Open in app ↗

Sign up

Sign In

Search Medium

Bellman-Ford Algorithm

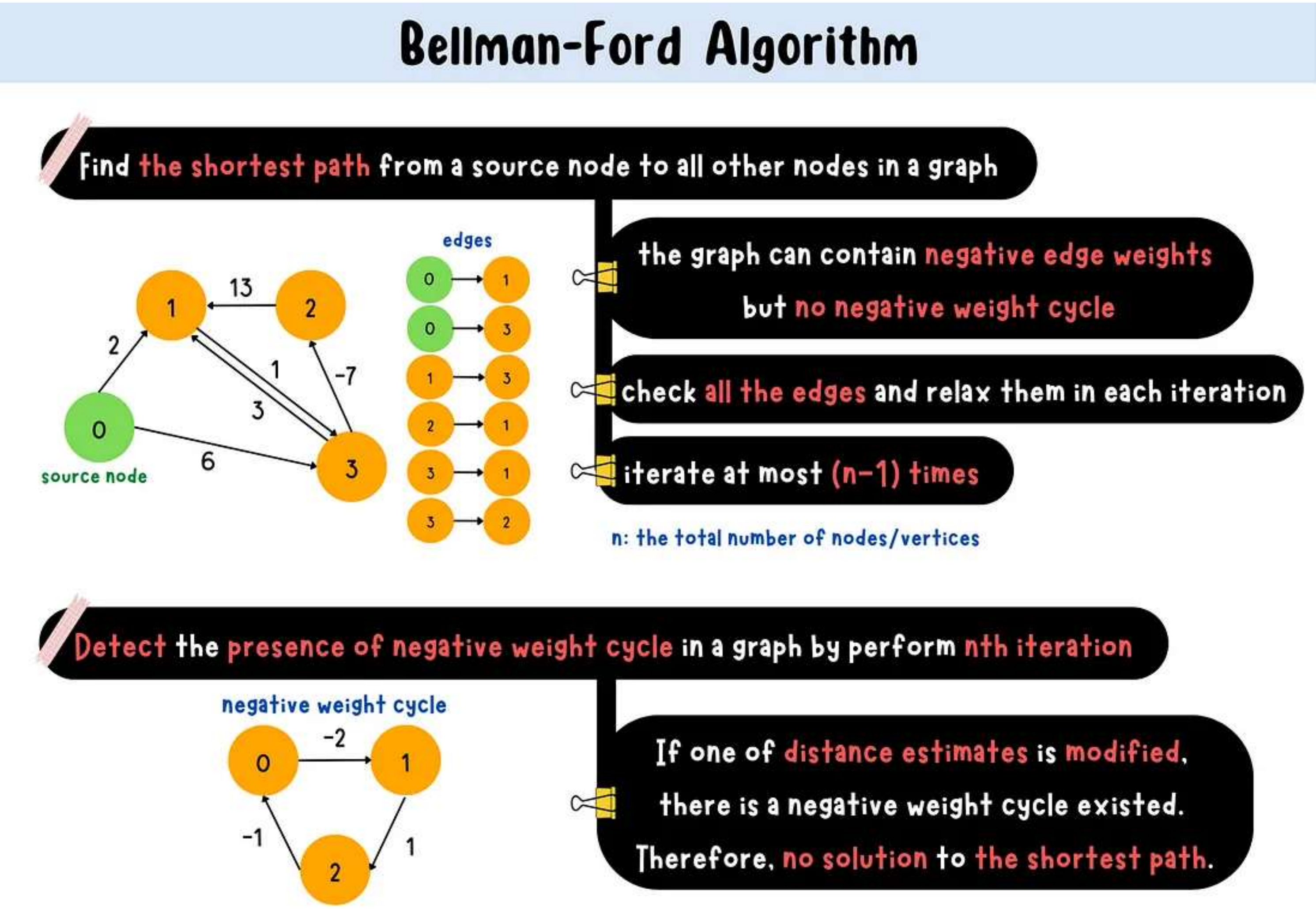
Claire Lee · Follow

7 min read · Sep 7, 2022

Listen

Share

> *Bellman-Ford Algorithm* computes **the single-source shortest path** from a source node to all other nodes in a graph that can contain **negative edge weights.** However, if a graph contains a **negative weight cycle**, the solution to the **shortest path** will **not be produced.** *This algorithm is* also used to **detect the presence of negative weight cycle** in a graph.



Bellman-Ford Algorithm summary card

· · ·

Table of Contents

. . .

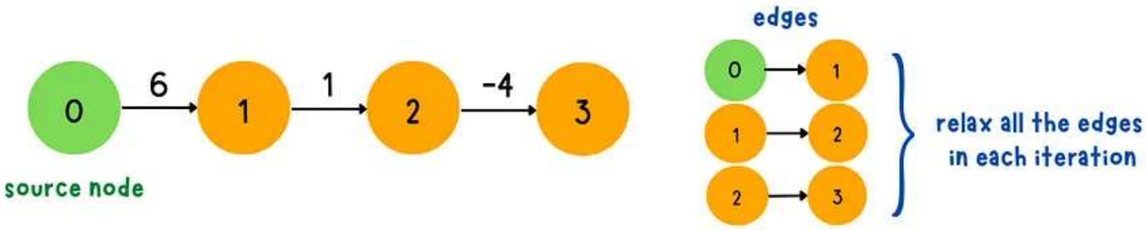
How Bellman-Ford Algorithm works?

This algorithm **overestimates** the **distance** from the source node to all other nodes in a graph initially. Then repeatedly **visited all the edges** and **relax** the estimate when a new shorter path is discovered in each iteration. If an iteration is **not** resulted in an **update**, we can **stop the algorithm** because the shortest path has been found. Otherwise, iterate all the edges at most **`n-1` times**(n is the total number of nodes/vertices) since `n-1` is the maximum length of the shortest path could take. If a graph has no negative cycle, we are guarantee to find the optimized result after `n-1` iterations.

Why Iterate (n-1) times in Bellman-Ford Algorithm?

In Bellman-Ford Algorithm, we can iterate all the edges **in any order** and relax them. For the same graph, when we consider all the edges in different orders, we may get the optimized results after less than `n-1` iterations or , in the worst case, need at most `n-1` iterations to produces the shortest path .

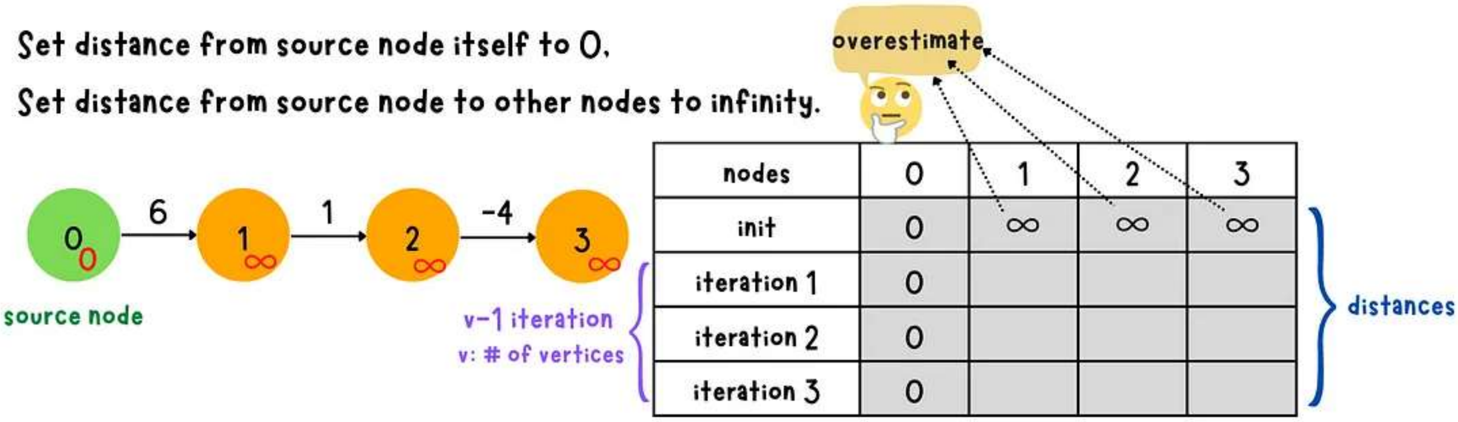
Take the following graph as an example. There are four nodes in the graph and we will visit all the edges in two different orders.



