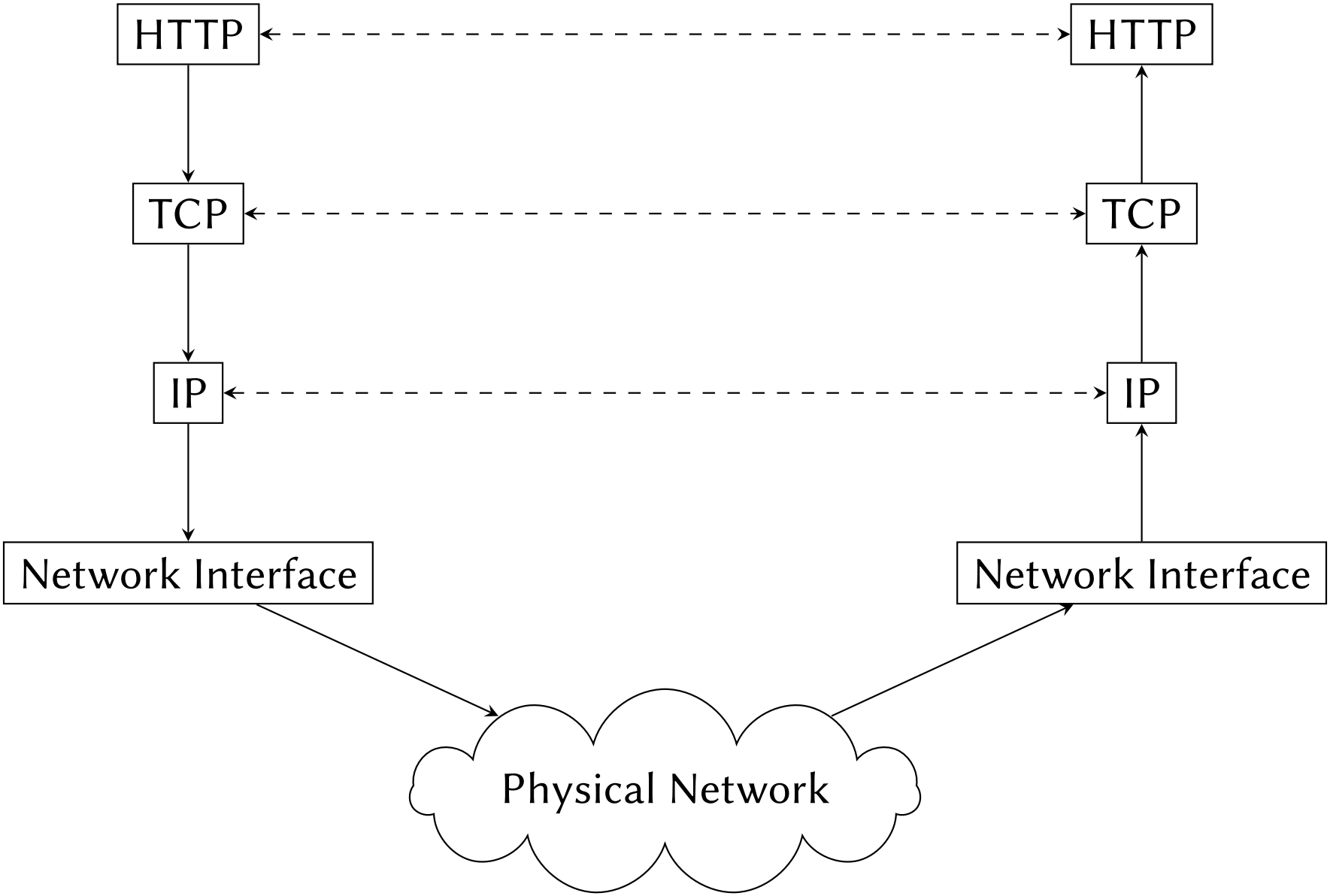
# Chapter 8:
# Routing, Arbitrage

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

# Outline

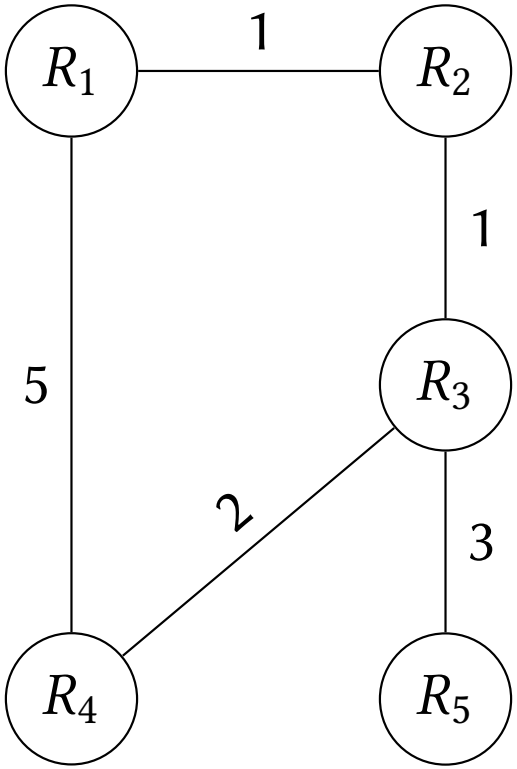
# The TCP/IP Protocol Stack with HTTP



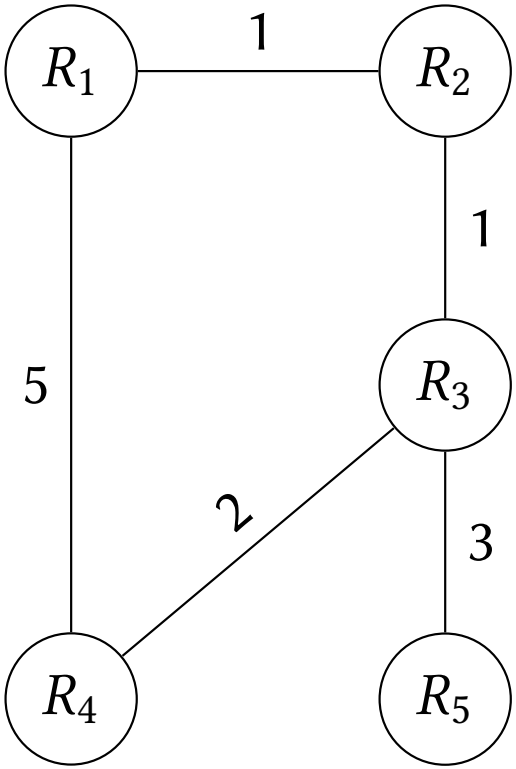
# Routing

- The Internet consists of a set of *autonomous systems*.
- Routing between autonomous systems is done with the Border Gateway Protocol (BGP).
- Routing inside an autonomous system is done with the Routing Information Protocol (RIP).

# An Autonomous System with Five Routers



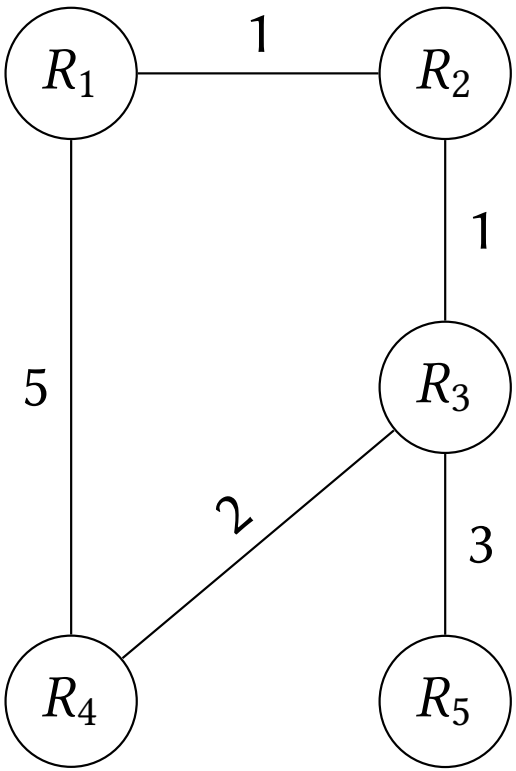
# RIP Example (1)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | - | 5/D | - |
| $R_2$ | 1/D | 0 | 1/D | - | - |
| $R_3$ | - | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | - | 2/D | 0 | - |
| $R_5$ | - | - | 3/D | - | 0 |

Initial state.

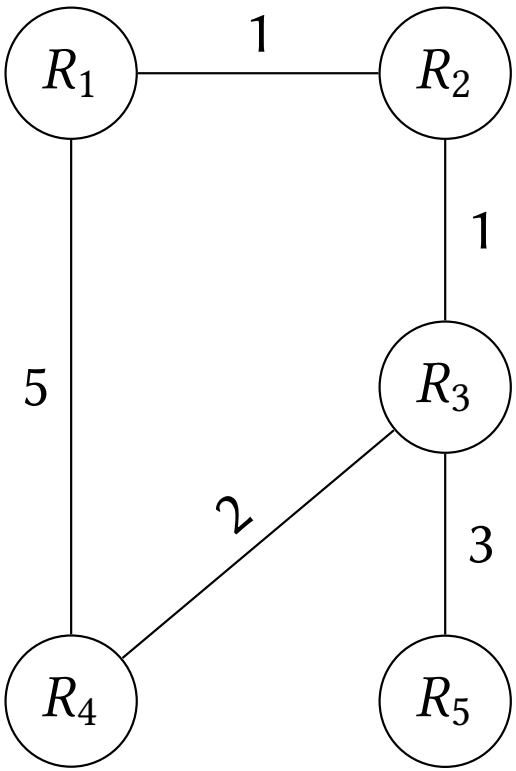
# RIP Example (2)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $\underline{2/R_2}$ | 5/D | - |
| $R_2$ | 1/D | 0 | 1/D | - | - |
| $R_3$ | - | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | - | 2/D | 0 | - |
| $R_5$ | - | - | 3/D | - | 0 |

$R_2 \longrightarrow R_1$.

