EDGE Algorithm Write-Up: ⟨Bellman–Ford⟩

Sanjana S. Mali, Toluwanimi O. Odemuyiwa, Serban D. Porumbescu, Joel S. Emer, John D. Owens

1 Introduction

This document explains the Bellman–Ford shortest path algorithm using the EDGE (Extended General Einsums) abstraction [6].

1.1 Why This Write-Up?

This write-up belongs to the EDGE Zoo, a growing collection of worked examples that each express a well-known algorithm as an extended Einsum cascade. The Zoo targets a variety of algorithms, and each write-up follows the same structure so that the specifications can be read against one another.

A worked example, not an algorithm tutorial. We assume the reader already knows Bellman–Ford. The emphasis here is on the *translation*: Section 3 runs one small graph through relaxation steps and then through the same graph expressed as tensors, and Section 5 states the full cascade, with each line explained in both graph-algorithm terms and Einsum terms. Reading the two views against each other is meant to make the notation concrete in a way that just the specification cannot.

Separating the algorithm from its implementation. Section 8 develops the observation that the dynamic-programming and vanilla formulations of Bellman–Ford induce the *same* EDGE expression, and differ only in the format and implementation spaces of the TeAAL pyramid of abstractions [5]. Distinguishing what is algorithmic from what is an implementation choice is a large part of what the Zoo is for.

2 Problem Statement

2.1 Problem Description

The Bellman–Ford algorithm solves the single-source shortest path problem on a weighted graph. Given a graph and a designated source vertex, the goal is to compute the shortest distance from the source to every other vertex in the graph [3, 8].

Unlike algorithms such as Breadth-First Search or Dijkstra's algorithm, Bellman–Ford allows edge weights to be negative. However, the algorithm assumes that the graph does not contain any negative-weight cycles that are reachable from the source vertex. If such a cycle exists, shortest path distances are not well-defined, as path costs can decrease indefinitely.

The algorithm works by iteratively improving distance estimates. Each iteration considers paths that are one edge longer than those considered in the previous iteration. After a sufficient number of iterations, the distance estimates converge to the shortest path distances.

2.2 Inputs

The algorithm takes the following inputs:

- A weighted directed graph $G = (V, E)$, where:
  - V is the set of vertices
  - E is the set of edges
  - Each edge $(u, v) \in E$ has an associated weight w
- A source vertex $s \in V$

The following assumptions are made:

- Edge weights may be negative
- No negative-weight cycle is reachable from the source vertex s
- If an edge (u, v) does not exist, it is treated as having infinite weight

2.3 Outputs

The output of the algorithm is a distance array D, where:

- $D[v]$ represents the length of the shortest path from the source vertex s to vertex v
- If a vertex v is not reachable from s, then $D[v] = +\infty$

2.4 Algorithmic Steps

The Bellman–Ford algorithm computes shortest-path distances using repeated edge relaxation. The algorithm proceeds as follows [7].

1. Initialize a distance array D:
   - Set $D[s] = 0$, because the shortest path from the source vertex to itself has length 0
   - Set $D[v] = +\infty$ for all vertices $v \neq s$, because initially these vertices are not yet reachable from s
2. Repeat the following process exactly $|V| - 1$ times:
   - For every edge $(u, v) \in E$:
     - Compute the candidate distance $D[u] + G[u][v]$
     - Update $D[v]$ to the minimum of its current value and this candidate distance
3. After completing the iterations, return the distance array D

In the EDGE formulation described later in this document, each iteration allows shortest-path information to propagate one edge farther from the source. Since any shortest path in a graph without negative cycles contains at most $|V| - 1$ edges, these iterations are sufficient to compute the correct shortest distances.

For reference, Algorithm 1 gives a standard sequential pseudocode presentation of Bellman–Ford.