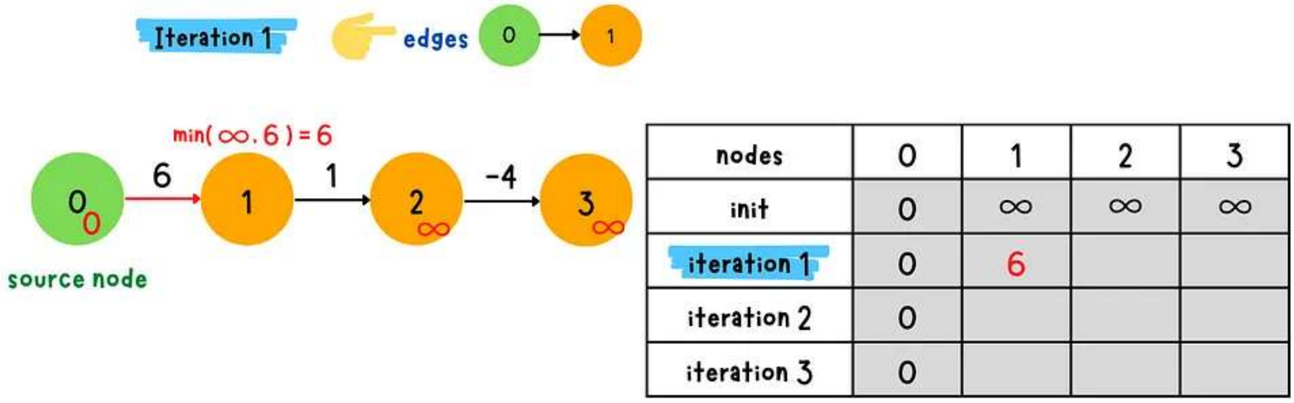
example graph

1. order: 0 → 1, 1 → 2, 2 → 3

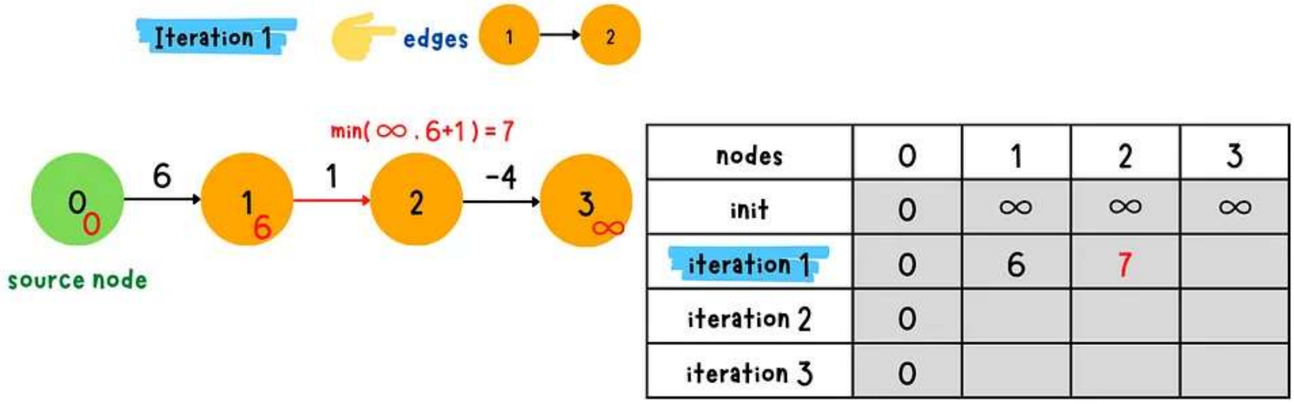
If we visited edges in (0 → 1, 1 → 2, 2 → 3) order, we can get the shortest path after two iterations. There is no update occurred at the end of second iteration, so we stop the algorithm.



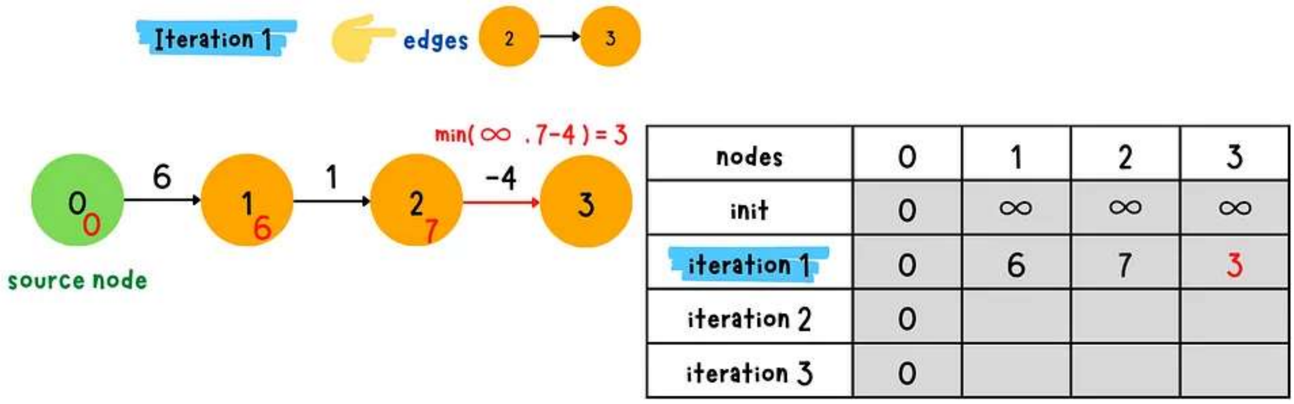
init settings



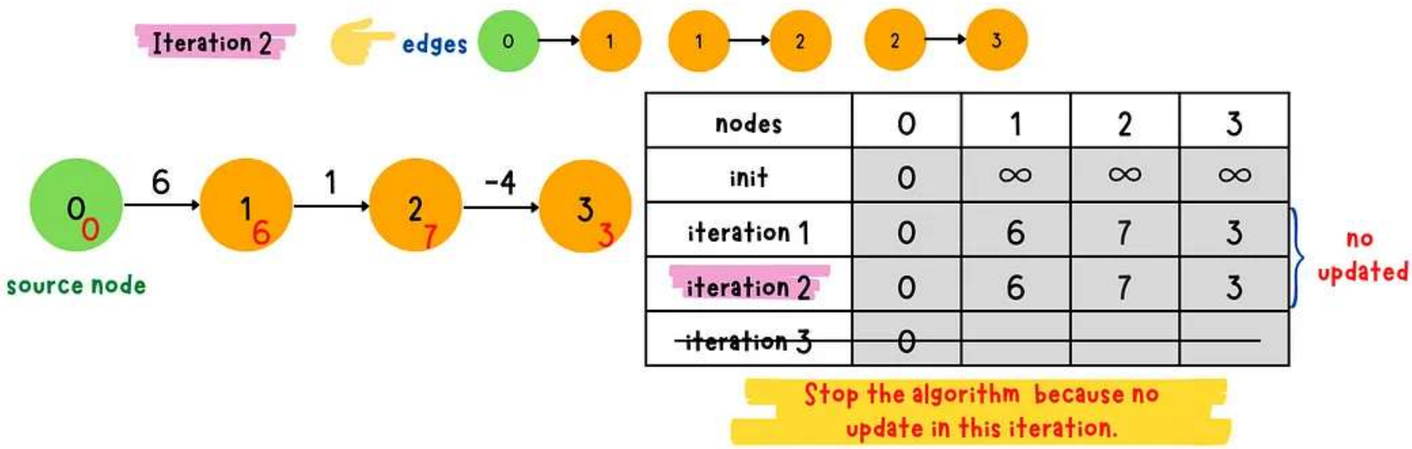
iteration 1–1



iteration 1–2



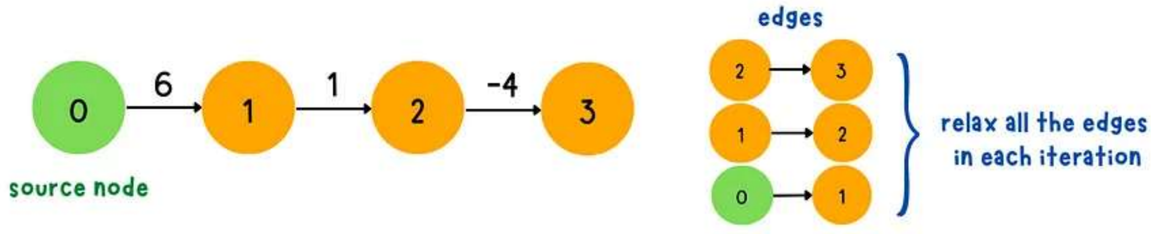
iteration 1–3



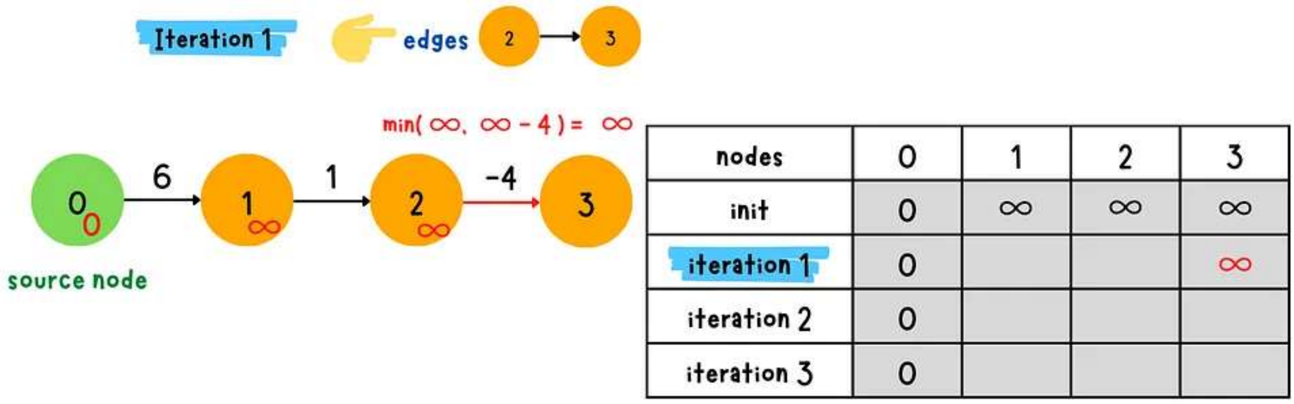
iteration 2

2. order: 2 → 3, 1 → 2, 0 → 1

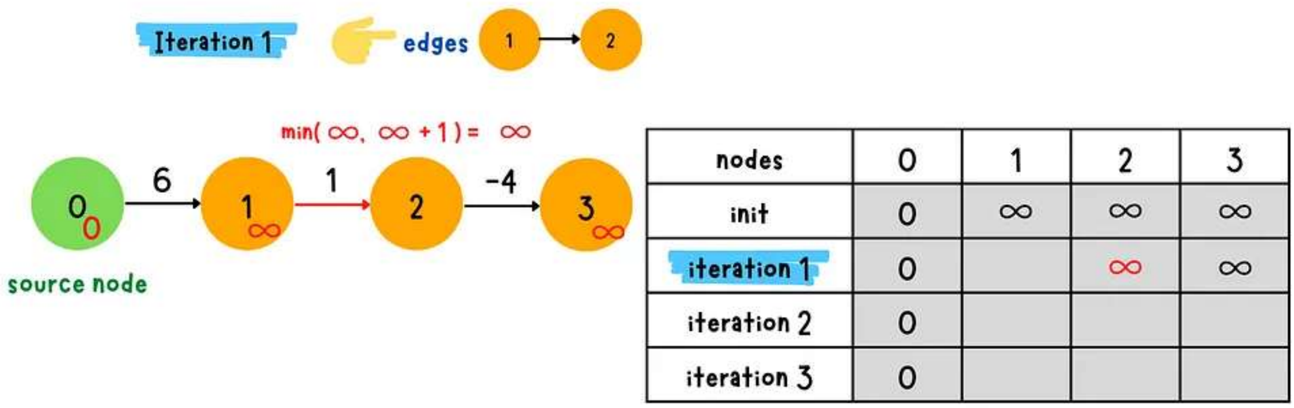
If we visited edges in (2 → 3, 1 → 2, 0→ 1) order, we will need `n-1` iterations to get the shortest path correctly.