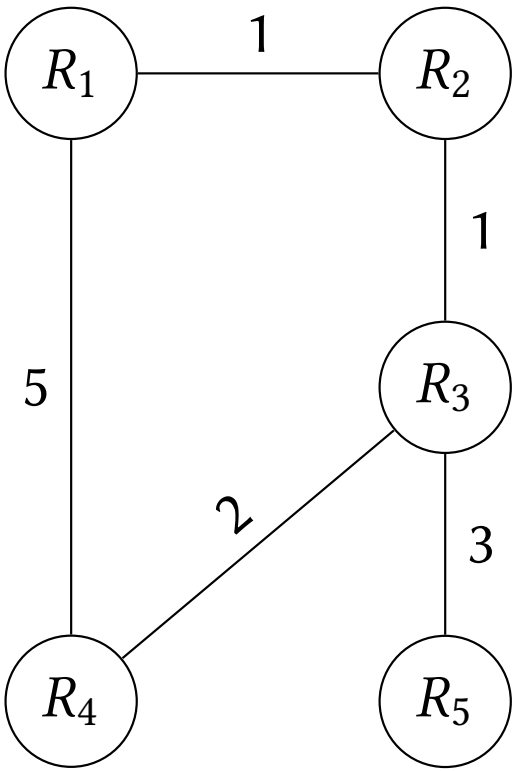
# RIP Example (3)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $2/R_2$ | 5/D | - |
| $R_2$ | 1/D | 0 | 1/D | $\underline{3/R_3}$ | $\underline{4/R_3}$ |
| $R_3$ | - | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | - | 2/D | 0 | - |
| $R_5$ | - | - | 3/D | - | 0 |

$R_3 \longrightarrow R_2$.

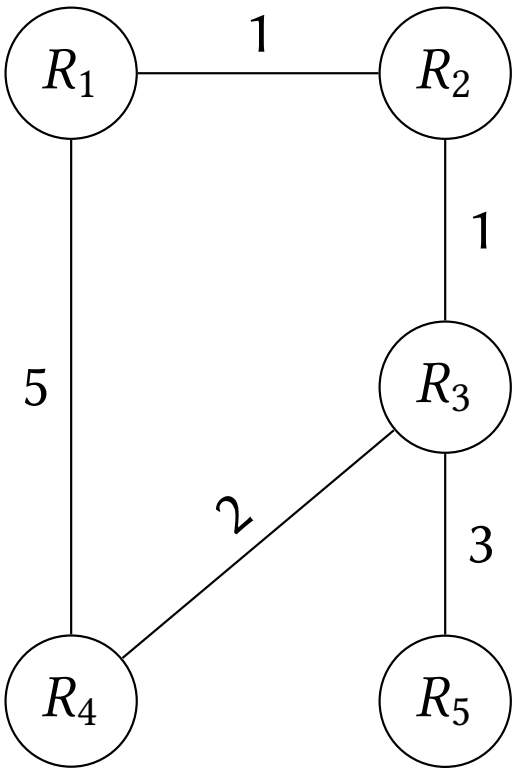
# RIP Example (4)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $2/R_2$ | $\underline{4/R_2}$ | $\underline{5/R_2}$ |
| $R_2$ | 1/D | 0 | 1/D | $3/R_3$ | $4/R_3$ |
| $R_3$ | - | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | - | 2/D | 0 | - |
| $R_5$ | - | - | 3/D | - | 0 |

$R_2 \longrightarrow R_1.$

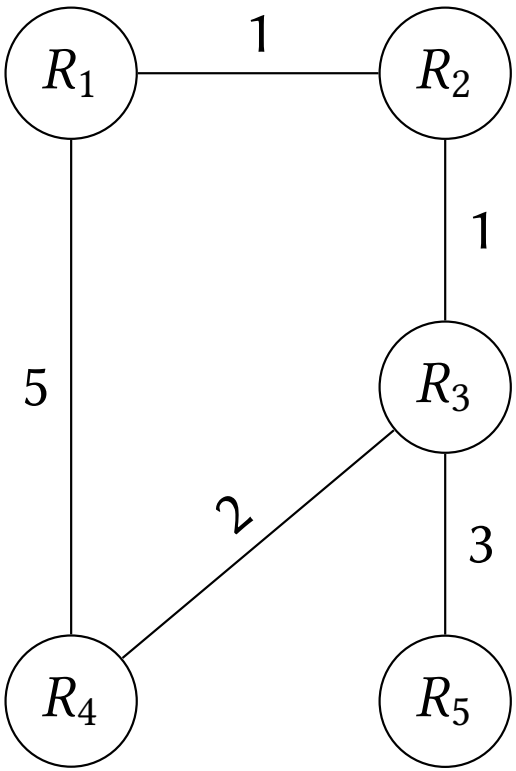
# RIP Example (5)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $2/R_2$ | $4/R_2$ | $5/R_2$ |
| $R_2$ | 1/D | 0 | 1/D | $3/R_3$ | $4/R_3$ |
| $R_3$ | - | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | $\underline{3/R_3}$ | 2/D | 0 | $\underline{5/R_3}$ |
| $R_5$ | - | - | 3/D | - | 0 |

$$R_3 \longrightarrow R_4.$$

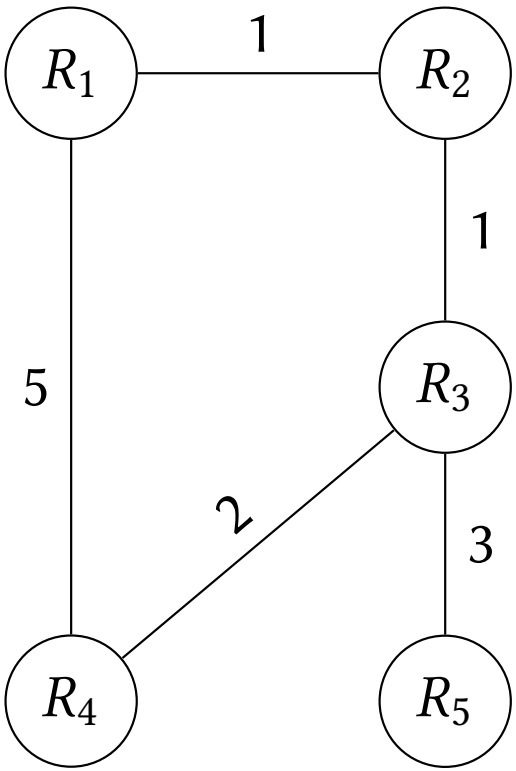
# RIP Example (6)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $2/R_2$ | $4/R_2$ | $5/R_2$ |
| $R_2$ | 1/D | 0 | 1/D | $3/R_3$ | $4/R_3$ |
| $R_3$ | - | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | $3/R_3$ | 2/D | 0 | $5/R_3$ |
| $R_5$ | - | $\underline{4/R_3}$ | 3/D | $\underline{5/R_3}$ | 0 |

$R_3 \longrightarrow R_5$.

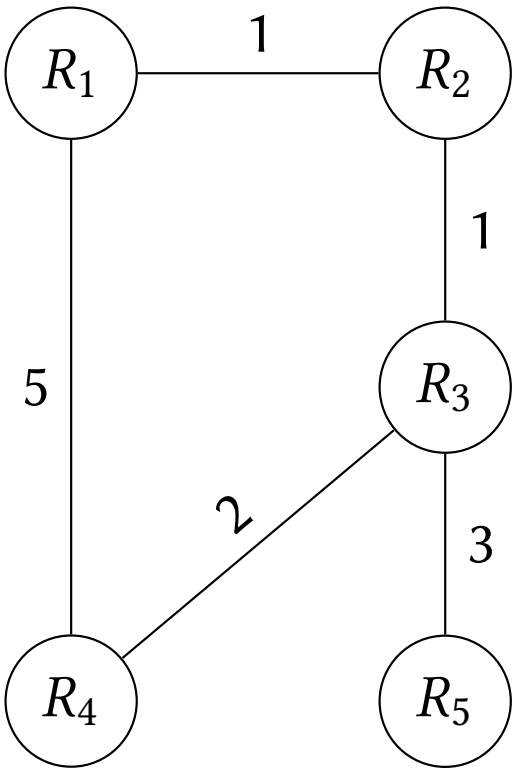
# RIP Example (7)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $2/R_2$ | $4/R_2$ | $5/R_2$ |
| $R_2$ | 1/D | 0 | 1/D | $3/R_3$ | $4/R_3$ |
| $R_3$ | $\underline{2/R_2}$ | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | $3/R_3$ | 2/D | 0 | $5/R_3$ |
| $R_5$ | - | $4/R_3$ | 3/D | $5/R_3$ | 0 |

$R_2 \longrightarrow R_3$.

# RIP Example (8)




| | $R_1$ | $R_2$ | $R_3$ | $R_4$ | $R_5$ |
|---|---|---|---|---|---|
| $R_1$ | 0 | 1/D | $2/R_2$ | $4/R_2$ | $5/R_2$ |
| $R_2$ | 1/D | 0 | 1/D | $3/R_3$ | $4/R_3$ |
| $R_3$ | $2/R_2$ | 1/D | 0 | 2/D | 3/D |
| $R_4$ | 5/D | $3/R_3$ | 2/D | 0 | $5/R_3$ |
| $R_5$ | $\underline{5/R_3}$ | $4/R_3$ | 3/D | $5/R_3$ | 0 |

$R_3 \longrightarrow R_5$.