Algorithm 1 Bellman–Ford Algorithm

```
1:  for each u ∈ V do
2:      d(u) ← +∞
3:  end for
4:  d(s) ← 0
5:  for k = 1 to |V| − 1 do
6:      for each v ∈ V do
7:          for each u ∈ V do
8:              if G[u][v] ≠ +∞ then    ▷ edge (u, v) exists
9:                  d(v) ← min(d(v), d(u) + G[u][v])
10:             end if
11:         end for
12:     end for
13: end for
```

3 Algorithm Execution Flow: Graph View

3.1 Example Graph

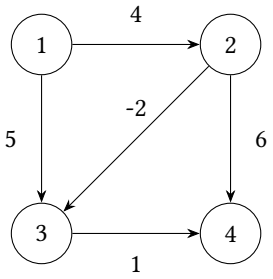

Figure 1. Example weighted directed graph

Consider the weighted directed graph shown in Figure 1. The graph contains four vertices and five weighted edges, including a negative edge, but no negative-weight cycles.

- Vertices: $V = \{1, 2, 3, 4\}$
- Source vertex: $s = 1$

The edges and their weights are:

- $1 \rightarrow 2$ with weight 4
- $1 \rightarrow 3$ with weight 5
- $2 \rightarrow 3$ with weight -2
- $2 \rightarrow 4$ with weight 6
- $3 \rightarrow 4$ with weight 1

This graph has no negative-weight cycles, so shortest path distances are well-defined.

3.2 Initialization

The algorithm begins by initializing the distance array D.

- The distance to the source vertex is set to zero:

$$D[1] = 0$$

- All other distances are initialized to infinity:

$$D = [0,\ +\infty,\ +\infty,\ +\infty]$$

At this point, only the source vertex is known to be reachable.

3.3 Iteration 1

In the first iteration, the algorithm "relaxes" every edge in the graph. Relaxation means updating the shortest distance to a vertex if a shorter path is found through another vertex [3]. For an edge (u, v) with weight $G[u][v]$, if going through u gives a shorter path from the source to v (that is, if $D[v] > D[u] + G[u][v]$), then we update $D[v]$ to $D[u] + G[u][v]$.

Here, all edge relaxations within a single iteration are conceptually performed in parallel: each update uses only distance values from the previous iteration. In other words, all reads occur before any writes are applied. This ensures that distance updates within the same iteration do not influence one another.

- From vertex 1 to vertex 2: $D[2] = \min(+\infty, 0 + 4) = 4$
- From vertex 1 to vertex 3: $D[3] = \min(+\infty, 0 + 5) = 5$
- From vertex 2 to vertex 3: No update occurs, since during this iteration all relaxations use the distances from the previous iteration, where $D[2] = +\infty$.
- Other edges do not produce improvements in this iteration.

After the first iteration:

$$D = [0,\ 4,\ 5,\ +\infty]$$

3.4 Iteration 2

The algorithm again relaxes all edges, using the distances computed in Iteration 1. As before, every relaxation reads the distances from the previous iteration.

- From vertex 2 to vertex 3: $D[3] = \min(5, 4 + (-2)) = 2$
- From vertex 2 to vertex 4: $D[4] = \min(+\infty, 4 + 6) = 10$
- From vertex 3 to vertex 4: $D[4] = \min(+\infty, 5 + 1) = 6$, using $D[3] = 5$ from the previous iteration. Combining both candidates for vertex 4 gives $D[4] = \min(10, 6) = 6$.
- Other edges do not lead to further improvements.

After the second iteration:

$$D = [0,\ 4,\ 2,\ 6]$$

3.5 Iteration 3

A third iteration is performed, since the algorithm runs for $|V| - 1 = 3$ iterations in total.

- From vertex 3 to vertex 4: $D[4] = \min(6, 2 + 1) = 3$
- Other edges do not lead to further improvements.

After the third iteration:

$$D = [0,\ 4,\ 2,\ 3]$$

3.6 Final Result

After completing $|V| - 1$ iterations, the algorithm terminates. The distance array now contains the shortest path distances from the source vertex to every other vertex:

- Shortest distance to vertex 1: 0
- Shortest distance to vertex 2: 4
- Shortest distance to vertex 3: 2
- Shortest distance to vertex 4: 3

This example illustrates how Bellman–Ford progressively refines distance estimates by repeatedly relaxing edges, allowing shorter paths to be discovered over successive iterations. The next section will revisit this same example and show how the same computation is expressed using EDGE tensor operations.