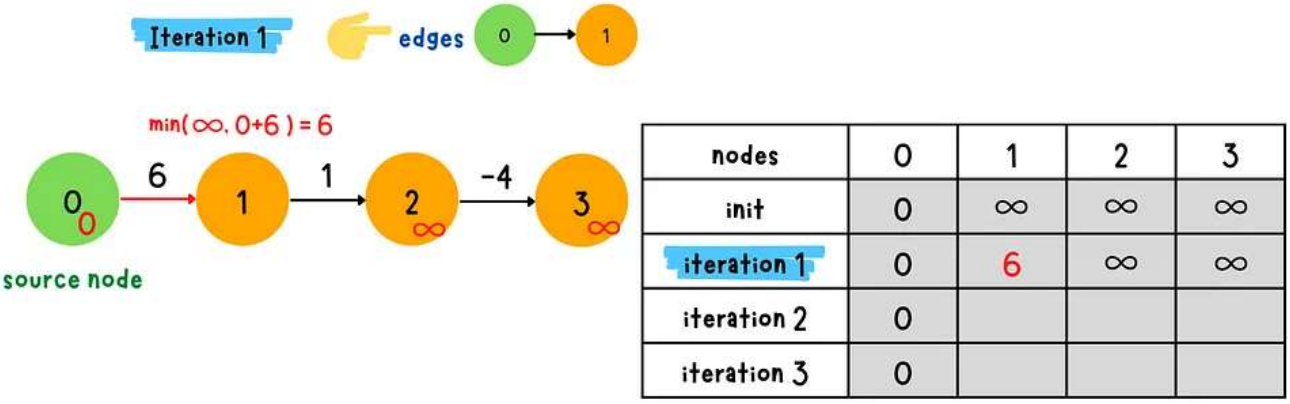
example graph



iteration 1–1

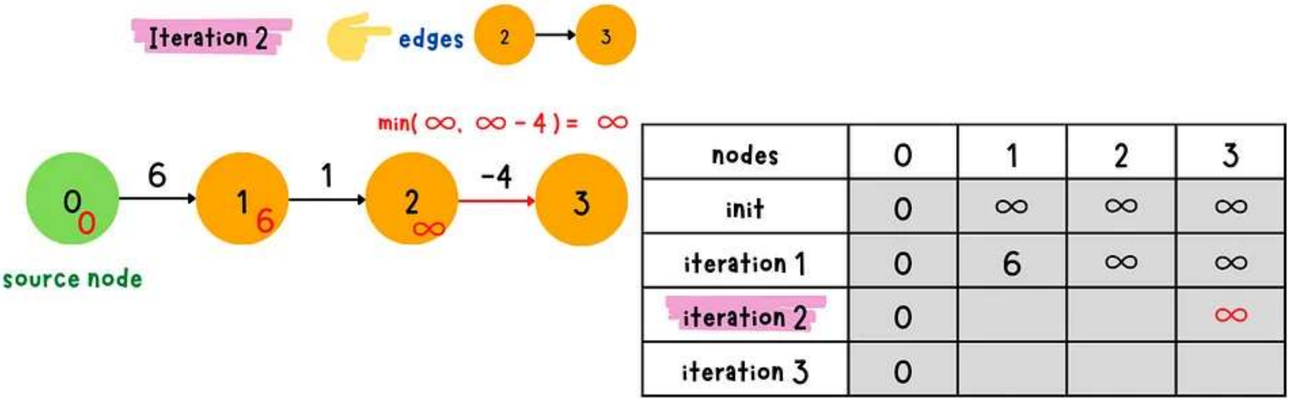


iteration 1–2



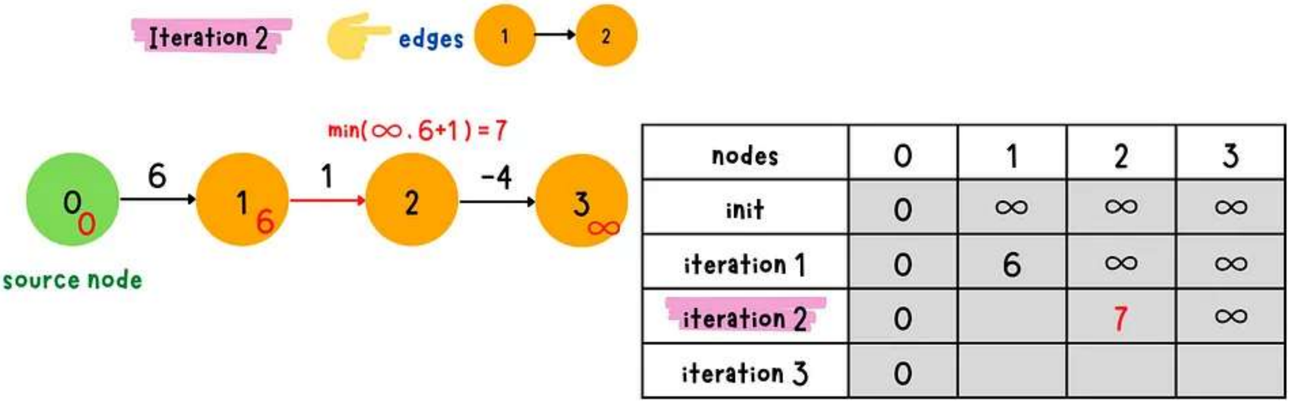
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	6	∞	∞
iteration 2	0			
iteration 3	0			

iteration 1–3



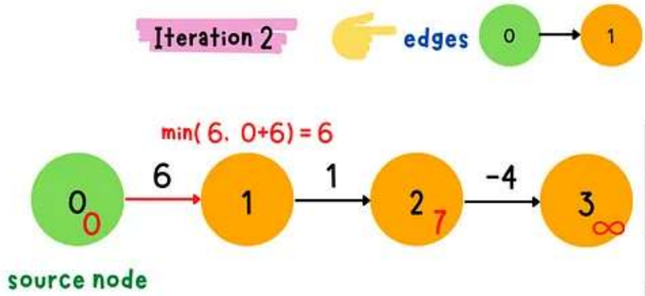
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	6	∞	∞
iteration 2	0			∞
iteration 3	0			

iteration 2–1



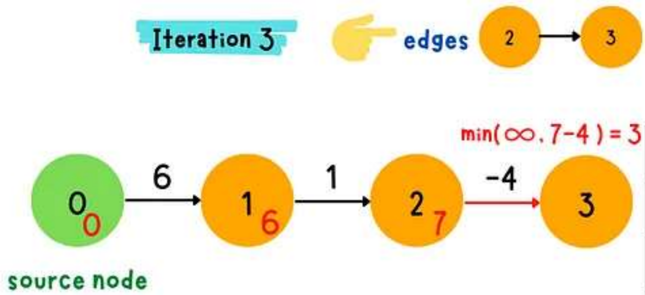
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	6	∞	∞
iteration 2	0		7	∞
iteration 3	0			

iteration 2–2



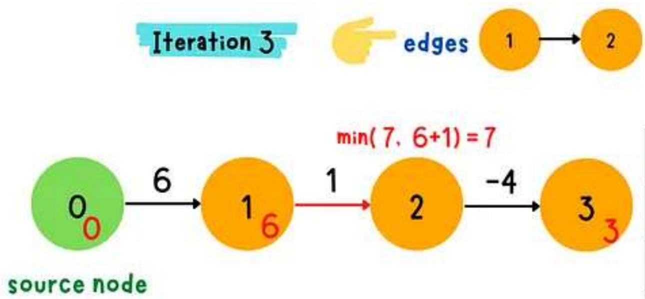
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	6	∞	∞
iteration 2	0	6	7	∞
iteration 3	0			

