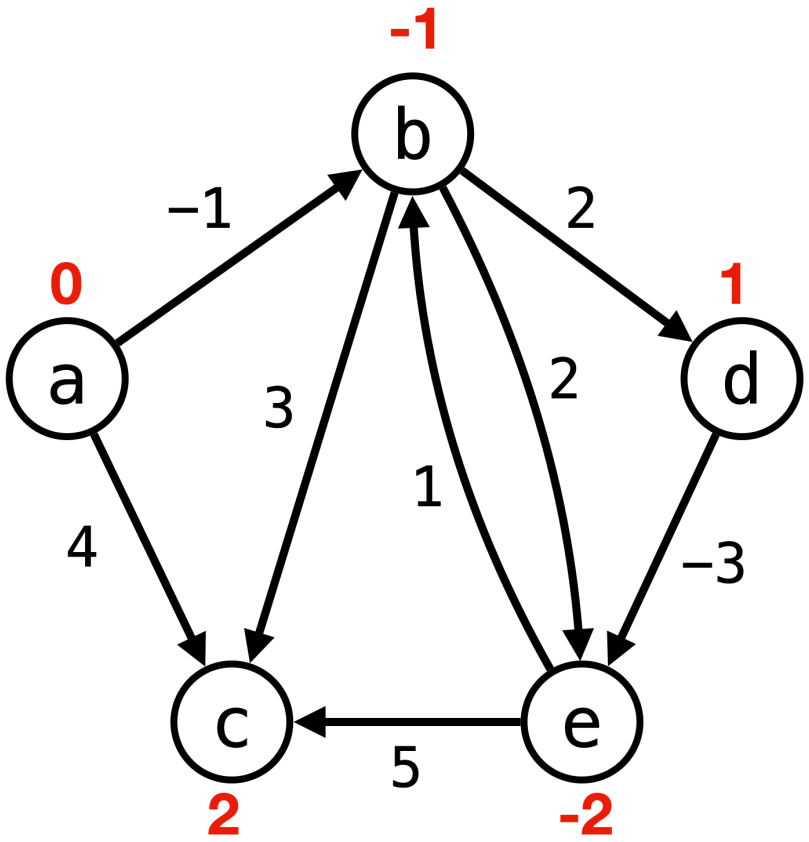
2. Relax every edge - **3 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



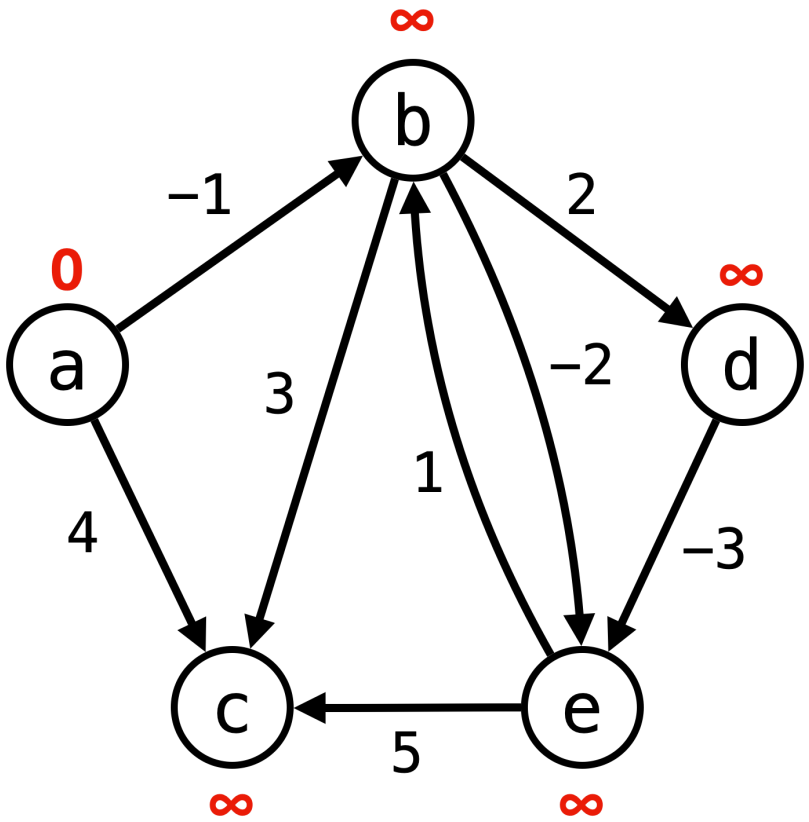
2. Relax every edge - **4 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 1: Negative Cycle Does Not Exist



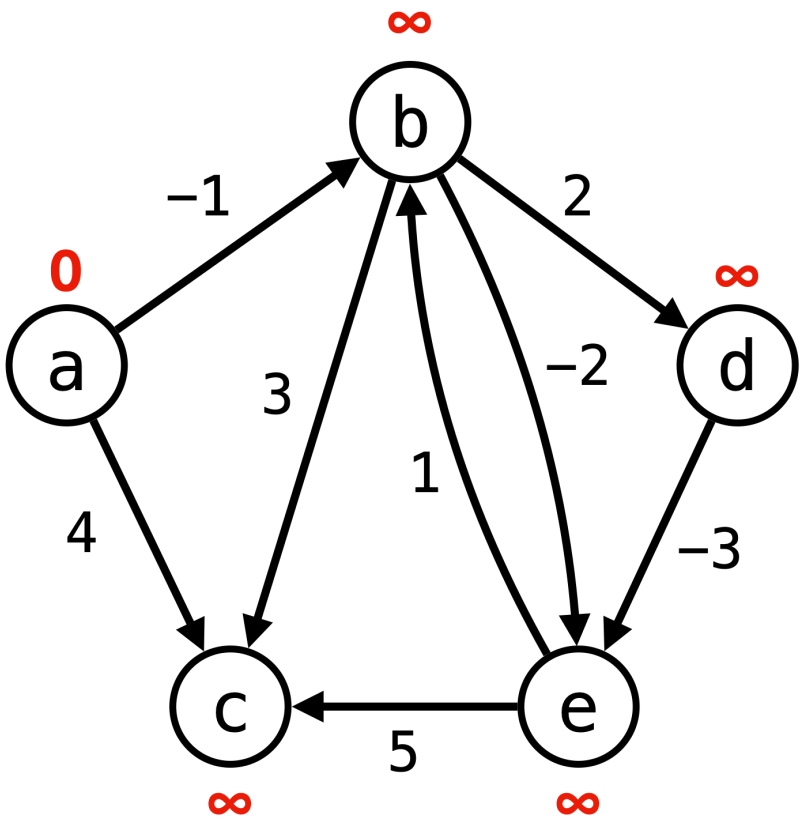
3. No edge can be relaxed.
Return the distance estimates.