4 Algorithm Execution Flow: EDGE Tensor View

4.1 Tensors for the Working Example

We reuse the example graph from Section 3 with $V = \{1, 2, 3, 4\}$ and source $s = 1$. In EDGE, we represent the graph as an adjacency (weighted) tensor $G_{u,v}$ and the evolving shortest-path estimates as a distance tensor $D_{i,v}$, where i is the iterative rank.

Graph tensor G**.** The tensor $G_{u,v}$ stores the weight of the directed edge $u \to v$. The first index u corresponds to the *row* (source vertex), and the second index v corresponds to the *column* (destination vertex). If no edge exists, the entry is set to $+\infty$. For this example:

$$G = \begin{array}{c|cccc} G_{u,v} & v=1 & v=2 & v=3 & v=4 \\ \hline u=1 & +\infty & 4 & 5 & +\infty \\ u=2 & +\infty & +\infty & -2 & 6 \\ u=3 & +\infty & +\infty & +\infty & 1 \\ u=4 & +\infty & +\infty & +\infty & +\infty \end{array}$$

Distance tensor D**.** The tensor $D_{i,v}$ stores the distance estimate for vertex v after iteration i. We initialize all entries to $+\infty$ and set the source distance to 0 at iteration 0:

$$D_{0,v} = [0,\ +\infty,\ +\infty,\ +\infty]$$

Candidate tensor N**.** At each iteration, we compute a candidate distance tensor $N_{i,v}$ using the current distances $D_{i,u}$ and the graph weights $G_{u,v}$. Conceptually, $N_{i,v}$ represents the minimum distance to v obtained by taking exactly one more edge from some one-hop predecessor u, extending a path represented in D_i.

4.2 Building the EDGE Expression

Bellman–Ford repeatedly applies the same two-stage pattern:

1. **Relaxation candidate:** compute $N_{i,v}$ by combining all possible one-hop predecessors u of v.
2. **Distance update:** merge the new candidate $N_{i,v}$ into the running distance estimate.

Step 1: compute candidates $N_{i,v}$**.** For each destination vertex v, we consider the edge weight of every possible one-hop predecessor u to v. To each edge weight, we add the corresponding current distance of the path from the source s to u. We then take the minimum distance across all u. In EDGE notation, this corresponds to the min-reduction over the index u:

$$N_{i,v} = G_{u,v} \cdot D_{i,u} :: \bigwedge + (\cap) \ \bigvee \min(\cup) \tag{4.1}$$

Intuitively: "try all incoming edges into v, and keep the minimum distance we can form using one more edge."

Step 2: update distances $D_{i+1,v}$**.** Bellman–Ford keeps the minimum distance found so far, so we take the minimum of the previous distance and the new candidate:

$$D_{i+1,v} = N_{i,v} \cdot D_{i,v} :: \bigwedge \min(\cup) \tag{4.2}$$

Equations (4.1) and (4.2) are applied repeatedly for $i = 0, 1, \ldots, |V| - 2$.

4.3 Iteration-by-Iteration Values

Below we compute $N_{i,v}$ and $D_{i+1,v}$ for the first few iterations of the example. Each iteration uses the same EDGE pattern above.