iteration 2–3



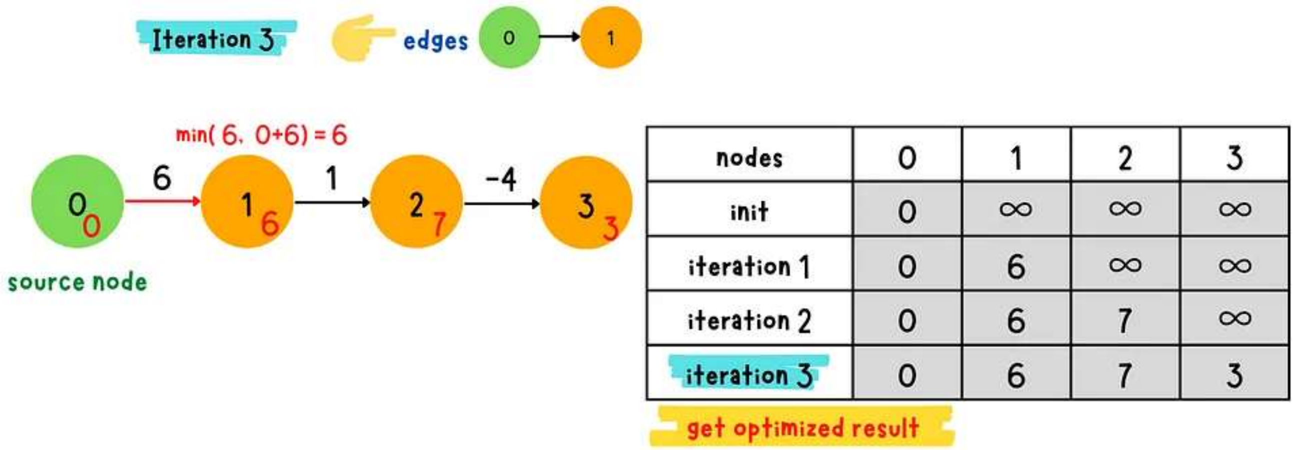
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	6	∞	∞
iteration 2	0	6	7	∞
iteration 3	0			3

iteration 3–1



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	6	∞	∞
iteration 2	0	6	7	∞
iteration 3	0		7	3

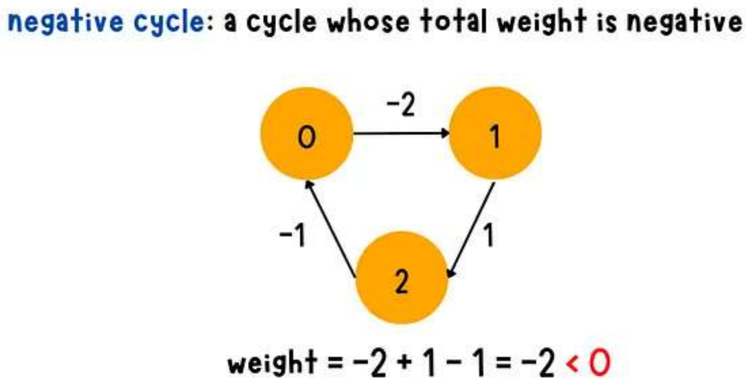
iteration 3–2



iteration 3–3

Detect negative cycles

When a graph contain **negative cycles**, it is not possible to find **the shortest path** from the source node to all other nodes. When a negative weight cycle is existed in a graph, every iteration of the cycle will give a shorter path.



negative cycle

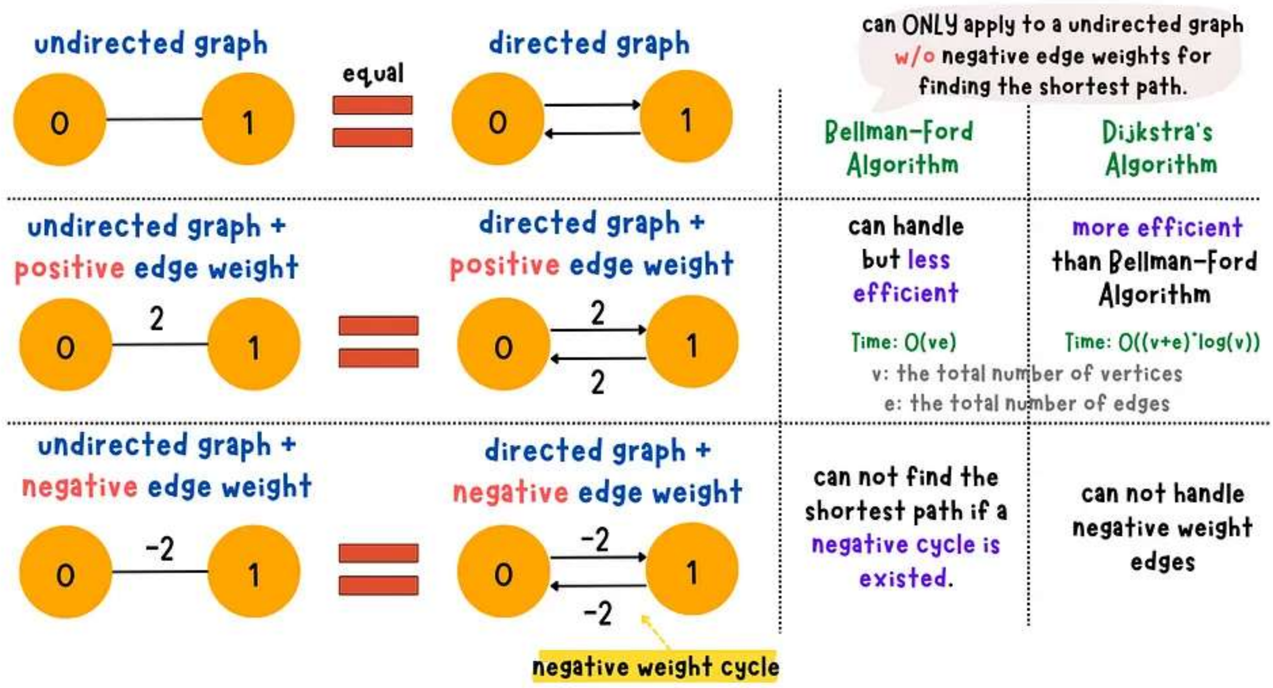
Bellman Ford algorithm provides a way to detect negative weight cycles in a graph. After we completed `n-1` iterations, we **perform the `nth` iteration**. If any **distance estimate** is **updated**, a **negative weight cycle** must **exist**.

Does Bellman-Ford Algorithm work for Undirected graph?

Bellman-Ford Algorithm can apply to **an undirected graph without negative edge weights** because a negative edge weight forms a negative weight cycle in the undirected graph. Therefore, Bellman-Ford will not able to find the shortest path in that graph due to the presence of negative weight cycles.

For an undirected graph with positive edge weights, we can use **Dijkstra's algorithm** to find the shortest path because it is more efficient(less time complexity) than Bellman-Ford algorithm.

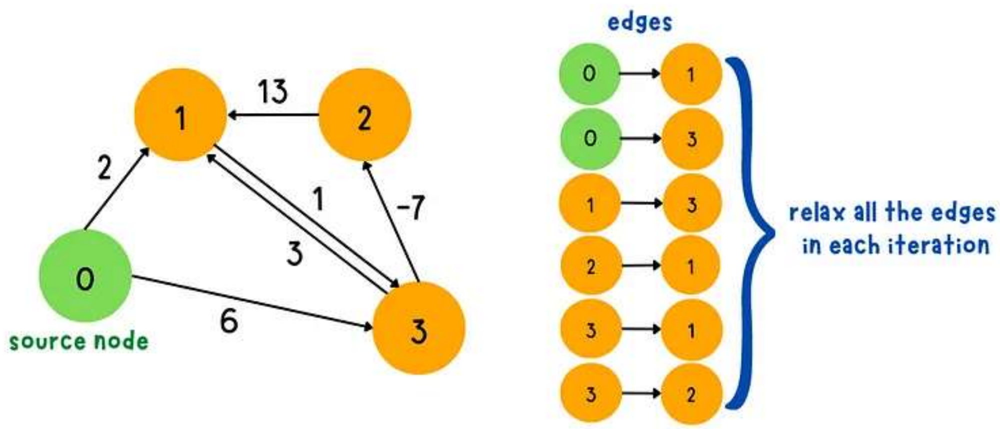
Therefore, Bellman-Ford Algorithm works for a **positive undirected graph** but it is less efficient than Dijkstra's algorithm if we want to use it to find the shortest path.



Graphical Explanation

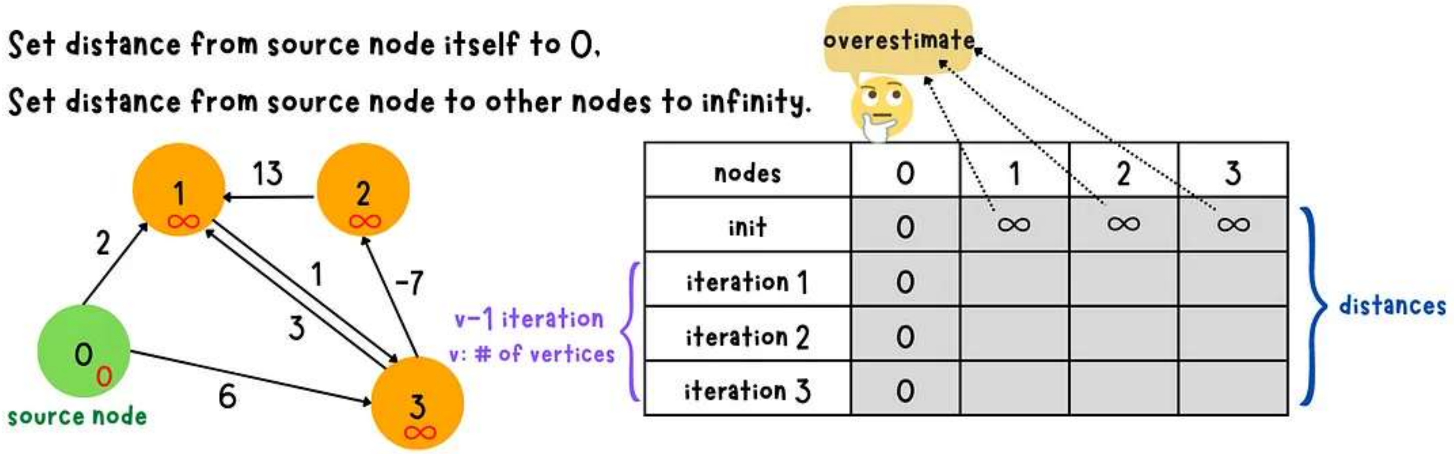
> *Because we can visited all the edges* ***in any order****, so we just choose one of the order to do the following graphical presentation.*