# Outline

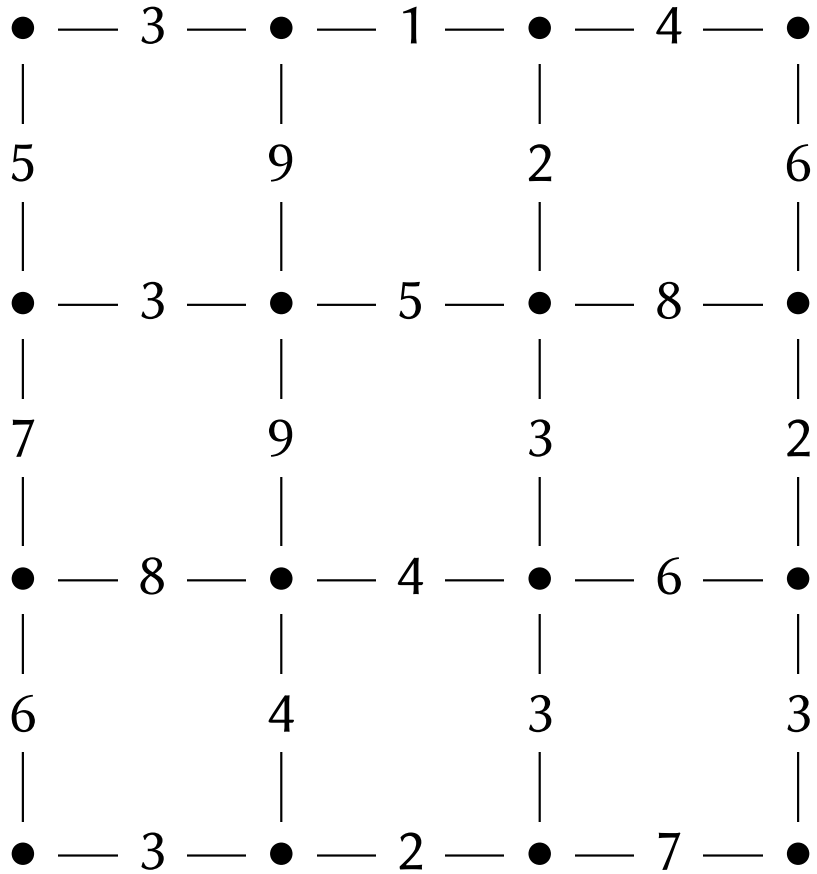
# Basic Idea

- The algorithm is named after Richard Bellman and Lester Ford, Jr., who published the algorithm in 1958 and 1956.
- It is also called Bellman-Ford-Moore because Edward F. Moore also published the algorithm in 1957.
- We initialize the graph with estimates for the shortest paths from the starting node.
- The estimate for the starting node is 0, for all other nodes it is $\infty$.
- We check all the edges of the graph and we update (relax) the estimates for all nodes.
- We repeat the process $|V| - 1$ times, where $|V|$ is the number of vertices in the graph.

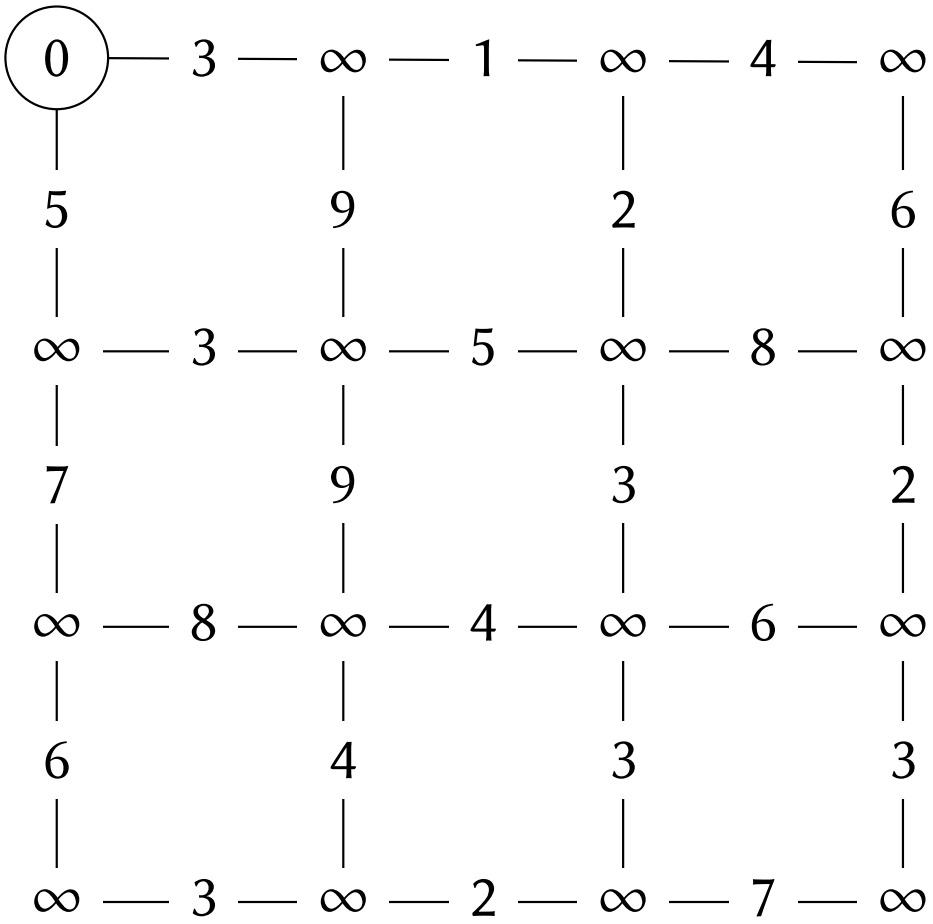
# Bellman-Ford and RIP

- RIP is a distributed version of the Bellman-Ford algorithm.
- Each time a node receives a packet, it updates its state of knowledge regarding shortest paths, incorporating paths with ever-increasing number of edges.
- After some time, it will have update its state of knowledge for shortest paths containing up to $|V| - 1$ edges.