4.3.1 Iteration $i = 0$.

We start from:

$$D_{0,v} = [0,\ +\infty,\ +\infty,\ +\infty]$$

Compute $N_{0,v}$**.** We evaluate $N_{0,v} = \min(D_{0,u} + G_{u,v})$ for each v:

- $N_{0,1} = +\infty$ (no incoming edges to 1)
- $N_{0,2} = \min(0 + 4,\ +\infty,\ +\infty,\ +\infty) = 4$
- $N_{0,3} = \min(0 + 5,\ +\infty,\ +\infty,\ +\infty) = 5$
- $N_{0,4} = +\infty$

So:

$$N_{0,v} = [+\infty,\ 4,\ 5,\ +\infty]$$

Update $D_{1,v}$**.**

$$D_{1,v} = \min(D_{0,v}, N_{0,v}) = [0,\ 4,\ 5,\ +\infty]$$

4.3.2 Iteration $i = 1$.

Now:

$$D_{1,v} = [0,\ 4,\ 5,\ +\infty]$$

Compute $N_{1,v}$**.**

- $N_{1,1} = +\infty$
- $N_{1,2} = \min(0 + 4,\ +\infty,\ +\infty,\ +\infty) = 4$
- $N_{1,3} = \min(0 + 5,\ 4 + (-2),\ +\infty,\ +\infty) = 2$
- $N_{1,4} = \min(+\infty,\ 4 + 6,\ 5 + 1,\ +\infty) = 6$

So:

$$N_{1,v} = [+\infty,\ 4,\ 2,\ 6]$$

Update $D_{2,v}$**.**

$$D_{2,v} = \min(D_{1,v}, N_{1,v}) = [0,\ 4,\ 2,\ 6]$$

4.3.3 Iteration $i = 2$. Now:

$$D_{2,v} = [0,\ 4,\ 2,\ 6]$$

Compute $N_{2,v}$**.**

- $N_{2,1} = +\infty$
- $N_{2,2} = 4$
- $N_{2,3} = \min(0 + 5,\ 4 + (-2),\ +\infty,\ +\infty) = 2$
- $N_{2,4} = \min(+\infty,\ 4 + 6,\ 2 + 1,\ +\infty) = 3$

So:

$$N_{2,v} = [+\infty,\ 4,\ 2,\ 3]$$

Update $D_{3,v}$**.**

$$D_{3,v} = \min(D_{2,v}, N_{2,v}) = [0,\ 4,\ 2,\ 3]$$

4.4 Convergence

For this example, the distance vector stops changing after the third update:

$$D_{i+1,v} \equiv D_{i,v} \tag{4.3}$$

which indicates convergence to the final shortest-path distances. This matches the result obtained in the Graph View execution flow (§ 3).

The next sections will generalize this construction and present the complete EDGE formulation using tensor definitions, initialization rules, and the final extended Einsums.

5 EDGE Expression

The full EDGE cascade for Bellman–Ford is as follows:

▹ **Tensors**

$$G^{U\equiv|V|,\ V\equiv|V|} \longrightarrow \text{Integer},\ \text{empty} = +\infty$$
$$D^{I,\ V\equiv|V|} \longrightarrow \text{Integer},\ \text{empty} = +\infty$$
$$N^{I,\ V\equiv|V|} \longrightarrow \text{Integer},\ \text{empty} = +\infty$$

▹ **Initializations**

$$G_{u,v} \longrightarrow \langle\text{user-specified}\rangle$$
$$D_{0,v} = +\infty$$
$$D_{0,s} = 0$$

▹ **Extended Einsums**

$$N_{i,v} = G_{u,v} \cdot D_{i,u} :: \bigwedge +(\cap)\ \bigvee \min(\cup) \tag{5.1}$$

[Compute relaxation candidates for each destination vertex]

$$D_{i+1,v} = N_{i,v} \cdot D_{i,v} :: \bigwedge \min(\cup) \tag{5.2}$$

[Keep the minimum distance found so far]

$$\diamond :\ D_{i+1,v} \equiv D_{i,v} \tag{5.3}$$

[Terminate when the distance tensor stops changing]

Cascade 1. Bellman–Ford EDGE Cascade

6 EDGE Expression Walk-Through

We now walk through the EDGE expression for Bellman–Ford step by step. The goal is to connect each tensor operation to the "relax edges and update distances" view of the algorithm, while also stating what each line means as an Einsum-style computation.