1. graph w/o negative weight cycle



demo graph without negative weight cycle

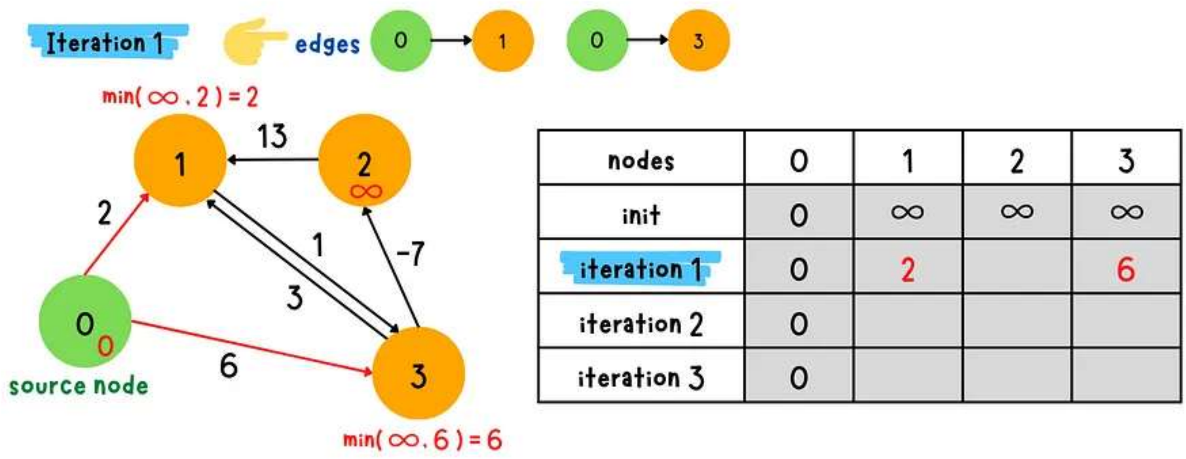
- *Step 1*: Set the distance to the **source node** itself to **0** and the distance to all **other nodes** to **infinity**.



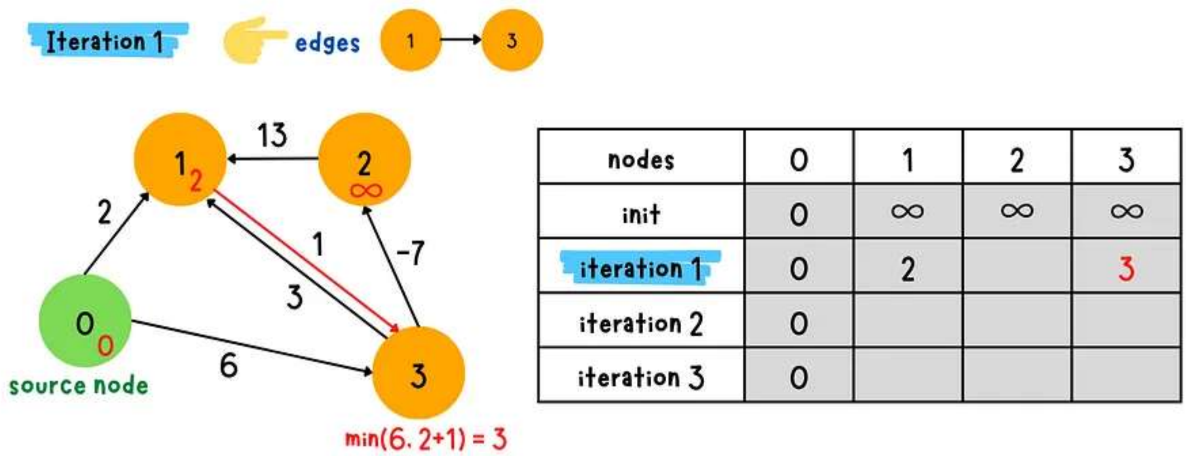
init setting

- *Step 2*: Check **all the edges** and relax them in each iteration.

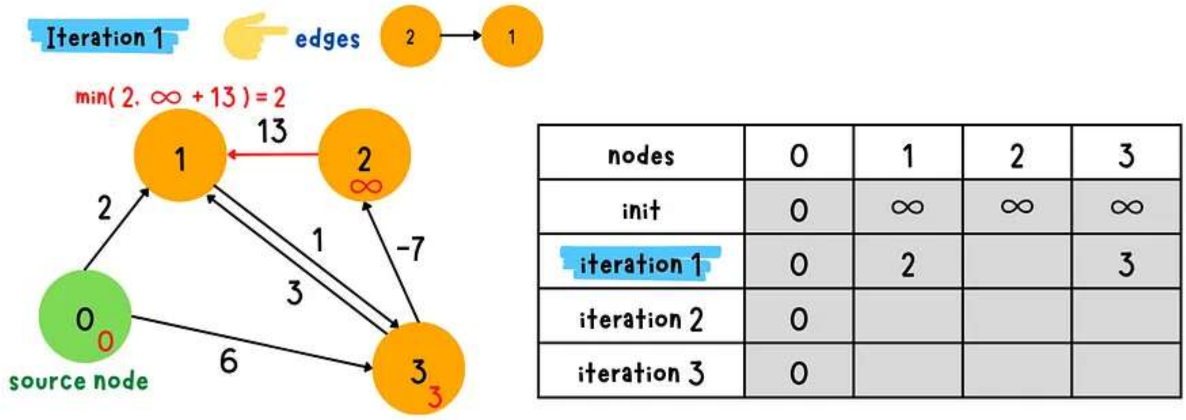
```
# ex. nodes a, b; edge a->b
min(
distance[b],
distance[a] + edge_weight(a, b),
)
```



iteration 1–1

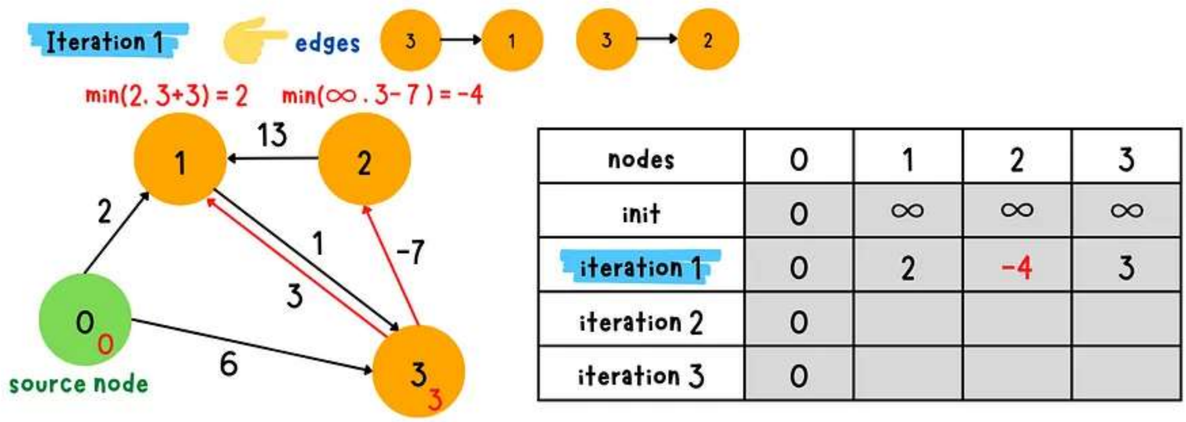


iteration 1–2



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2		3
iteration 2	0			
iteration 3	0			

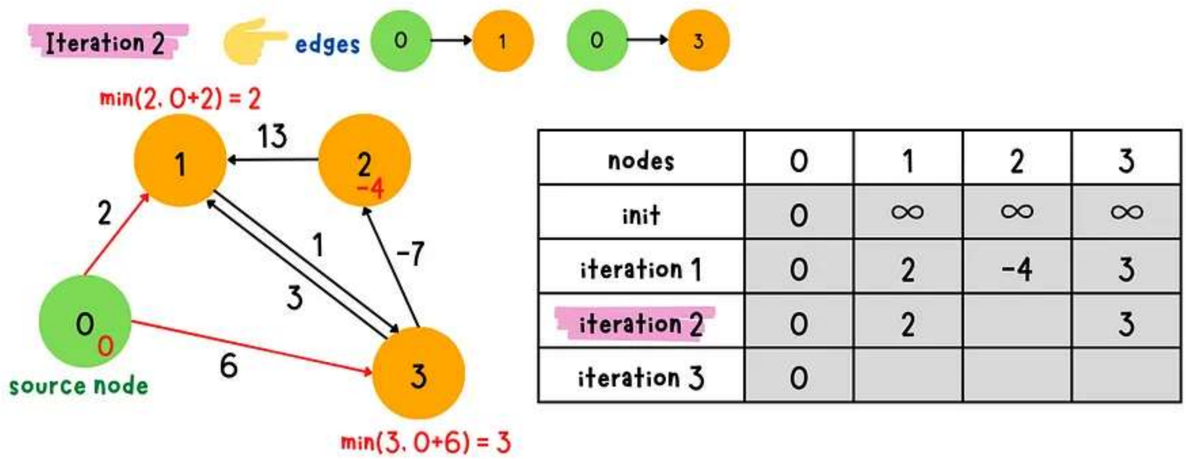
iteration 1–3



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0			
iteration 3	0			

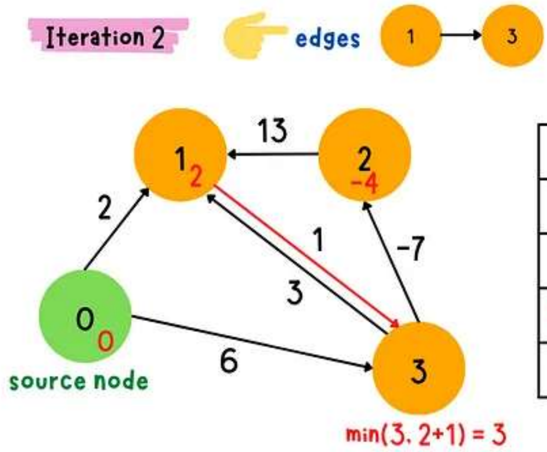
iteration 1–4

- *Step 3*: If an iteration does not result in any distance update, we stop the algorithm. Otherwise, keep iterating until `n-1` loops are completed.