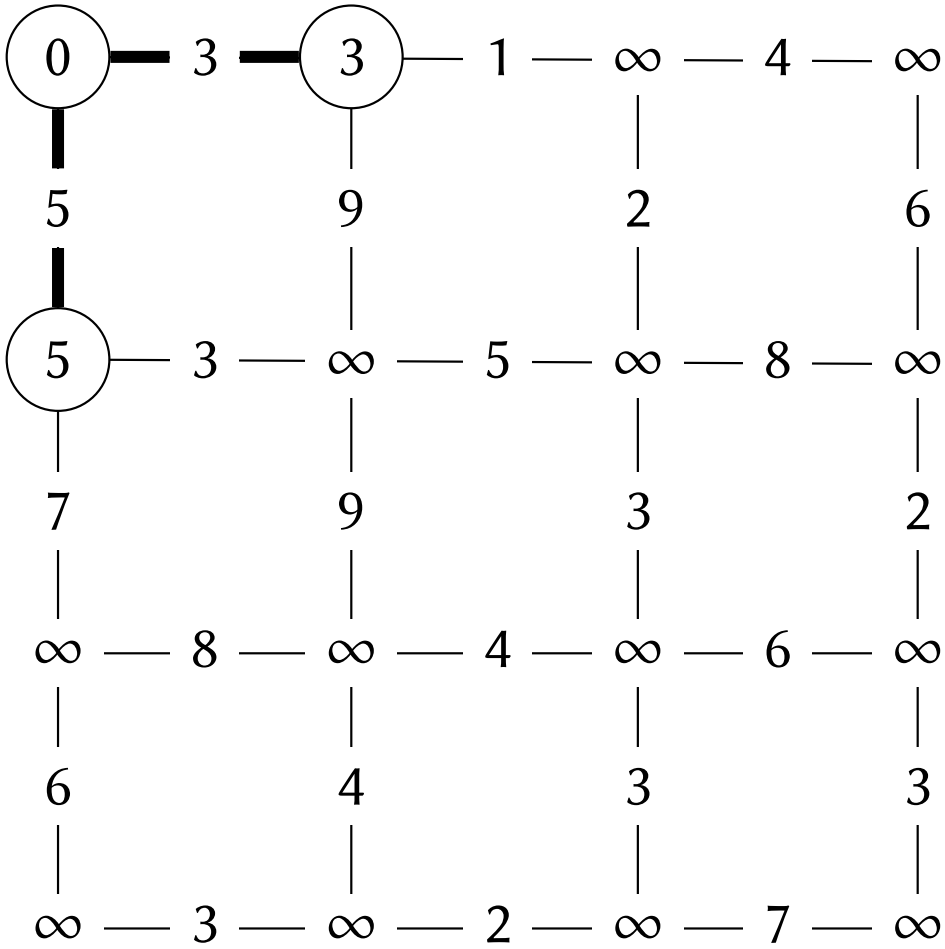
# A Traffic Grid



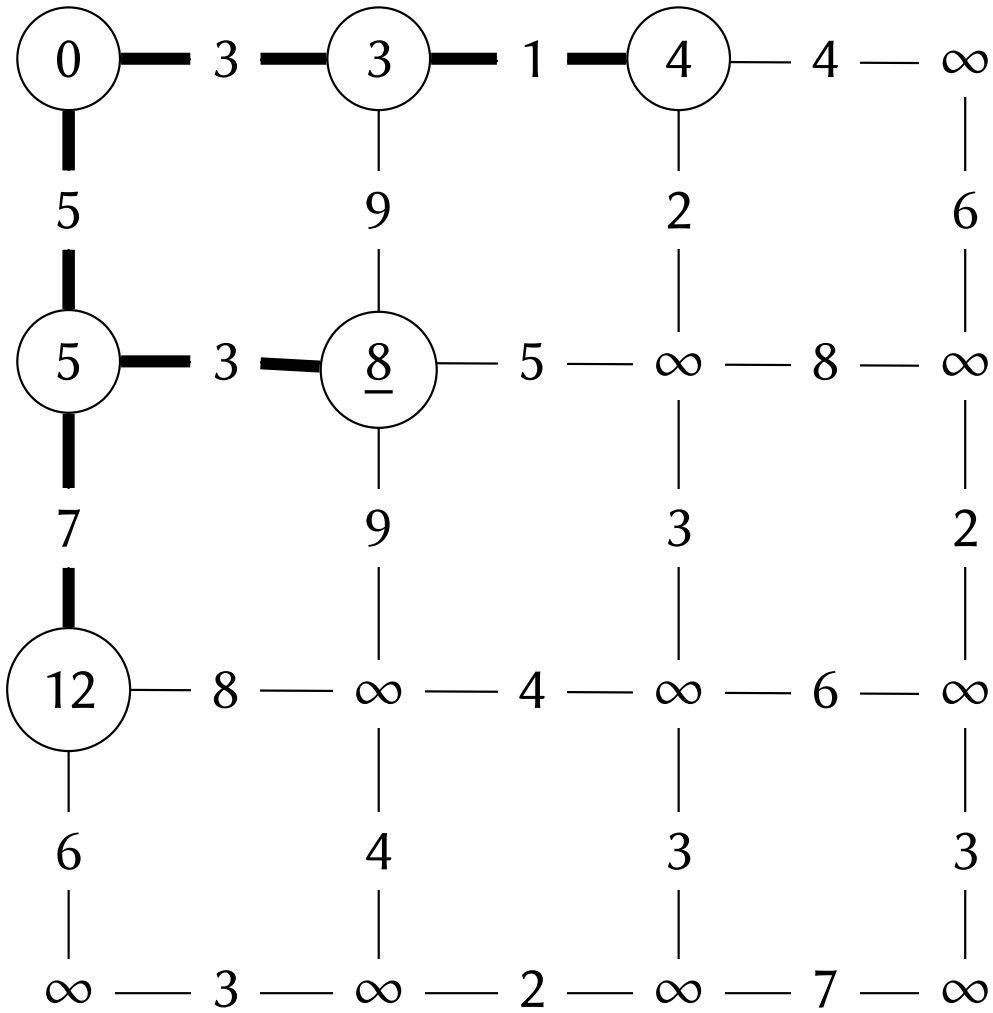
# Bellman-Ford Example (1)



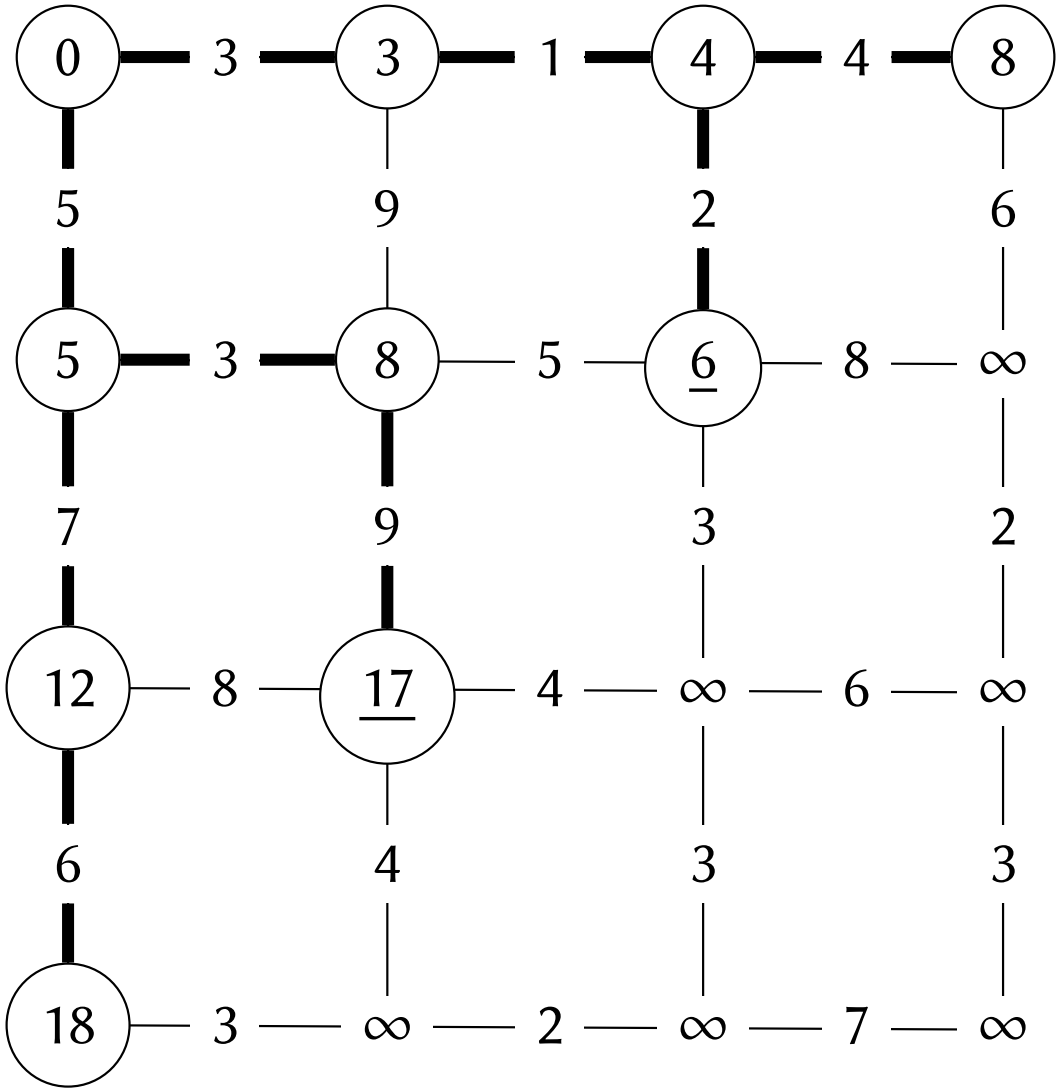
# Bellman-Ford Example (2)



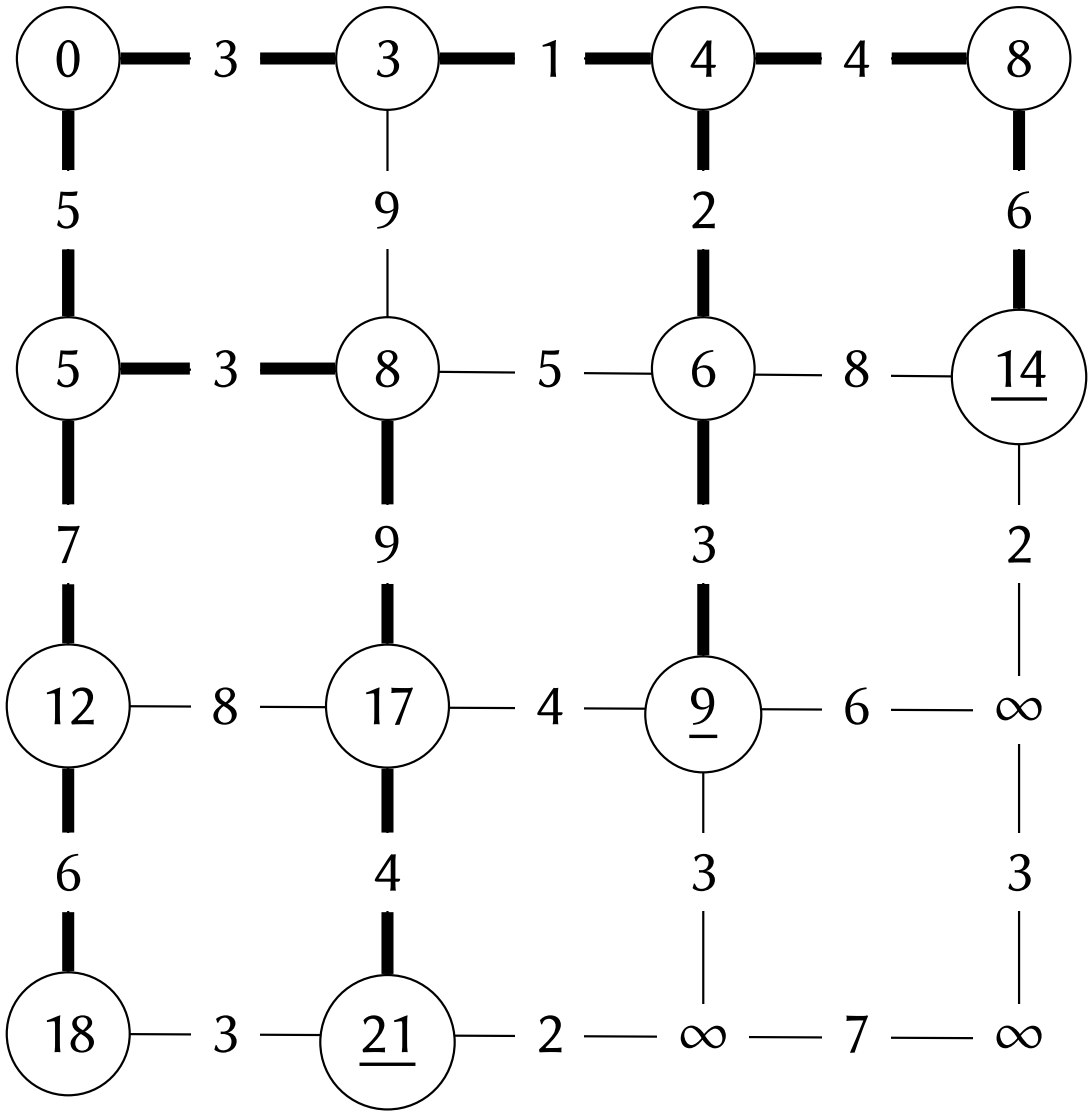
# Bellman-Ford Example (3)