Example 2: Negative Cycle Exists



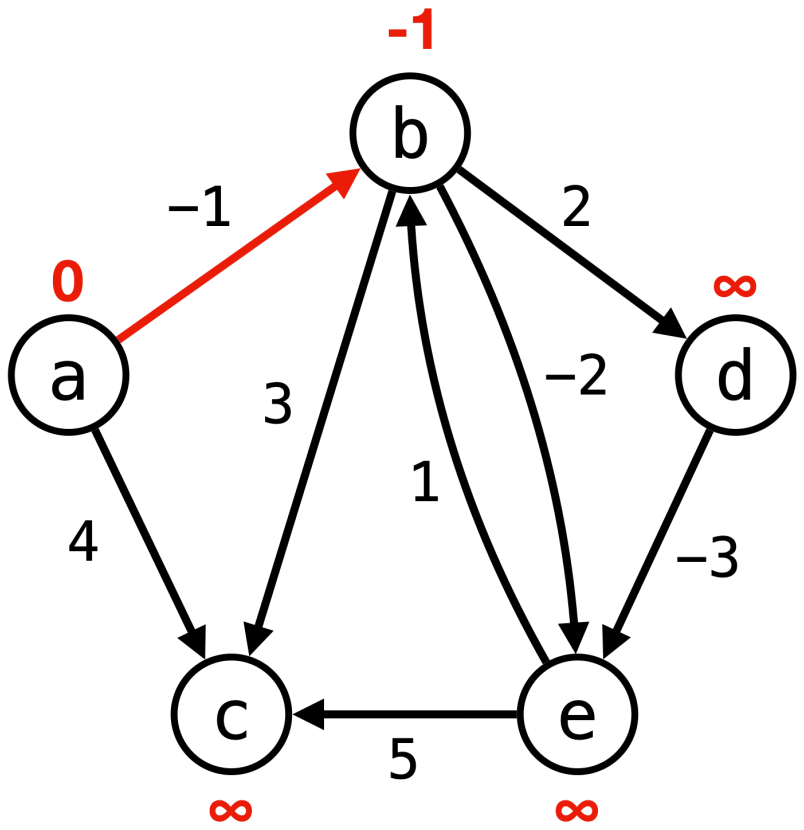
1. Initialize d[s] = 0, others ∞

Example 2: Negative Cycle Exists



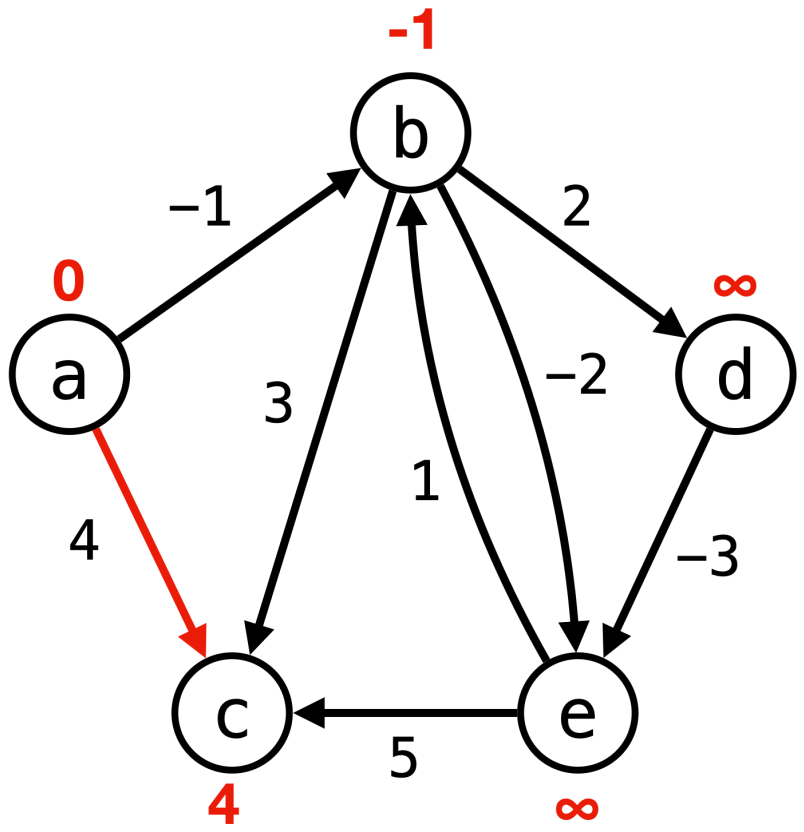
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



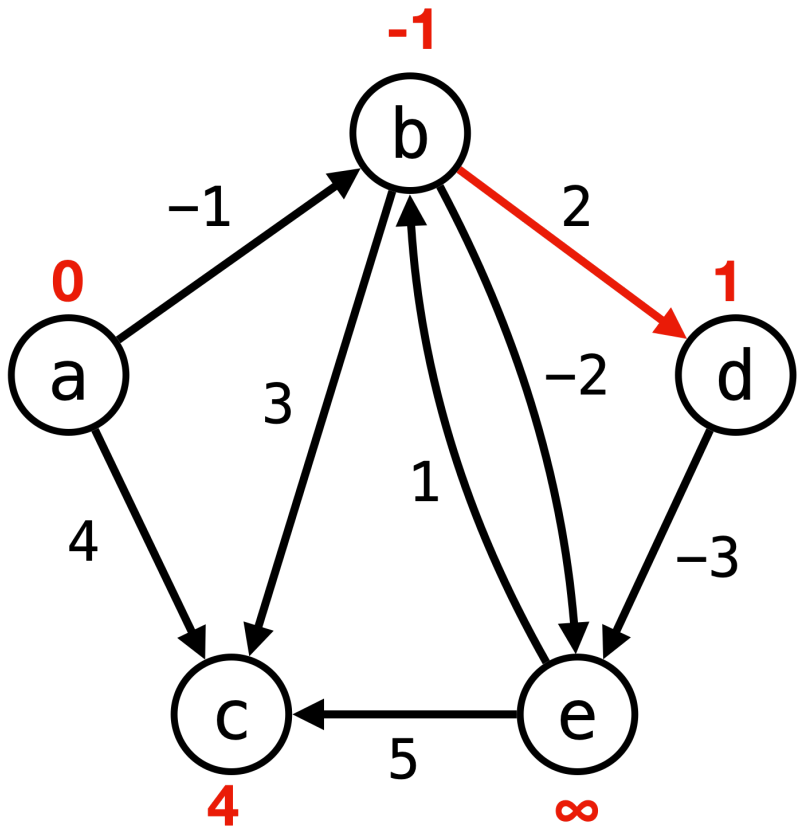
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



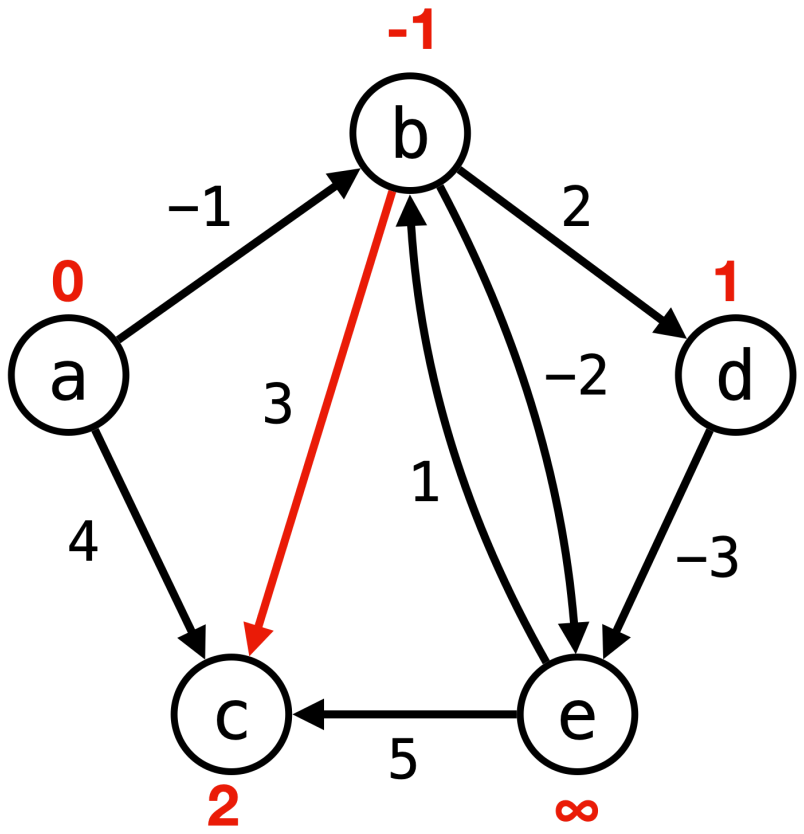
2. Relax every edge - **1 out of 4 reps**
(a, b), **(a, c)**, (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