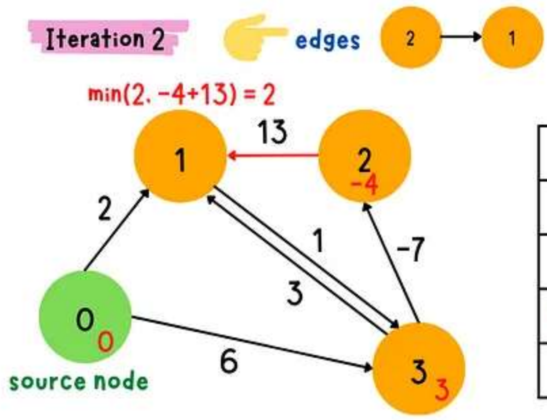
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	2		3
iteration 3	0			

iteration 2–1



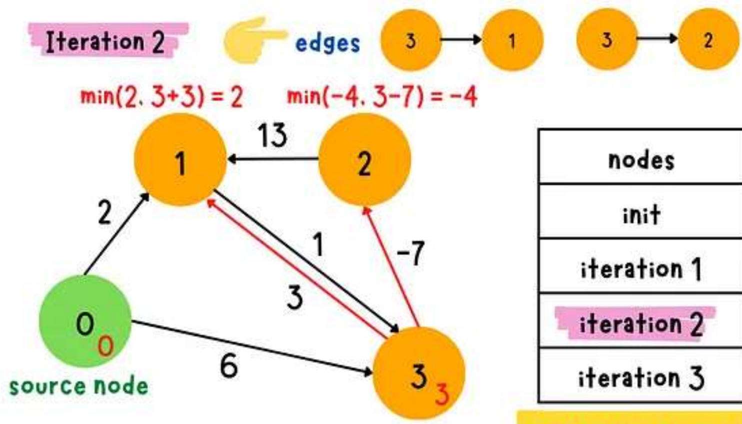
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	2		3
iteration 3	0			

iteration 2–2



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	2		3
iteration 3	0			

iteration 2–3



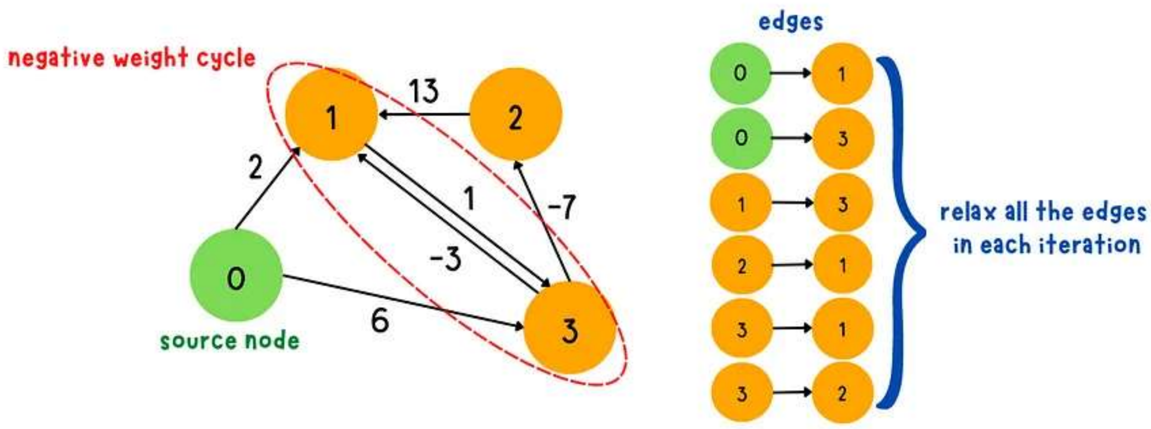
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	2	-4	3
iteration 3	0			

no updated

Stop the algorithm if an iteration does not result in an update.

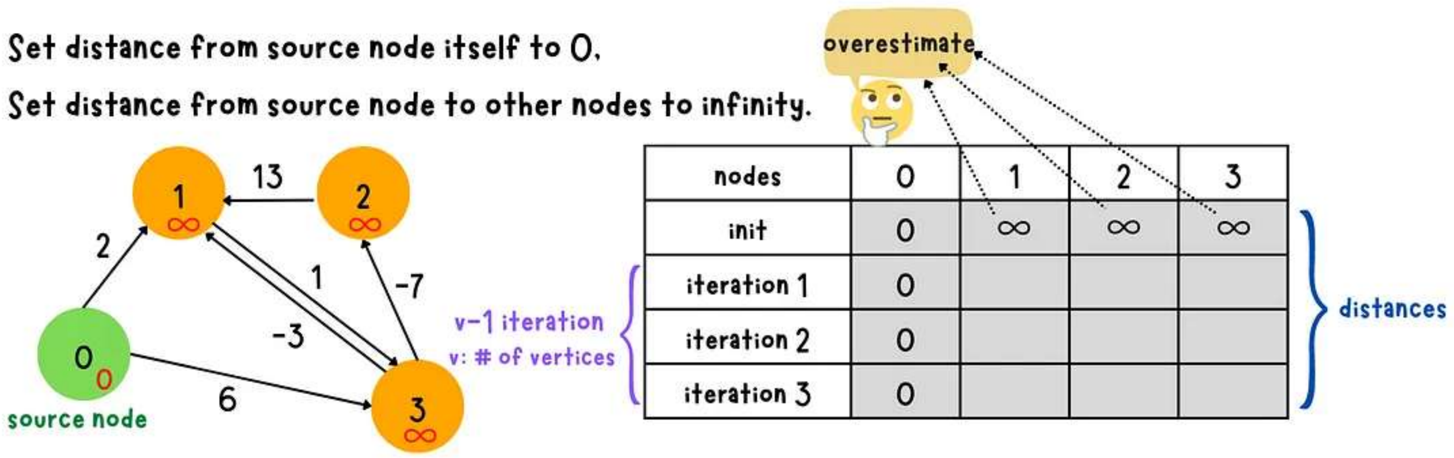
iteration 2–4

2. graph with negative weight cycle



demo graph with negative weight cycle

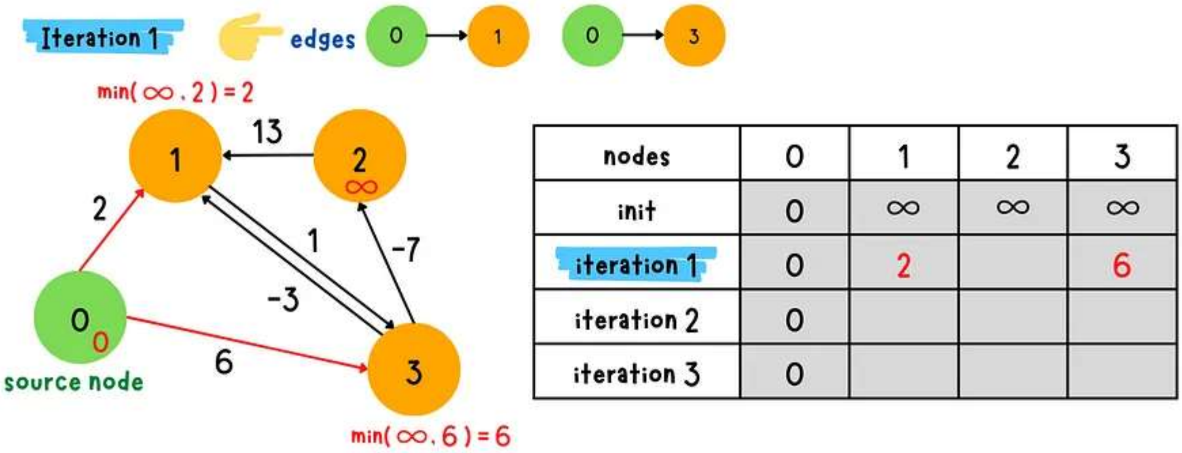
- *Step 1*: Set the distance to the **source node** itself to **0** and the distance to all **other nodes** to **infinity**.



- *Step 2*: Check **all the edges** and relax them in each iteration.