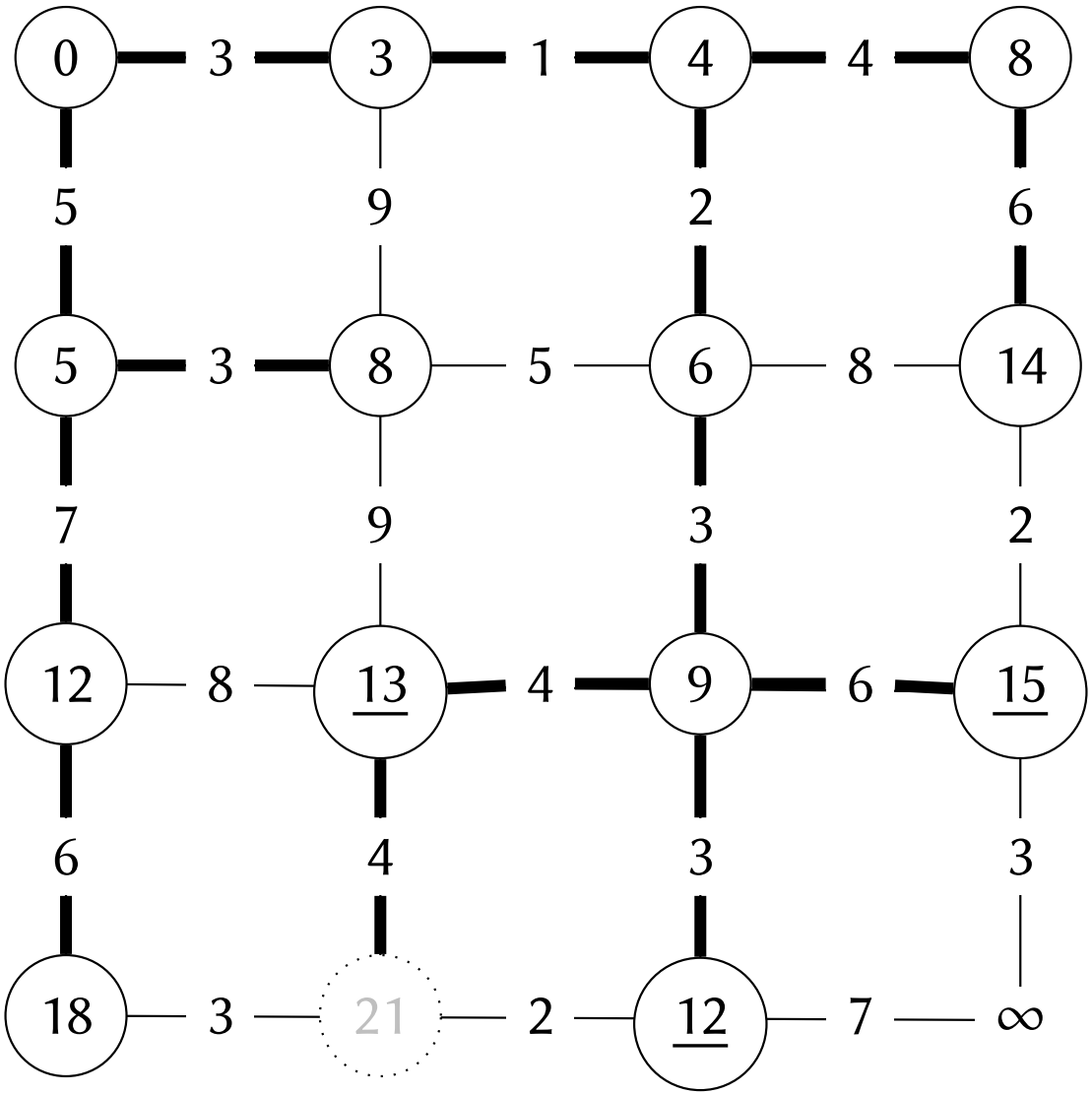
# Bellman-Ford Example (4)



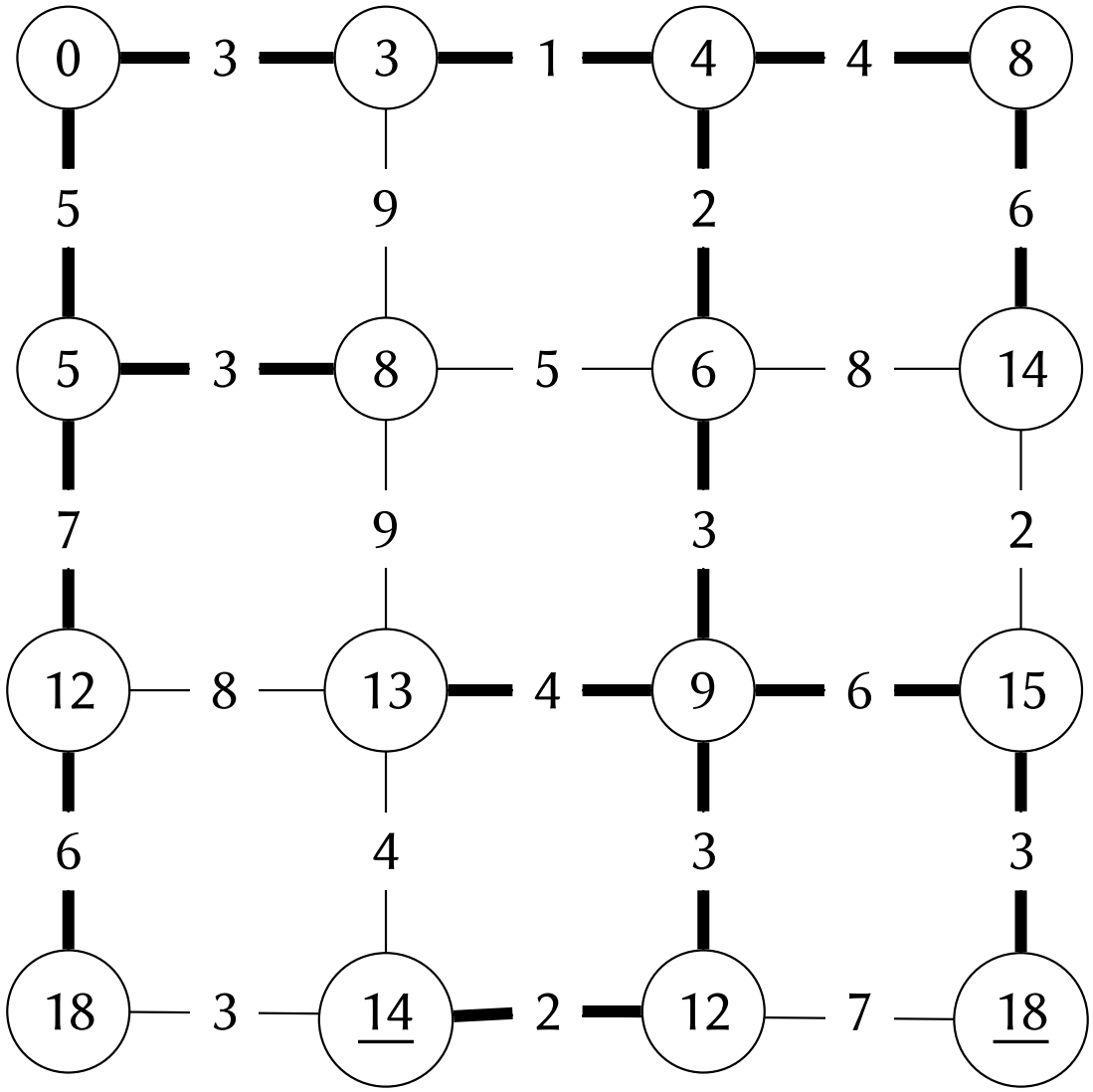
# Bellman-Ford Example (5)



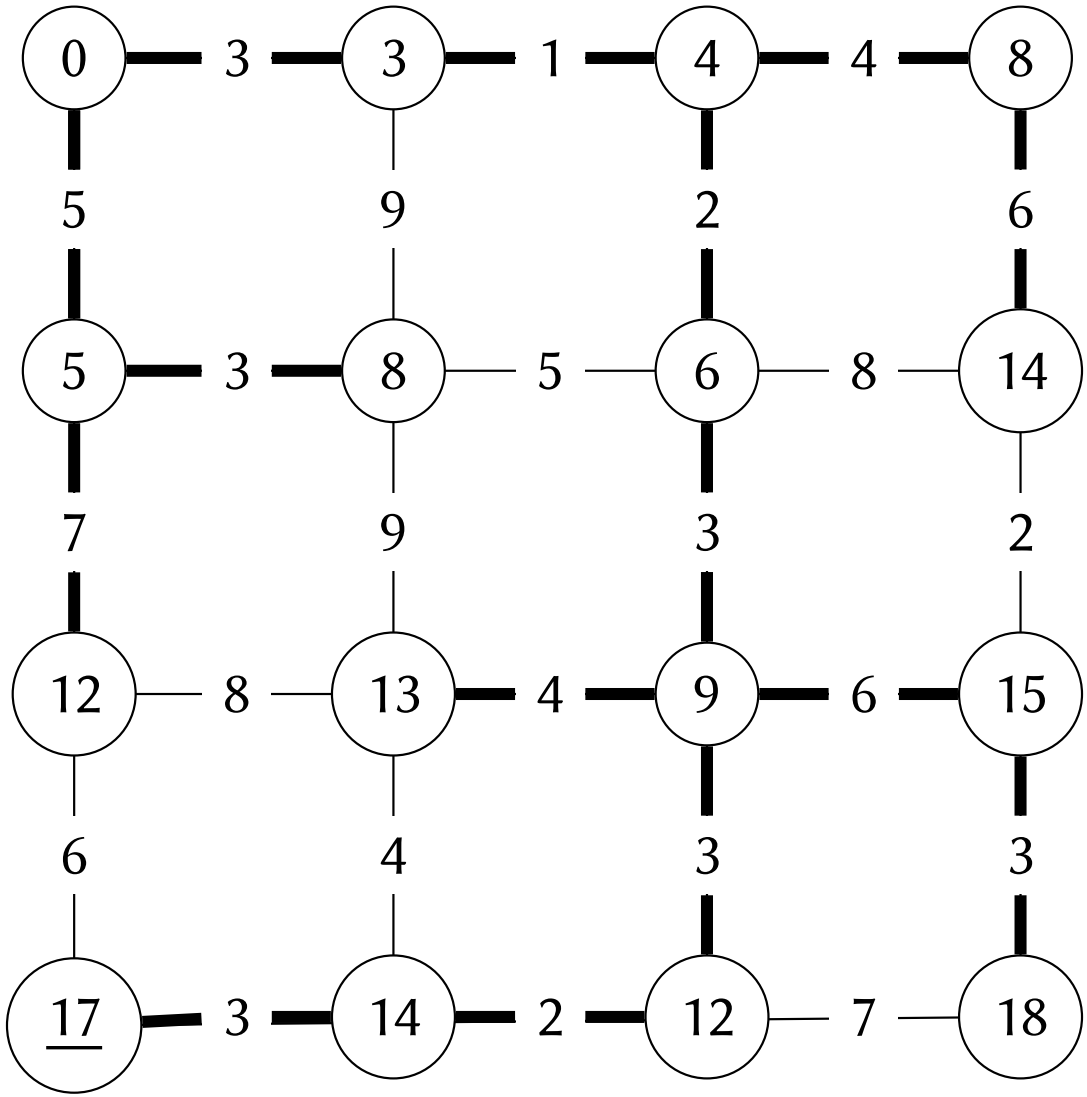
# Bellman-Ford Example (6)