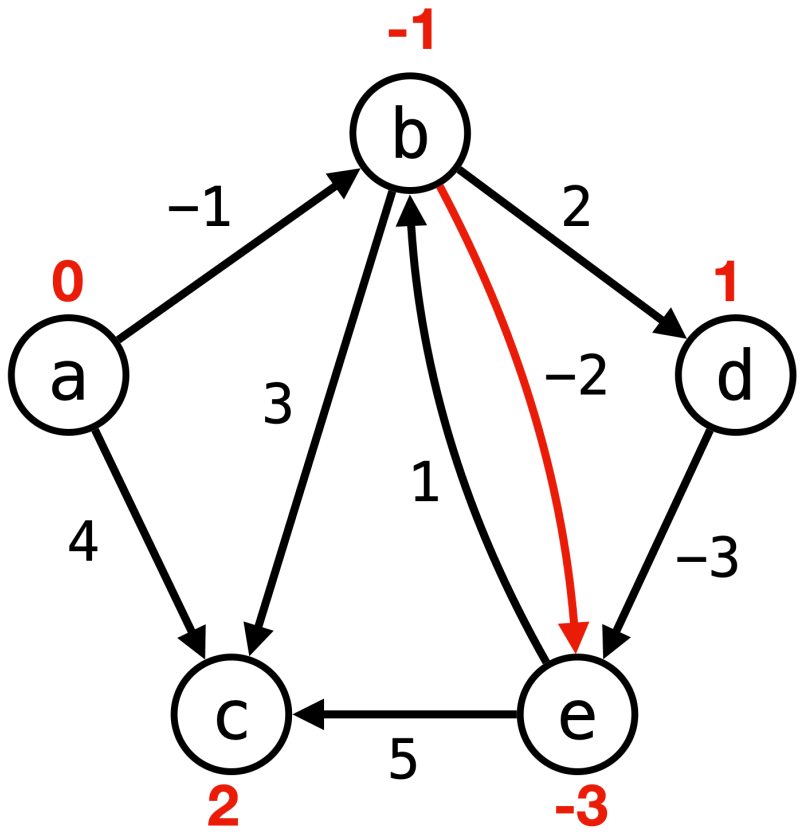
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), **(b, d)**, (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



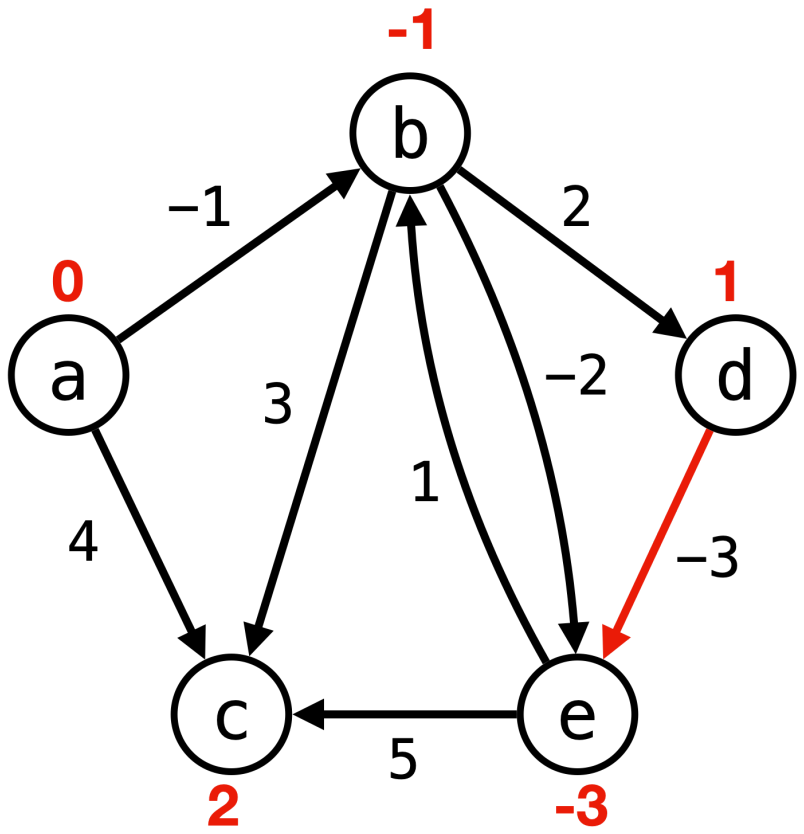
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), **(b, c)**, (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



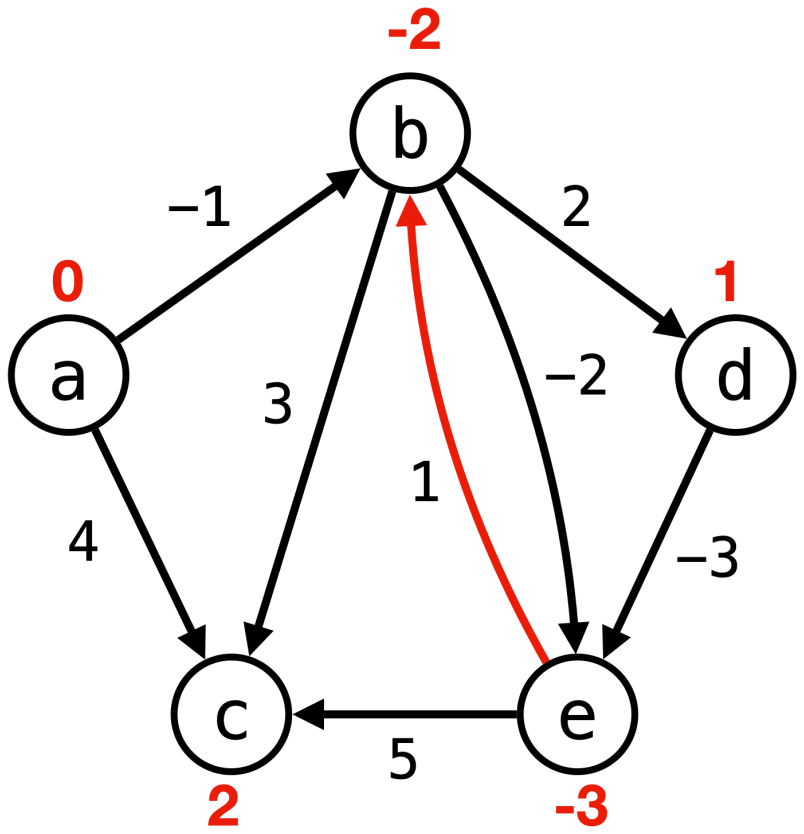
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), **(b, e)**, (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