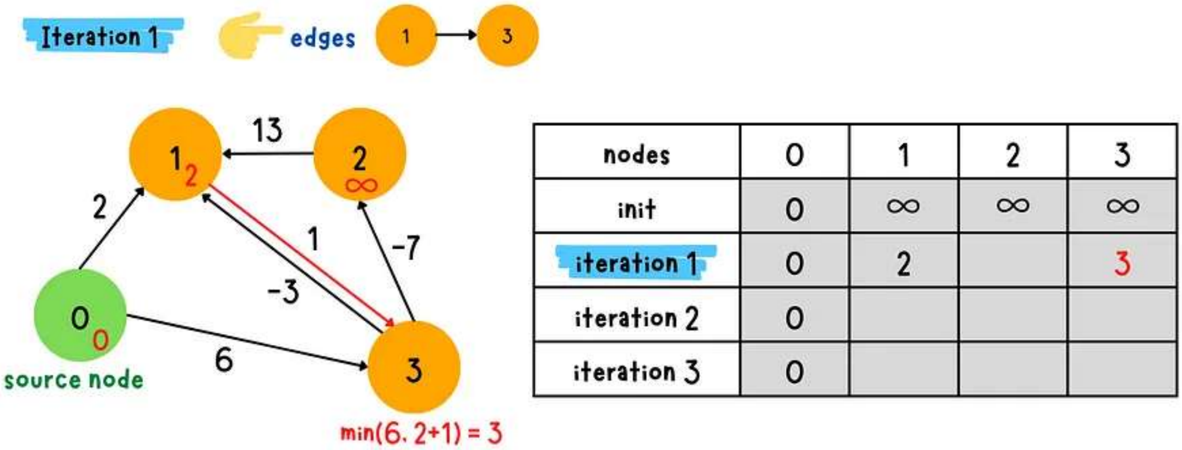
```
// ex. nodes a, b; edge a->b
min(
distance[a],
distance[b] + edge_weight(a, b),
)
```

- *Step 3*: Keep iterating until `n-1` loops are completed.



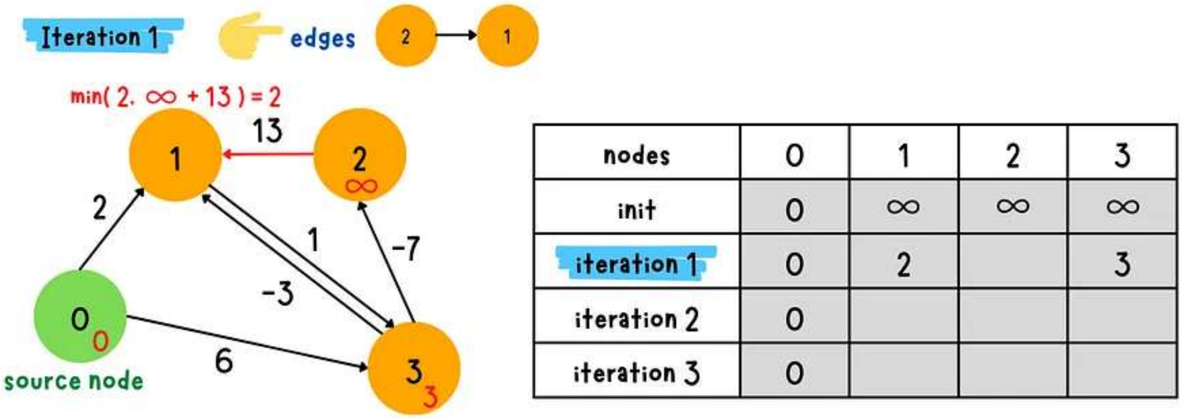
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2		6
iteration 2	0			
iteration 3	0			

iteration 1–1



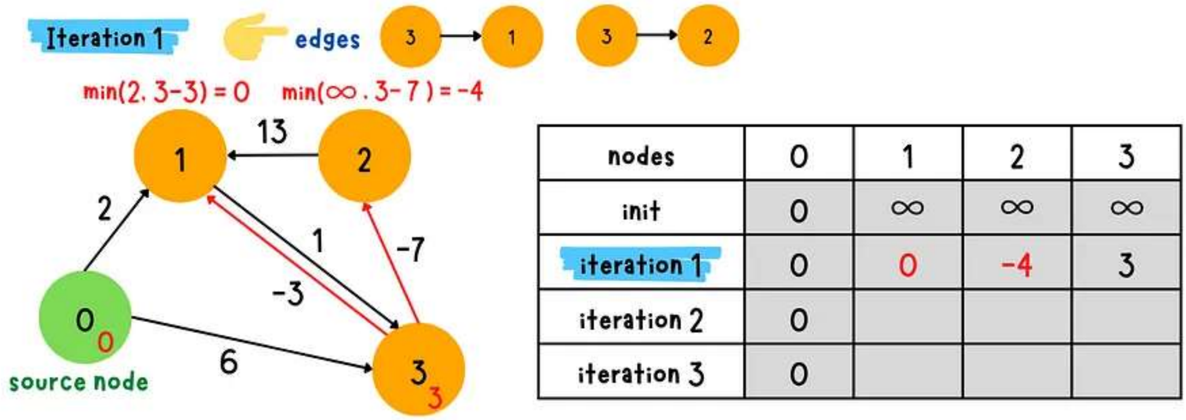
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2		3
iteration 2	0			
iteration 3	0			

iteration 1–2



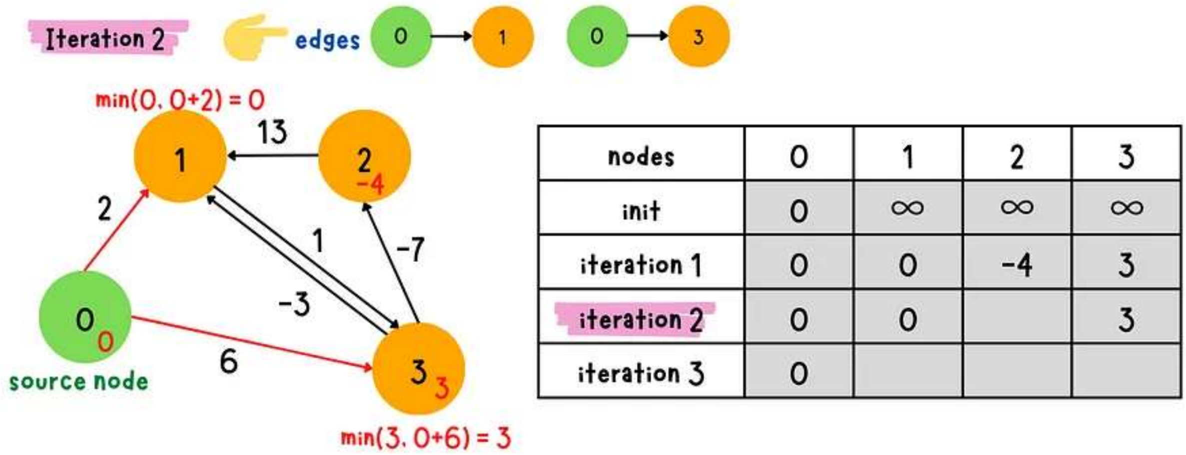
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2		3
iteration 2	0			
iteration 3	0			

iteration 1–3



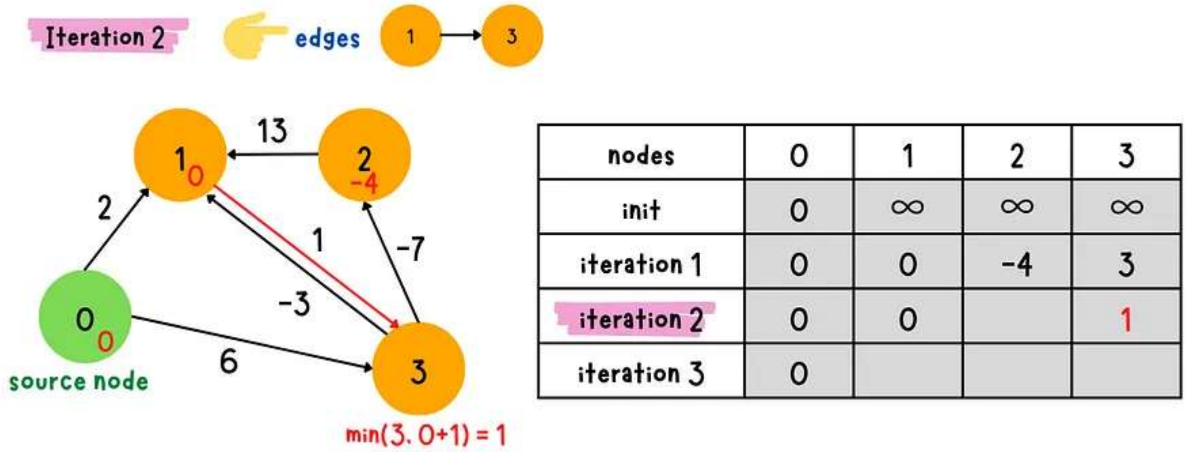
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	0	−4	3
iteration 2	0			
iteration 3	0			

iteration 1–4



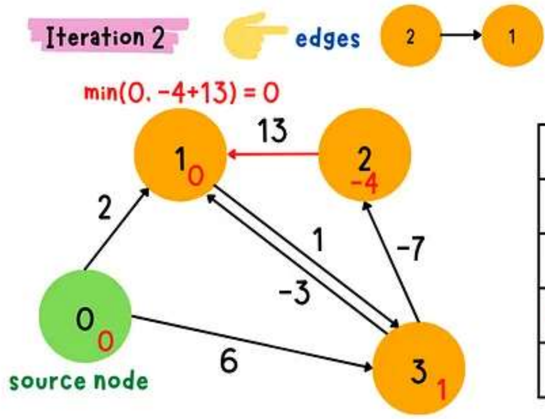
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	0	−4	3
iteration 2	0	0		3
iteration 3	0			