# Bellman-Ford Example (7)



# Bellman-Ford Example (8)



# The Algorithm

**Algorithm:** Bellman-Ford.

`BellmanFord`$(G, s) \rightarrow (pred, dist)$

**Input:** $G = (V, E)$, a graph
$s$, the starting node

**Output:** $pred$, an array of size $|V|$ such that $pred[i]$ is the predecessor of node $i$ in the shortest path from $s$
$dist$, an array of size $|V|$ such that $dist[i]$ is the length of the shortest path calculated from node $s$ to $i$

```
1  pred ← CreateArray(|V|)
2  dist ← CreateArray(|V|)
3  foreach v in V do
4      pred[v] ← −1
5      if v ≠ s then
6          dist[v] ← ∞
7      else
8          dist[v] ← 0
9  for i ← 0 to |V| do
10     foreach (u, v) in E do
11         if dist[v] > dist[u] + Weight(G, u, v) then
12             dist[v] ← dist[u] + Weight(G, u, v)
13             pred[v] ← u
14 return (pred, dist)
```

# Algorithm Complexity

- The initialization requires $|V|$ iterations.
- Initialization operations require constant time, i.e., $O(1)$.
- So the initialization requires time $O(|V|)$.
- The loop in lines 9–13 is repeated $|V| - 1$ times.
- Each time we check all edges, requiring time $O((|V| - 1)|E|) = O(|V||E|)$.
- The total time required by the algorithm is $O(|V| + |V||E|)$, that is, $O(|V||E|)$.

# Improvements

- In its present form, the algorithm checks in each iteration all edges.
- However, in the $i$th iteration the algorithm finds the shortest paths consisting of up to $i$ edges.
- So, in the $(i+1)$th iteration the algorithm will only need to check the edges of the nodes that were updated in the $i$ iteration.
- Therefore, we can keep these nodes in a queue and check only them.

# Queue-based Bellman-Ford

**Algorithm:** Queue-based Bellman-Ford.

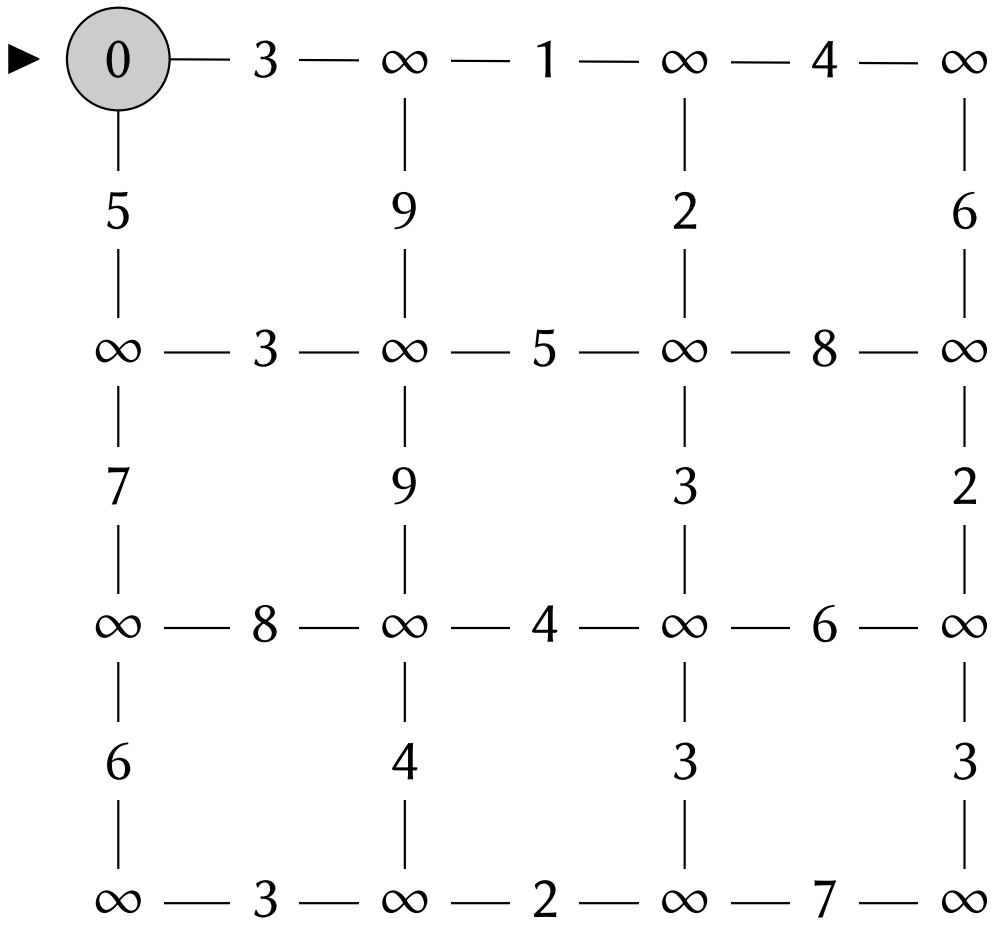
`BellmanFordQueue`$(G, s) \rightarrow (pred, dist)$

**Input:** $G = (V, E)$, a graph
$s$, the starting node

**Output:** $pred$, an array of size $|V|$ such that $pred[i]$ is the predecessor of node $i$ in the shortest path from $s$
$dist$, an array of size $|V|$ such that $dist[i]$ is the length of the shortest path calculated from node $s$ to $i$

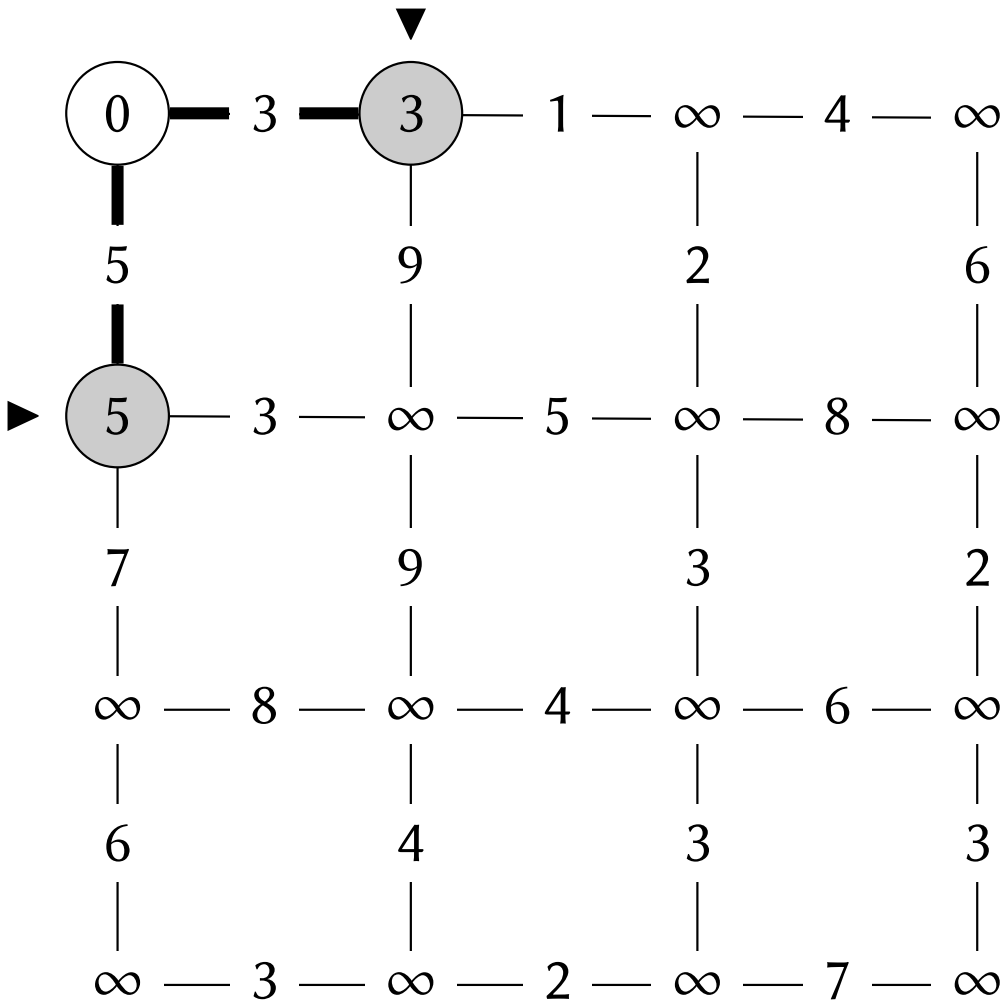
```
1  inqueue ← CreateArray(|V|)
2  Q ← CreateQueue()
3  foreach v in V do
4      pred[v] ← −1
5      if v ≠ s then
6          dist[v] ← ∞
7          inqueue[v] ← FALSE
8      else
9          dist[v] ← 0
10 Enqueue(Q, s)
11 inqueue[s] ← TRUE
12 while Size(Q) ≠ 0 do
13     u ← Dequeue(Q)
14     inqueue[u] ← FALSE
15     foreach v in AdjacencyList(G, u) do
16         if dist[v] > dist[u] + Weight(G, u, v) then
17             dist[v] ← dist[u] + Weight(G, u, v)
18             pred[v] ← u
19             if not inqueue[v] then
20                 Enqueue(Q, v)
21                 inqueue[v] ← TRUE
22 return (pred, dist)
```