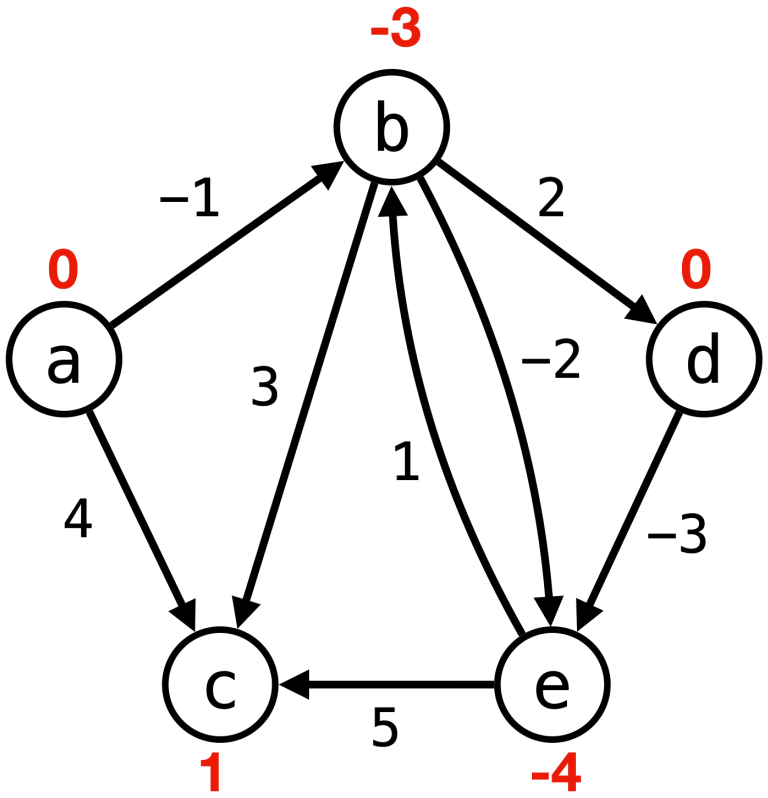
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), **(d, e)**, (e, b), (e, c)

Example 2: Negative Cycle Exists



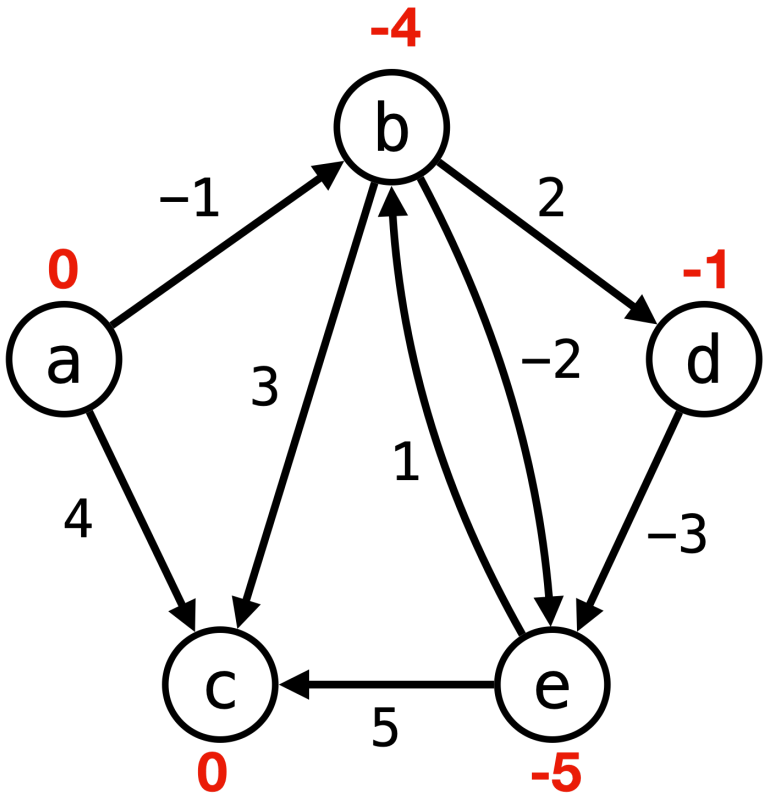
2. Relax every edge - **1 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), **(e, b)**, (e, c)

Example 2: Negative Cycle Exists



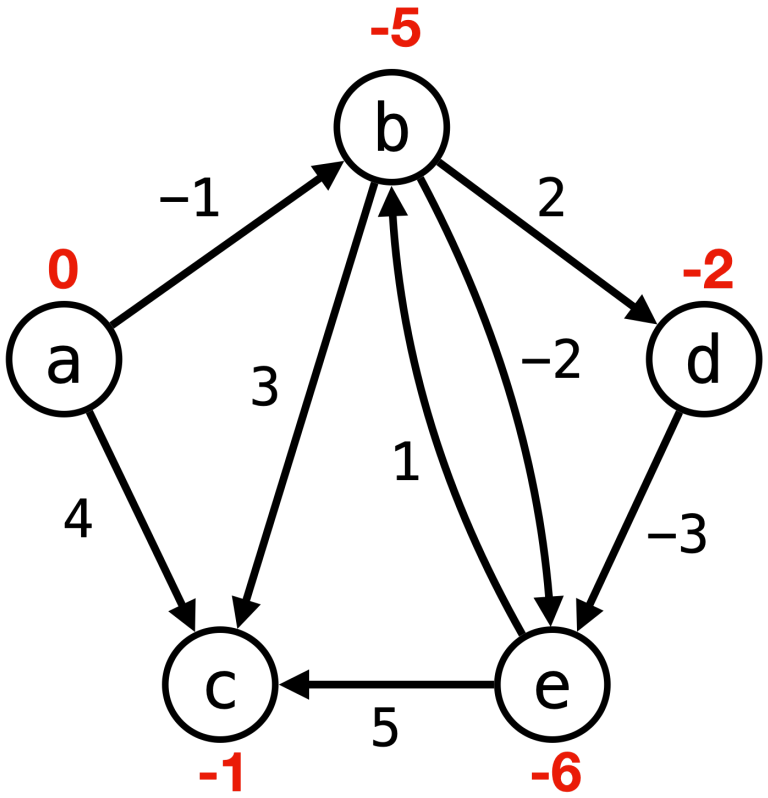
2. Relax every edge - **2 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