Step 1: Compute candidate distances by relaxing all edges. At iteration i, we start with the current distance estimates $D_{i,u}$ for every vertex u. To compute a new candidate distance for a destination vertex v, we consider every possible predecessor u and form the value:

$$D_{i,u} + G_{u,v}$$

If there is no edge $u \longrightarrow v$, then $G_{u,v} = +\infty$, so that predecessor does not affect the minimum. We then take the minimum across all predecessors u. This gives the minimum distance to v that can be achieved by taking one edge from some u using the distances known at iteration i.

$$N_{i,v} = G_{u,v} \cdot D_{i,u} :: \bigwedge +(\cap)\ \bigvee \min(\cup) \tag{6.1}$$

Graph meaning: "Try relaxing every edge into v, and keep the minimum candidate distance."

Einsum meaning: For each destination vertex v, the output tensor element $N_{i,v}$ is computed by considering every possible predecessor u. For each such u, we form the candidate value $D_{i,u} + G_{u,v}$. We then take the minimum over all u.

On Map: The Map operation maps over the rank u. The intersection on u means that for each vertex u with a valid distance $D_{i,u}$, we look at its outgoing edges (u, v) in $G_{u,v}$. In other words, for each u, we retrieve the list of neighbors v and propagate the candidate distance $D_{i,u} + G_{u,v}$ to them.

This corresponds to the $(\min, +)$ semiring used in GraphBLAS[1].

[1]In the GraphBLAS semiring formalism, $\oplus \equiv \min$ and $\otimes \equiv +$ over the domain $\mathbb{R} \cup \{+\infty\}$. The additive identity and multiplicative annihilator are

Step 2: Update distances by keeping the minimum value so far. The candidate $N_{i,v}$ in Equation (6.1) is a newly computed option for reaching v, but Bellman–Ford must preserve the minimum distance found across all previous iterations as well. So the next distance estimate is the minimum of:

- the old value $D_{i,v}$, and
- the new candidate $N_{i,v}$.

$$D_{i+1,v} = N_{i,v} \cdot D_{i,v} :: \bigwedge \min(\cup) \tag{6.2}$$

Graph meaning: "If the new relaxed value improves the distance to v, update it; otherwise keep the old distance."

Einsum meaning: Take an elementwise min between two tensors over v.

Step 3: Repeat until distances stop changing. Steps 1 and 2 are repeated. The algorithm stops when an iteration produces no changes in the distance values.

$$\diamond : \ D_{i+1,v} \equiv D_{i,v} \tag{6.3}$$

Graph meaning: "No edge relaxation improves any distance, so all shortest distances have been found."

Einsum meaning: Check the condition with $\diamond$. Stop if the tensor D_{i+1} does not change from the previous iteration (is elementwise equal to D_i).

7 Other Notes

Folding the two Einsums into one. In the standard presentation, Bellman–Ford is written as two steps: first compute the relaxation candidates $N_{i,v} = \min_u(D_{i,u} + G_{u,v})$, and then update the distance by $D_{i+1,v} = \min(D_{i,v}, N_{i,v})$. However, since $D_{i,v}$ does not depend on the reduction index u, the two operations can be combined algebraically. Specifically, we may treat the previous distance $D_{i,v}$ as an additional candidate inside the same min reduction. This yields an equivalent single extended Einsum formulation.

$$\begin{aligned} D_{i+1,v} = \big((G_{u,v} \cdot^1 D_{i,u})_{i,u,v} \cdot^2 D_{i,v}\big) \\ :: \overset{1}{\bigwedge} +(\cap) \ \overset{2}{\bigwedge} \min(\cap) \ \overset{2}{\bigvee} \min(\cup) \end{aligned} \tag{7.1}$$

In this formulation, the first combine operator + forms the edge-based candidates $D_{i,u} + G_{u,v}$. The second combine operator min compares each such candidate with the carry-over value $D_{i,v}$ (broadcast across u), and the final reduction takes the minimum over all u. Since $D_{i,v}$ is independent of u, this expression is equivalent to $\min(D_{i,v}, \min_u(D_{i,u} + G_{u,v}))$.