# Bellman-Ford with Queue Example (1)



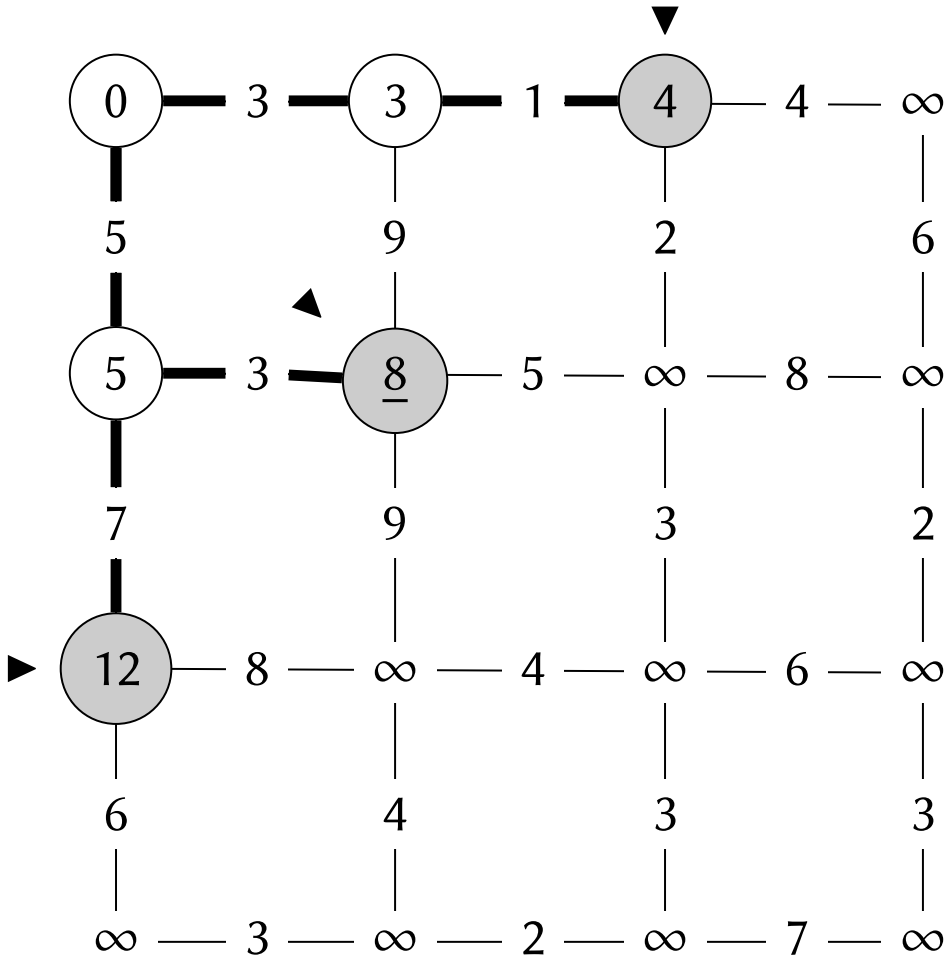

(0, 0)