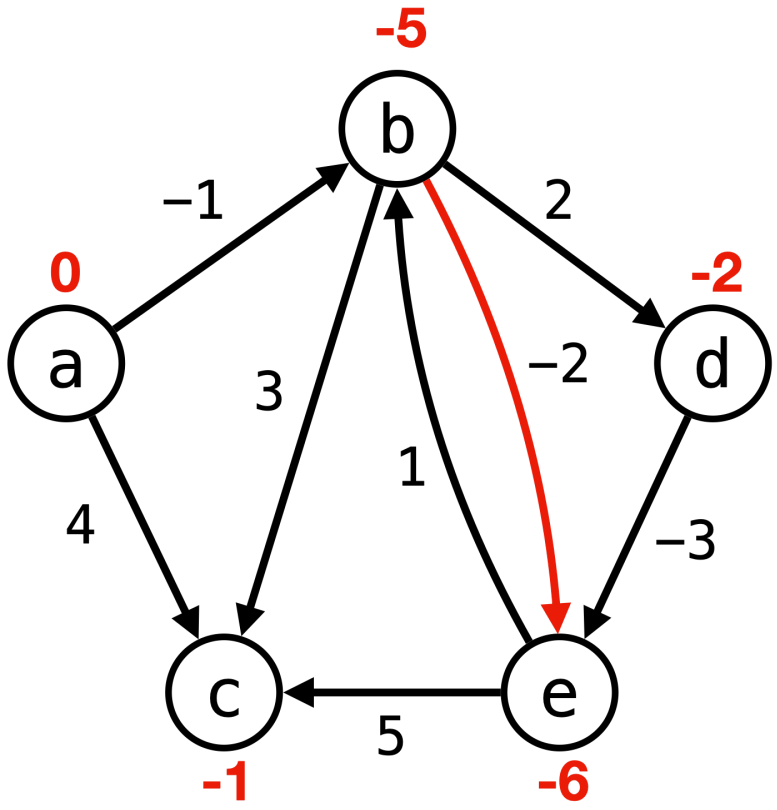
2. Relax every edge - **3 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



2. Relax every edge - **4 out of 4 reps**
(a, b), (a, c), (b, d), (b, c), (b, e), (d, e), (e, b), (e, c)

Example 2: Negative Cycle Exists



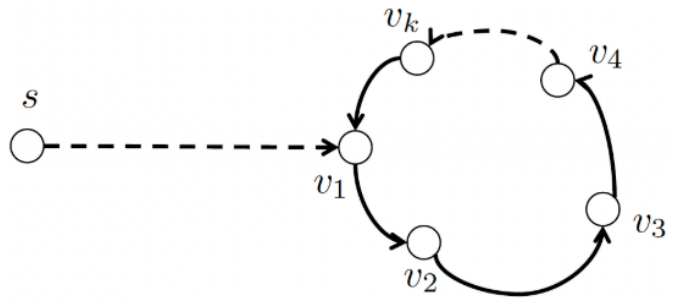
3. The edge (b, e) can still be relaxed!
Negative Cycle Detected

Correctness of Bellman-Ford

The Bellman-Ford algorithm guarantees one of two outcomes:

1. If there is a **negative cycle** reachable from the source → it reports “Negative Cycle”.
2. Otherwise, it computes the correct shortest path distances $d(s, v)$ for all $v \in V$.

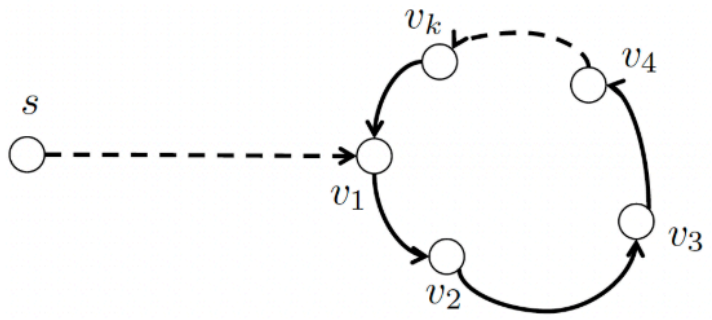
Claim 1: If there is a negative cycle reachable from the source, it reports "Negative Cycle".



- Let the negative cycle be $C = (v_1, \ldots, v_k)$ with

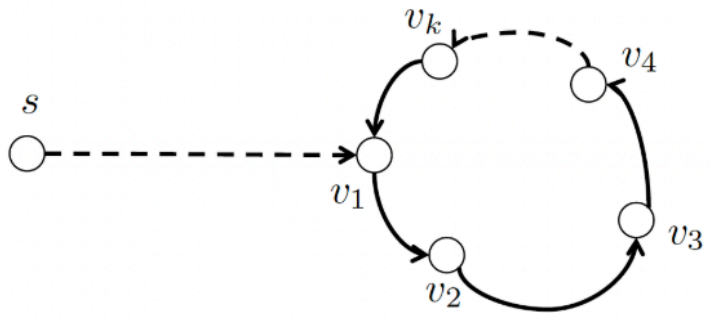
$$\sum_{i=1}^{k} w(v_i, v_{i+1}) < 0, \quad v_{k+1} = v_1$$

- Since C is reachable from s, there exists a path from s to v_1 and to every node on C.
- For each node on C, there exists a simple path (no cycles) from s with **at most $n-1$ edges**.



- For the sake of contradiction, suppose Bellman-Ford does not report "Negative Cycle," even though C exists.
  - After $n-1$ iterations, $d[v_i]$ will be some finite number $< \infty$ for $i = 1, \ldots, k$.
  - Since no negative cycle was reported (i.e., no edge can be released):

$$d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1}), \quad i = 1, \ldots, k$$



- Since no cycle is reported,

$$d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1}), \quad i = 1, \ldots, k$$

- Summing over the cycle C, we obtain

$$\sum_{i=1}^{k} d[v_{i+1}] \leq \sum_{i=1}^{k} d[v_i] + \sum_{i=1}^{k} w(v_i, v_{i+1}) \quad \rightarrow \quad 0 \leq \sum_{i=1}^{k} w(v_i, v_{i+1})$$

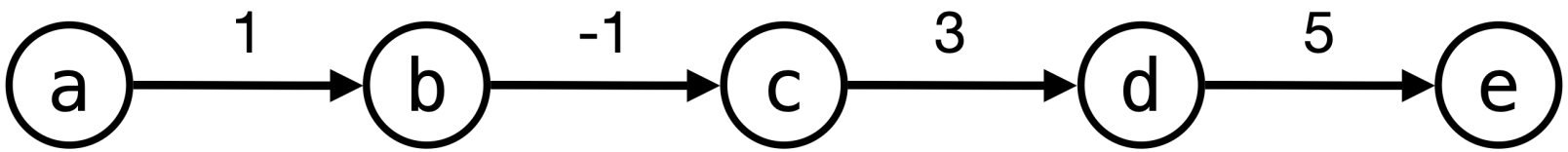
  This contradicts that C is a negative cycle.