Iteration bound vs. convergence check. Classical Bellman–Ford performs exactly $|V| - 1$ relaxation rounds (and optionally one additional round for negative-cycle detection). The EDGE version uses a fixed-point termination condition ($\diamond$), which can stop earlier if an iteration produces no improvements. Both are correct when no negative-weight cycle is reachable from s.

Negative-cycle detection (optional extension). A common extension is to perform one additional relaxation pass after $|V| - 1$ rounds. If any distance can still be decreased, then a negative-weight cycle is reachable from s. This check is omitted from the core EDGE expression in this section to keep the formulation minimal; Section 9 adds it as an optional extension.

Early stopping and active sets. Practical implementations often track whether any distance changed during an iteration and terminate early if not. The EDGE formulation already captures the early-exit optimization implicitly through the condition ($\diamond$), since the computation terminates as soon as no distances change, without requiring explicit tracking of updates.

8 Algorithmic Variants

8.1 Dynamic Programming (DP) Formulation of Bellman–Ford

One *implementation* of Bellman–Ford is its dynamic programming (DP) formulation [2, 7].

From the perspective of the TeAAL pyramid of abstractions [5], the EDGE specification captures the algorithm at the *algorithmic space*, while the DP formulation corresponds to a choice at the *implementation level*, determining how the distance tensor is stored and updated across iterations.

In a standard (vanilla) implementation, the distance tensor D is represented as a single vector, where each iteration overwrites the previous one. In contrast, the DP formulation represents D as a two-dimensional table, where one dimension explicitly corresponds to the iteration index. In the DP formulation, we define:

$D[i][v]$ = shortest distance from s to v using at most i edges.

How it differs from the vanilla formulation. In the standard Bellman–Ford algorithm, the iteration index is implicit: each relaxation round conceptually allows paths with one additional edge. The DP formulation makes this explicit by introducing an additional index i, corresponding to the number of edges used.

Algorithmically, the DP recurrence is:

$$D[i][v] = \min\Big(D[i-1][v],\ \min_{u \in V}\big(D[i-1][u] + G[u][v]\big)\Big)$$

with base case:

$$D[0][s] = 0, \quad D[0][v] = +\infty \ \text{ for } v \neq s.$$

Interpretation. At iteration i, the DP formulation computes the minimum distance to each vertex using paths of length at most i. Each iteration extends the allowable path

both $+\infty$. See the GraphBLAS C API Specification [4] and the algebraic treatment in [1].

length by one edge. Since any shortest path in a graph without negative-weight cycles has at most $|V| - 1$ edges, the final answer is obtained from $D[|V| - 1][v]$.

Relationship to the EDGE formulation. With EDGE, we can apply a separation of concerns approach to understand the algorithm's design choices [5]. Using the TeAAL pyramid of abstractions [5], EDGE specifies the *algorithmic space*.

The DP formulation does not change the underlying EDGE einsum expression for relaxation. Instead, it corresponds to a different choice in the *format* and *implementation* spaces. Specifically:

- In the vanilla implementation, the tensor D is stored as a single vector and updated in-place across iterations.
- In the DP formulation, D is represented as a two-dimensional table indexed by iteration, making the progression of states explicit.

Thus, the EDGE specification remains unchanged, while the implementation differs in how the tensor is materialized and updated.

Summary. The DP formulation of Bellman–Ford changes neither the underlying computation nor the complexity of the algorithm. Instead, it provides a clear semantic interpretation in terms of paths of bounded length, which is useful for correctness proofs and tensor-based formulations.