iteration 2–1



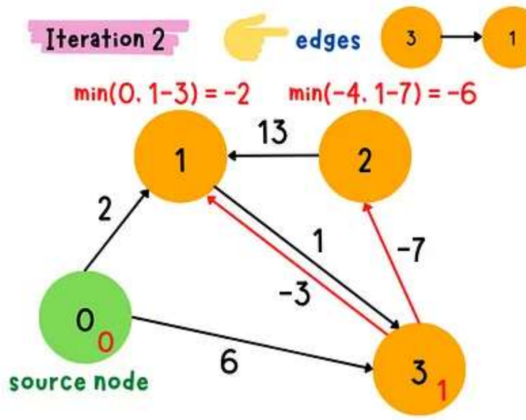
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	0	−4	3
iteration 2	0	0		1
iteration 3	0			

iteration 2–2



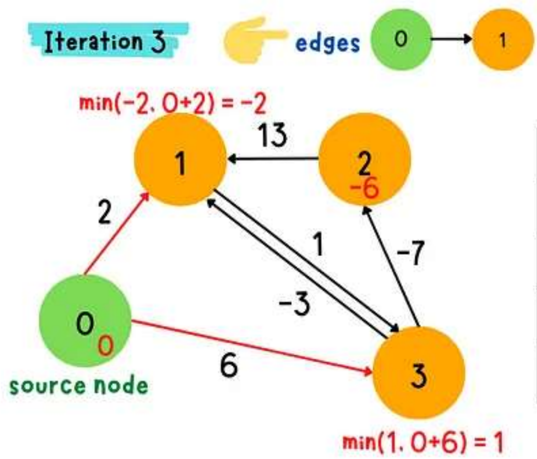
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	0	-4	3
iteration 2	0	0		1
iteration 3	0			

iteration 2–3



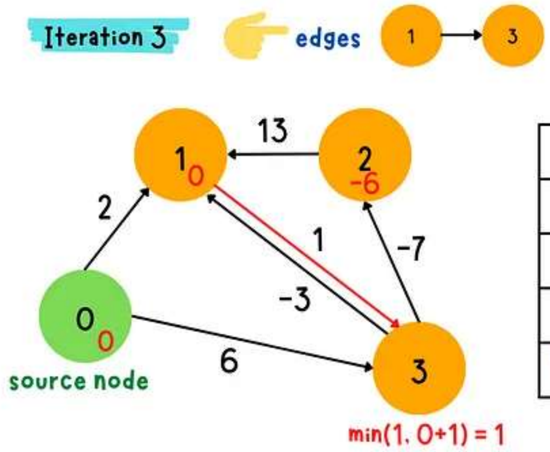
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	0	-4	3
iteration 2	0	-2	-6	1
iteration 3	0			

iteration 2–4



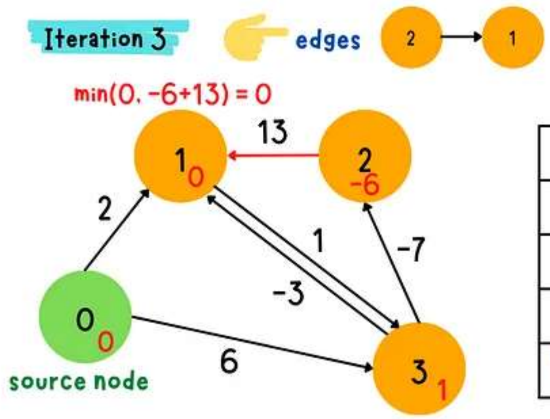
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	0		1

iteration 3–1



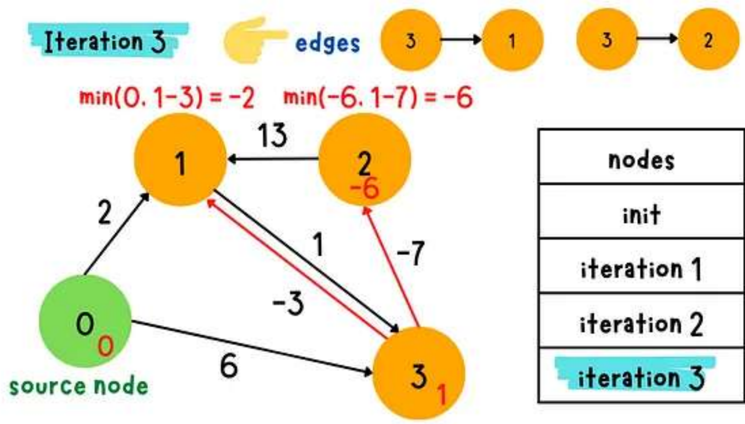
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	0		1

iteration 3–2



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	0		1

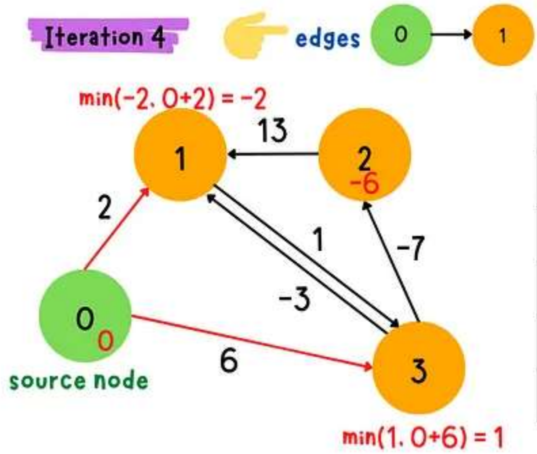
iteration 3–3



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	-2	-6	1

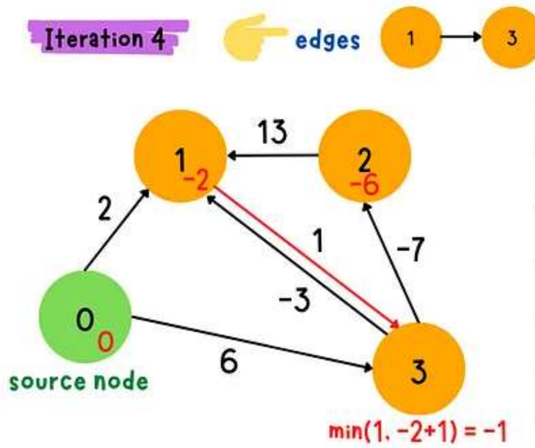
iteration 3–4

- *Step 4*: Perform the `nth` **iteration** to detect if a negative cycle is presented in the graph.



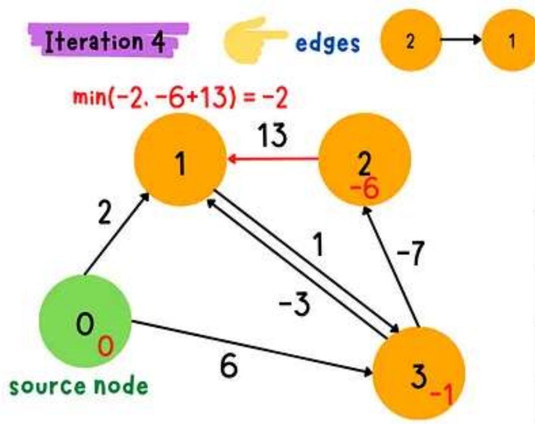
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	-2	-6	1
iteration 4	0	-2		1

iteration 4–1



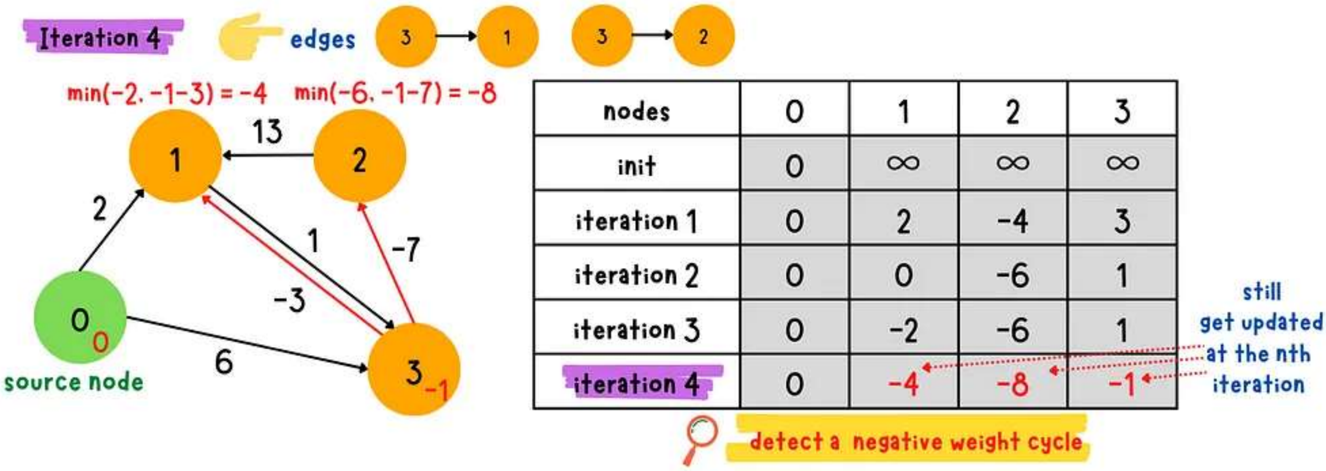
nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	-2	-6	1
iteration 4	0	-2		-1

iteration 4–2



nodes	0	1	2	3
init	0	∞	∞	∞
iteration 1	0	2	-4	3
iteration 2	0	0	-6	1
iteration 3	0	-2	-6	1
iteration 4	0	-2		-1

iteration 4–3



iteration 4–4

Code Implementation

Complexity

Time: O(v*e), Space: O(v)

v: the total number of vertices, e: the total number of edges

Golang