- Therefore, Bellman-Ford must report “Negative Cycle”! ✅

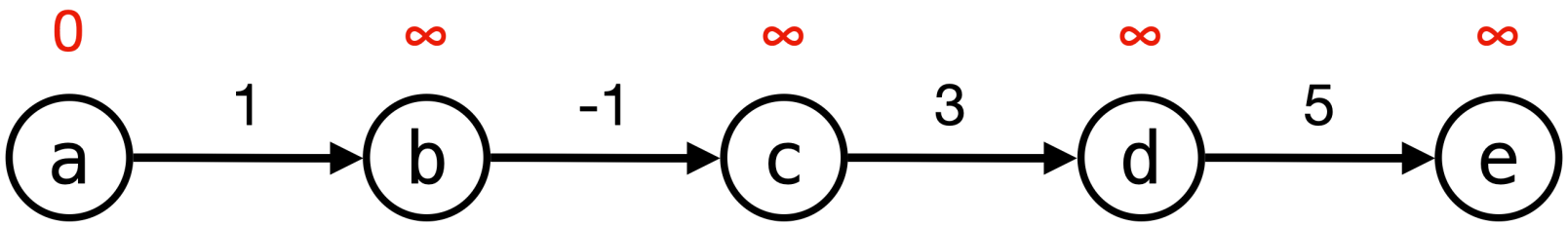
Correctness of Bellman-Ford

The Bellman-Ford algorithm guarantees one of two outcomes:

1. If there is a negative cycle reachable from the source → it reports “Negative Cycle”. ✅
2. Otherwise, it computes the **correct shortest path distances** $d(s, v)$ for all $v \in V$. ⬅️

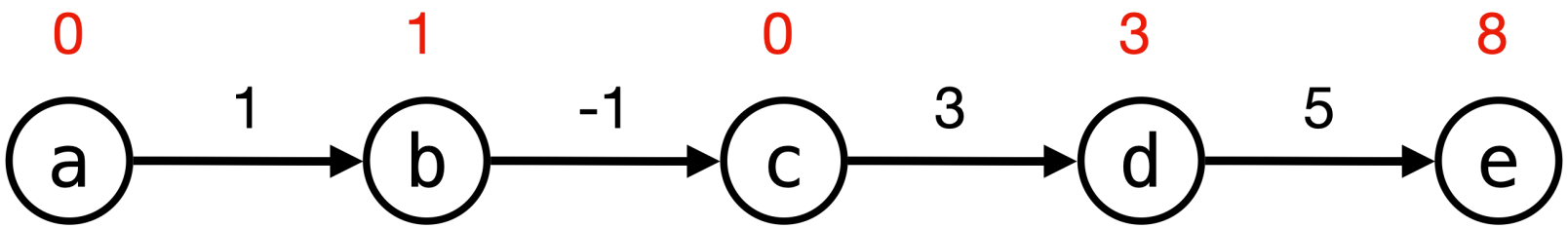


Edge Relaxation Order: (a, b), (b, c), (c, d), (d, e)



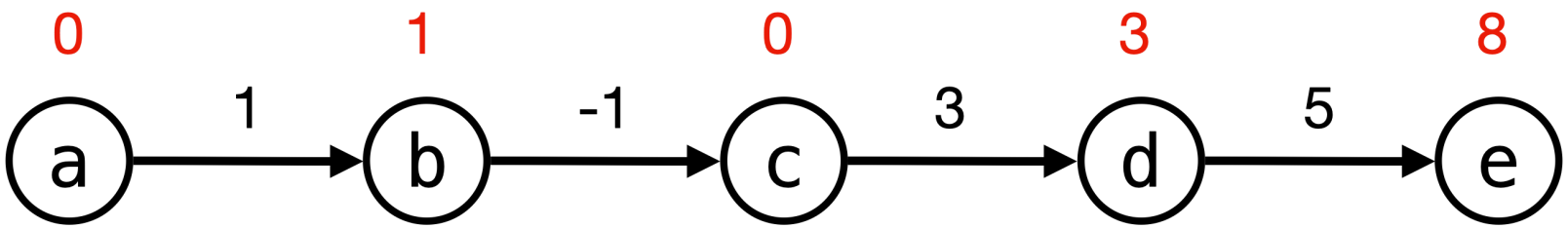
Edge Relaxation Order: (a, b), (b, c), (c, d), (d, e)

After 1st iteration



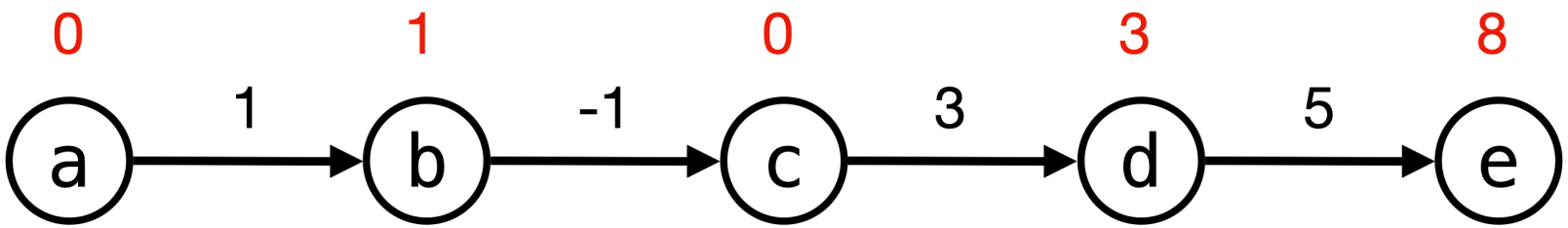
Edge Relaxation Order: (a, b), (b, c), (c, d), (d, e)

After 2nd iteration



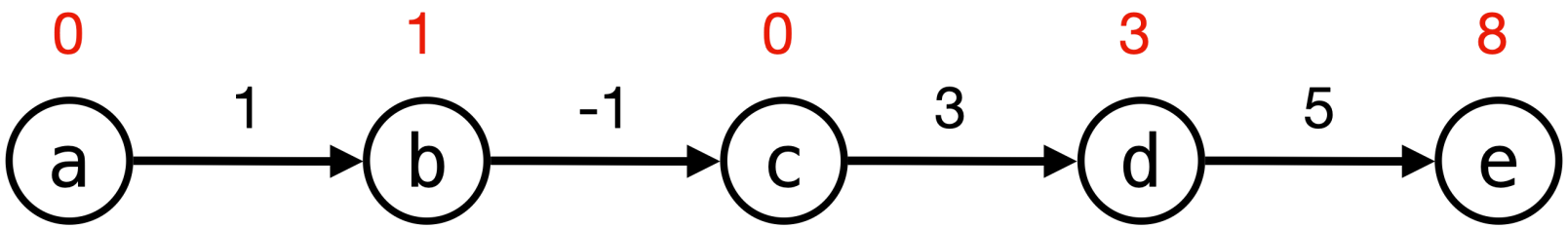
Edge Relaxation Order: (a, b), (b, c), (c, d), (d, e)

After 3rd iteration

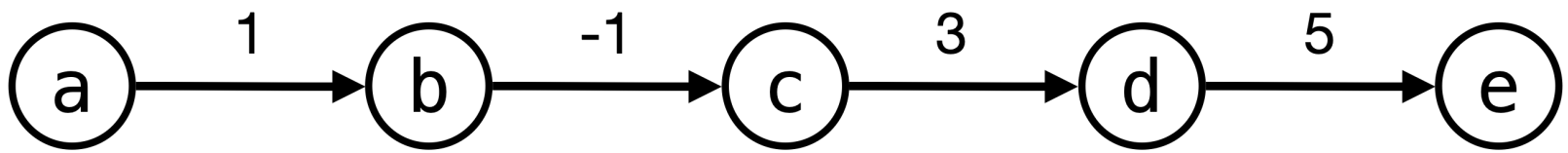


Edge Relaxation Order: (a, b), (b, c), (c, d), (d, e)