Algorithm 2 Bellman–Ford (Dynamic Programming Formulation)

▷ $D[i][v]$ = shortest distance from source s to vertex v using at most i edges
for each $v \in V$ **do**
  $D[0][v] \leftarrow +\infty$
end for
$D[0][s] \leftarrow 0$
for $i = 1$ to $|V| - 1$ **do**
  for each $v \in V$ **do**
    $D[i][v] \leftarrow D[i-1][v]$ ▷ carry over paths using $\leq i - 1$ edges
    for each $u \in V$ **do**
      if $G[u][v] \neq +\infty$ **then** ▷ edge (u, v) exists
        $D[i][v] \leftarrow \min\big(D[i][v],\ D[i-1][u] + G[u][v]\big)$
      end if
    end for
  end for
end for
return $D[|V| - 1][v]$ for all $v \in V$

9 Negative-Cycle Detection

The formulation in Section 5 assumes that no negative-weight cycle is reachable from the source vertex s. This section adds an optional check that detects when that assumption is violated. The tensors, initializations, and Einsums of Section 5 are unchanged.

The idea. Any shortest path in a graph without negative-weight cycles uses at most $|V| - 1$ edges, so $|V| - 1$ rounds of relaxation are enough to reach the final distances. If some distance still decreases on round $|V|$, then no finite shortest path exists for that vertex, and a negative-weight cycle must be reachable from s.

We capture this as a Boolean flag NC, which is true exactly when some vertex has less distance after round $|V|$ than it was after round $|V| - 1$.

▷ **Tensors**

$$NC \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$

▷ **Initializations**

$$NC = \texttt{false}$$

▷ **Extended Einsums**

$$NC = D_{i:i=|V|,\,v} \cdot D_{i:i=|V|-1,\,v} :: \bigwedge < (\cap)\ \bigvee \mathrm{OR}(\cup) \tag{9.1}$$

[Flag a negative cycle if any distance improves on round $|V|$]

Cascade 2. Negative-Cycle Detection

Reading the Einsum. The two operands are slices of the distance tensor at fixed values of the iterative rank: the distances after round $|V|$ and the distances after round $|V|-1$. The combine operator $<$ compares them vertex by vertex, producing a Boolean for each v, and the reduction $\bigvee \mathrm{OR}(\cup)$ collapses those Booleans over v. The result is a scalar: true if any vertex improved on the final round, false otherwise.

If no negative-weight cycle is reachable from s, the distances converge and relaxation stops at or before round $|V| - 1$; round $|V|$ is never produced, the Einsum never executes, and NC keeps its empty value of `false`. If a negative-weight cycle is reachable, the distances along that cycle keep decreasing, relaxation continues past round $|V| - 1$, and NC becomes true. In the classical formulation, which runs a fixed $|V|$ rounds rather than iterating to a fixed point, NC is simply read once the final round completes.

References

[1] Aydı n Buluç, Jeremy Kepner, John R. Gilbert, Timothy Mattson, and David Ediger. 2016. The Mathematical Foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC.2016.7761646

[2] Allen Downey. 2026. Dynamic Programming: Shortest Paths. https://downey.io/notes/omscs/cs6515/dynamic-programming-shortests-paths/. Accessed August 2026.

[3] GeeksforGeeks. 2026. Bellman–Ford Algorithm. https://www.geeksforgeeks.org/dsa/bellman-ford-algorithm-dp-23/. Accessed August 2026.

[4] GraphBLAS Forum. 2023. GraphBLAS C API Specification. https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf. Version 2.0.0. Accessed August 2026.

[5] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-56)*. ACM, 1255–1270. doi:10.1145/3613424.3623791

[6] Toluwanimi O. Odemuyiwa, Serban D. Porumbescu, Joel S. Emer, and John D. Owens. 2026. The EDGE Language: Extended General Einsums for Graph Algorithms. arXiv:2404.11591v2 [cs.DS] https://arxiv.org/abs/2404.11591v2

[7] University of Illinois at Urbana-Champaign. 2021. Shortest Paths: Bellman–Ford Algorithm. https://courses.grainger.illinois.edu/cs374/sp2021/scribbles/B-2021-04-01_2.pdf. CS374 lecture notes. Accessed August 2026.

[8] Wikipedia contributors. 2026. Bellman–Ford algorithm. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm. Accessed August 2026.