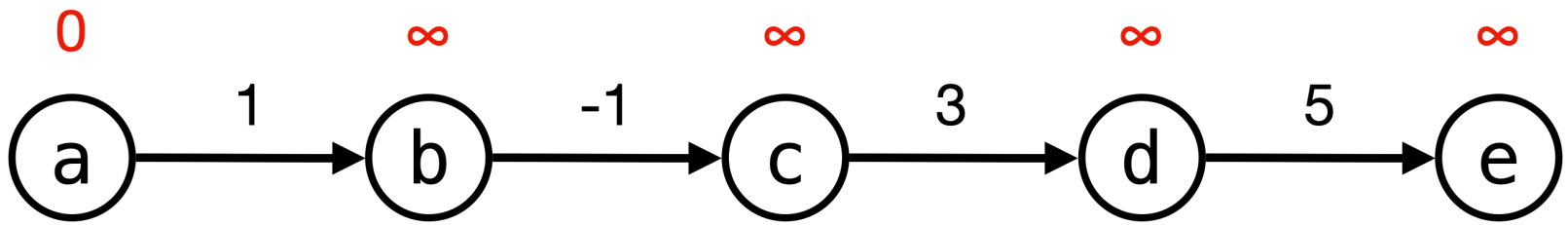
After 4th iteration



Edge Relaxation Order: (a, b), (b, c), (c, d), (d, e)

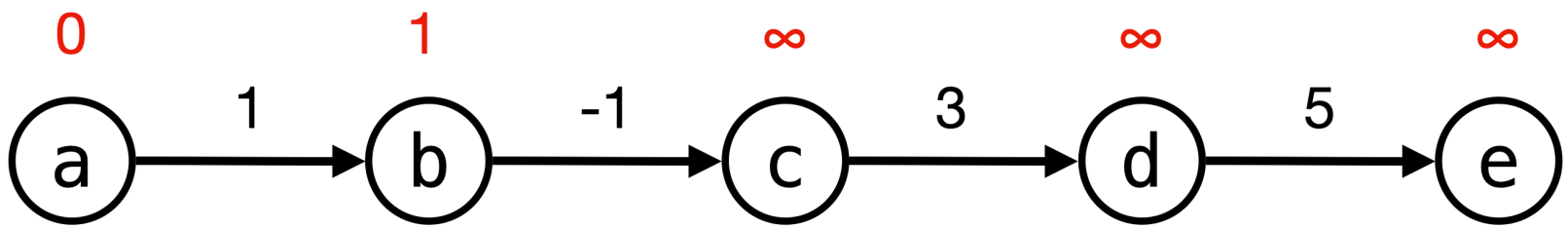


Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)



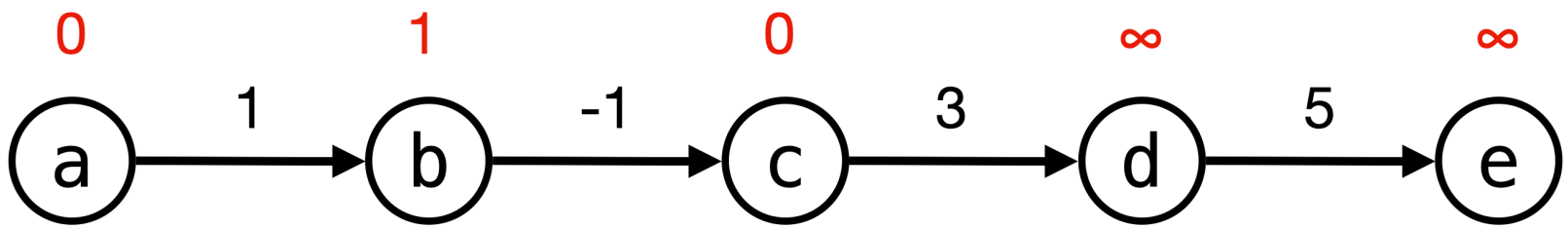
Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

After 1st iteration



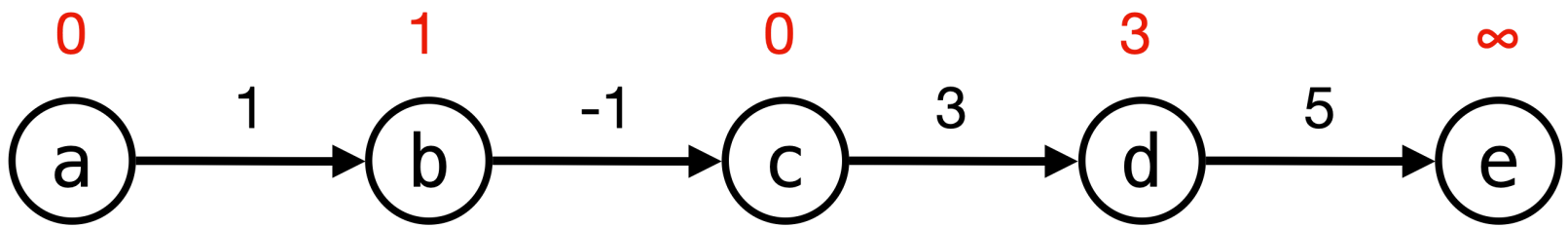
Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

After 2nd iteration



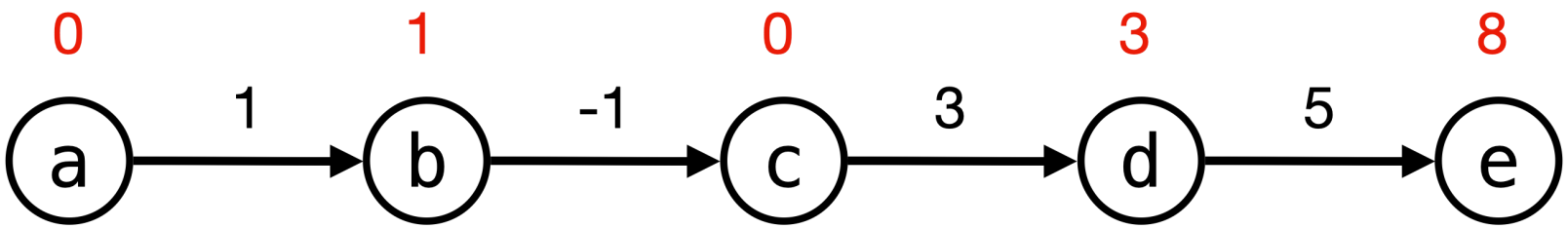
Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

After 3rd iteration



Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

After 4th iteration



Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

Correctness Proof Sketch

Regardless of **the order of edge relaxations**, the Bellman–Ford algorithm is guaranteed to find all shortest paths after $|V| - 1$ iterations.

This is because:

- Any simple path in a graph has at most $|V| - 1$ edges.
- Each iteration allows paths that are one edge longer to be considered.
- After $|V| - 1$ iterations, all possible simple paths have been explored.

Claim 2: If G has no negative cycle reachacle from s, then Bellman-Ford returns the correct shortest path distances, i.e., $d[v] = d(s, v), \forall v \in V$

Claim 2.1: Shortest paths have at most $n - 1$ edges

If there is a path from $s \rightarrow v$, then the shortest path from s to v must have $\leq n - 1$ edges.

Claim 2.1: Shortest paths have at most $n-1$ edges

If there is a path from $s \to v$, then the shortest path from s to v must have $\leq n-1$ edges.

Why?:

If a shortest path contains a cycle:

- The cycle cannot be negative (otherwise Bellman-Ford detects it - **Claim 1**).
- If the cycle has positive weight, removing it gives a shorter path.
- If the cycle has zero weight, removing it does not change the distance.

Thus, shortest paths are always simple paths (**no cycle!** 🚫). A simple path never revisits a vertex. With n vertices total, such a path can have $\leq n-1$ edges. ✅

Claim 2.1: Shortest paths have at most $n-1$ edges (proved ✅)

Ok. 🧐 Now let $d_k(v)$ = distance estimate after k iterations. We claim:

Claim 2.2: $d_k(v) \leq$ the minimum distance of a path from s to v with $\leq k$ edges.

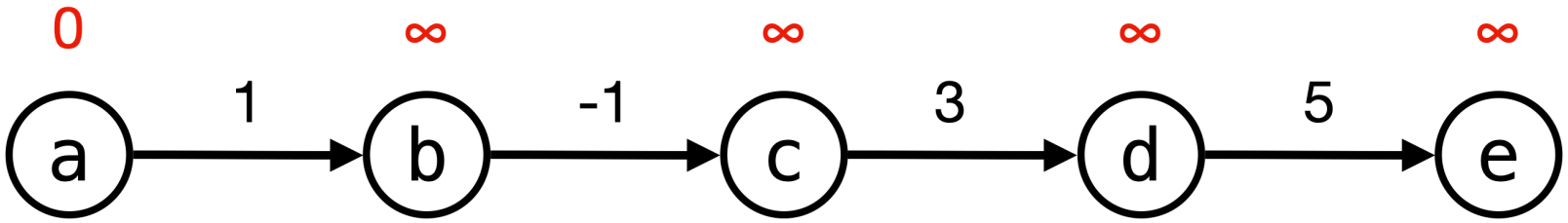
If this is true:

- After $n-1$ iterations: $d[v] = d_{n-1}(v)$
- By Claim 2.1, the shortest path from s to v should have $\leq n-1$ edges.
- By Claim 2.2, $d_{n-1}(v) \leq$ **the minimum distance of a path from s to v with $\leq n-1$ edges**, i.e., $d(s,v)$.
- Combinining above, $d[v] = d_{n-1}(v) \leq d(s,v)$.
- Since $d[v] \geq d(s,v)$ (same reasoning as Dijkstra), $d[v] = d(s,v)$ ✅

Let's prove **Claim 2.2**

> $d_k(v) \leq$ **the minimum distance of a path from** s **to** v **with** $\leq k$ **edges**

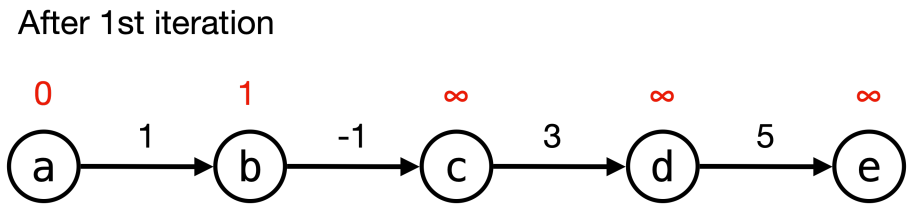
Base case ($k = 0$): $d_0(s) = 0, d_0(v) = \infty$ for $v \neq s$. Matches paths with ≤ 0 edges. ✅



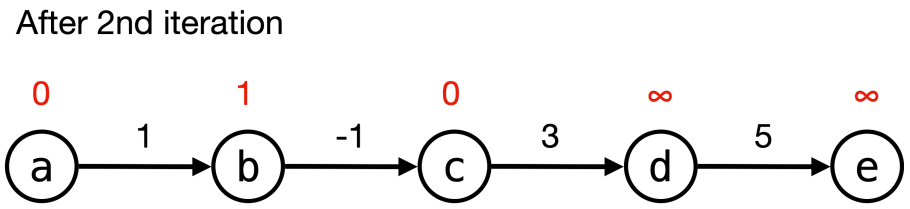
Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

Inductive hypothesis:

$d_{k-1}(v)$ is at most the minimum distance of a path from s to v with $\leq k-1$ edges.



Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)



Edge Relaxation Order: (d, e), (c, d), (b, c), (a, b)

Inductive step:

Consider $v \neq s$. Let P be a shortest path $P: s \to \cdots \to u \to v$ with $\leq k$ edges.

Let Q be the sub-path of P from s to u. Then, Q is a shortest path from s to u with $\leq k-1$ edges. By the inductive hypothesis, $d_{k-1}(u) \leq Q$'s distance weight.

Relaxing (u, v) in iteration k: $d_k(v) \leq d_{k-1}(u) + w(u, v) \leq \mathrm{w}(Q) + w(u, v) = w(P)$.

Thus, $d_k(v)$ is at most the cost of a shortest path with $\leq k$ edges. ✅

Claim 2.1: Shortest paths have at most $n-1$ edges ✅

Let $d_k(v)$ = distance estimate after k iterations.

Claim 2.2: $d_k(v) \leq$ the minimum distance of a path from s to v with $\leq k$ edges. ✅

- The final distance estimate $d[v]$ is $d_{n-1}(v)$.
- By Claim 2.1, the shortest path from s to v should have $\leq n-1$ edges.
- By Claim 2.2, $d_{n-1}(v) \leq$ **the minimum distance of a path from s to v with $\leq n-1$ edges = The shortest distance from s to v**
- Combinining above, $d[v] = d_{n-1}(v) \leq d(s,v)$.
- Since $d[v] \geq d(s,v)$ (same reasoning as Dijkstra), $d[v] = d(s,v)$ ✅

Runtime of Bellman-Ford Algorithm

- The total runtime of the Bellman-Ford algorithm is $O(mn)$.
- In the first for loop, we repeatedly update the distance estimates $n-1$ times on all m edges in time $O(mn)$.
- In the second for loop, we go through all m edges to check for negative cycles in time of $O(m)$.

Summary: Bellman-Ford Algorithm

The algorithm calculates shortest paths in a bottom-up manner.

- It first calculates the shortest distances which have at-most one edge in the path.
- Then, it calculates the shortest paths with at-most 2 edges, and so on.
- After the i-th iteration, the shortest paths with at-most i edges are calculated.
- There can be maximum $|V|-1$ edge in any simple path, that is why the loop runs $|V|-1$ time.

The Bellman-Ford algorithm is a Dynamic Programming algorithm!

Dynamic Programming (DP)

- A basic paradigm in algorithm design used to solve problems by relying on **intermediate solutions** to smaller subproblems.
- Core ingredients:
  - **Optimal Substructure**
  - **Overlapping Subproblems**

Used in many classic algorithms, e.g., shortest paths, sequence alignment, knapsack, etc.

Next Class

- We'll learn the concept of **Dynamic Programming (DP)** and understand why Bellman–Ford is actually a DP algorithm.

👩‍🏫 So far, we solved the Single-Source Shortest Path (SSSP) problem.

🧐 But what if we want the **shortest paths between every pair of nodes**?

- This is the **All-Pairs Shortest Paths (APSP)** problem!
- We'll learn the **Floyd–Warshall** algorithm, a dynamic programming approach to solving APSP.

Credits & Resources

Lecture materials adapted from:

- Stanford CS161 slides and lecture notes
  - https://stanford-cs161.github.io/winter2025/
- *Algorithms Illuminated* by Tim Roughgarden
  - https://algorithmsilluminated.com/