# Bellman-Ford with Queue Example (2)



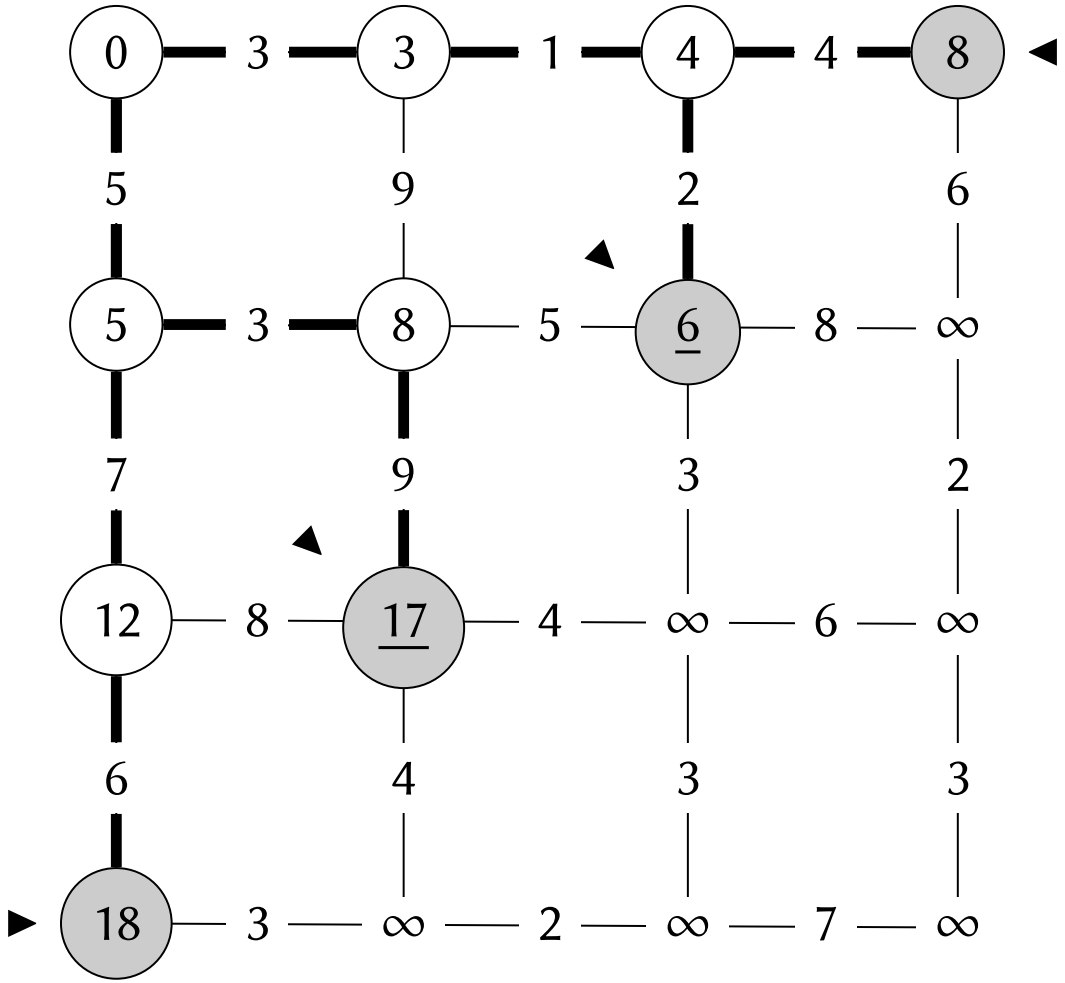

| (0, 1) | (1, 0) |
|---|---|

# Bellman-Ford with Queue Example (3)



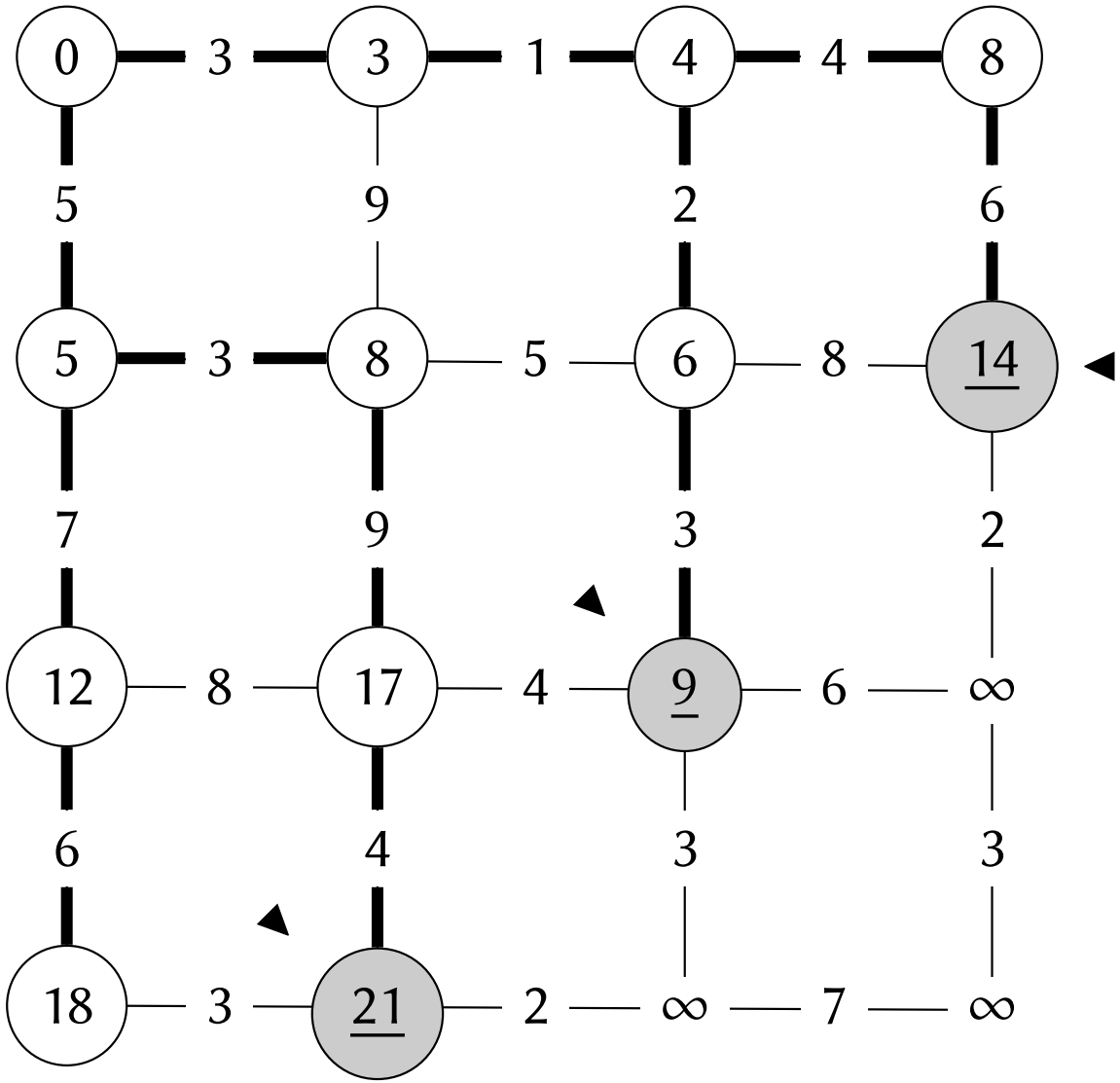

| (0, 2) | (1, 1) | (2, 0) |
|---|---|---|

# Bellman-Ford with Queue Example (4)



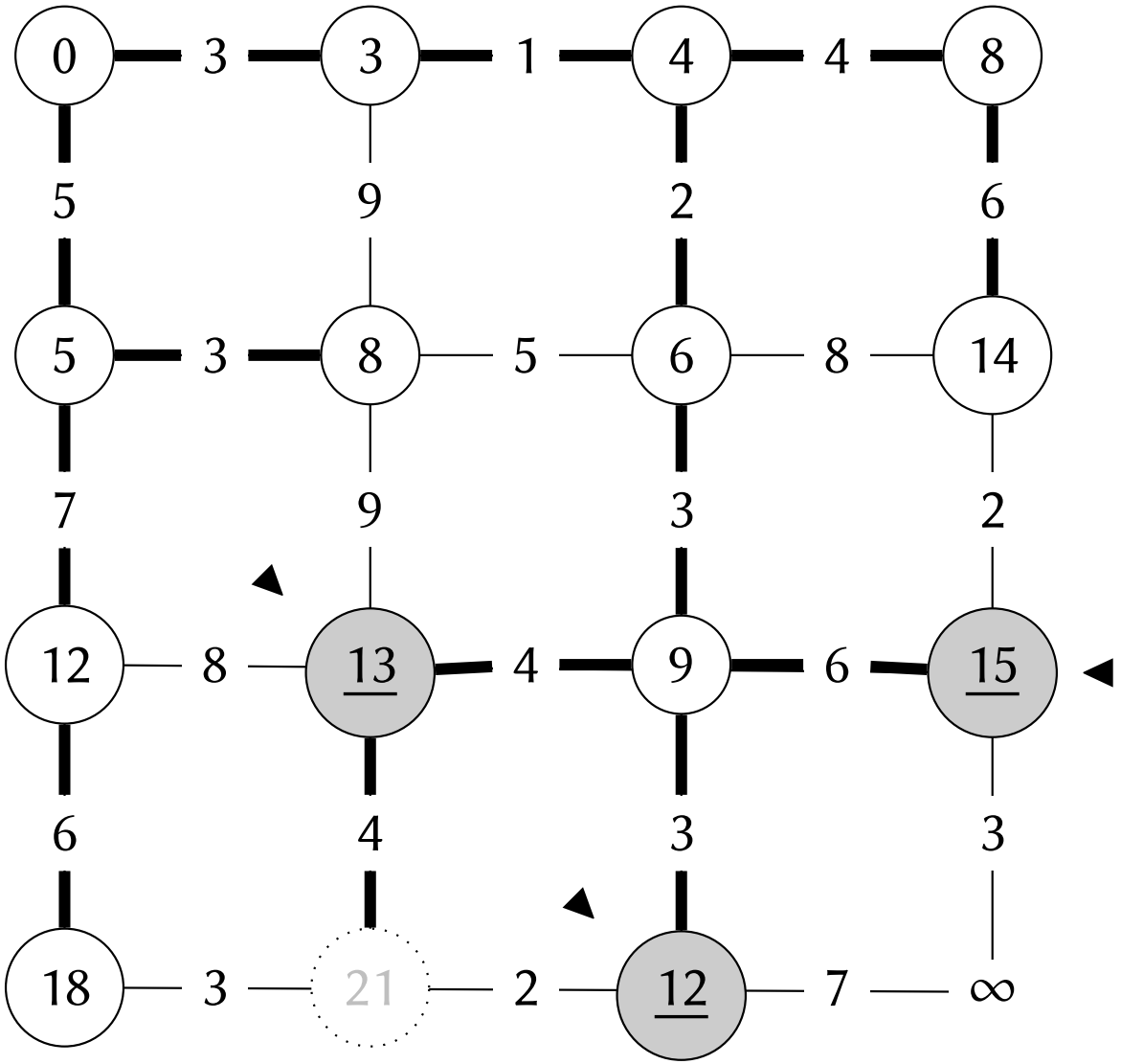

| (0, 3) | (2, 1) | (1, 2) | (3, 0) |
|---|---|---|---|

# Bellman-Ford with Queue Example (5)



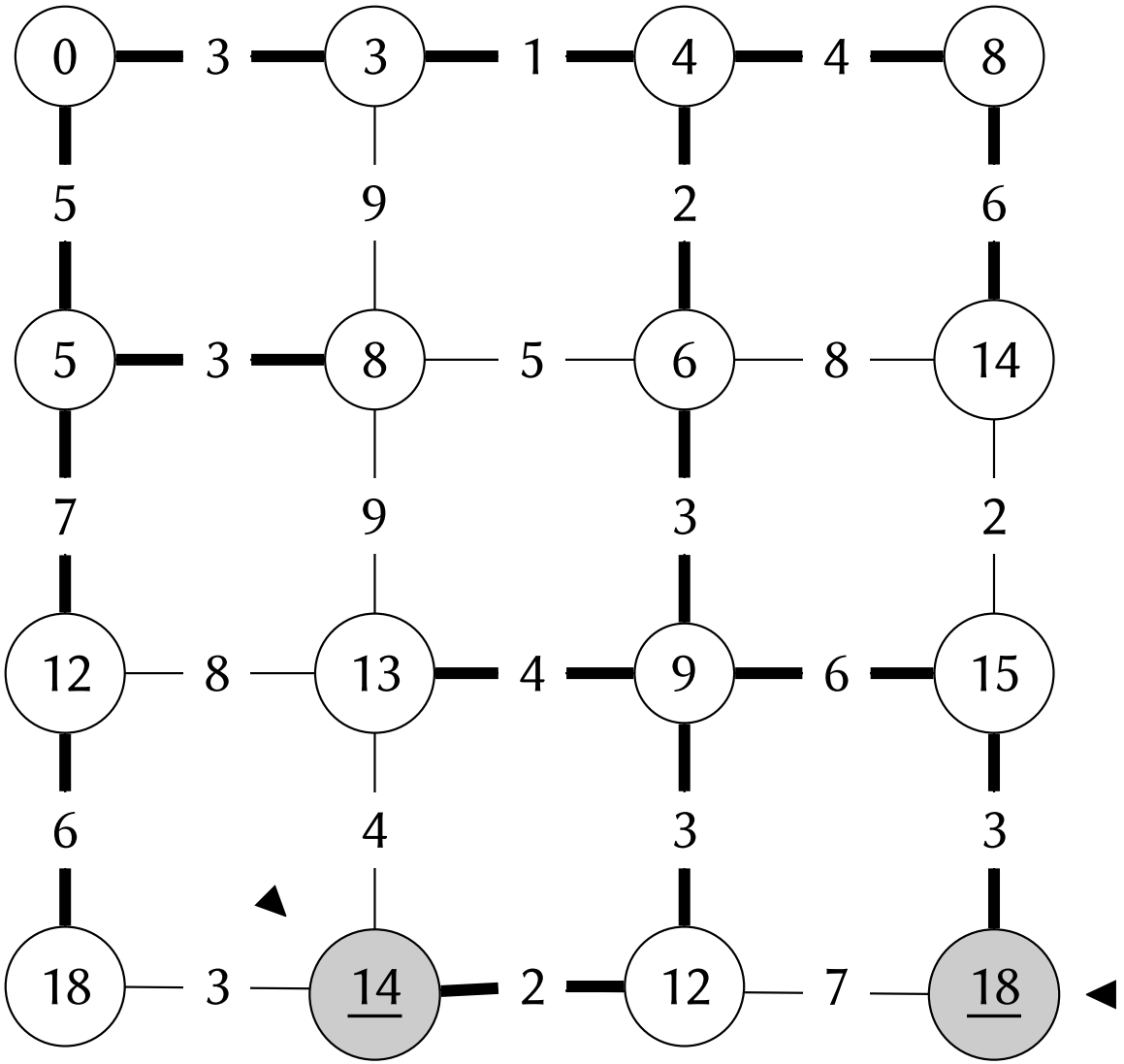

| (1, 3) | (2, 2) | (3, 1) |
|---|---|---|

# Bellman-Ford with Queue Example (6)

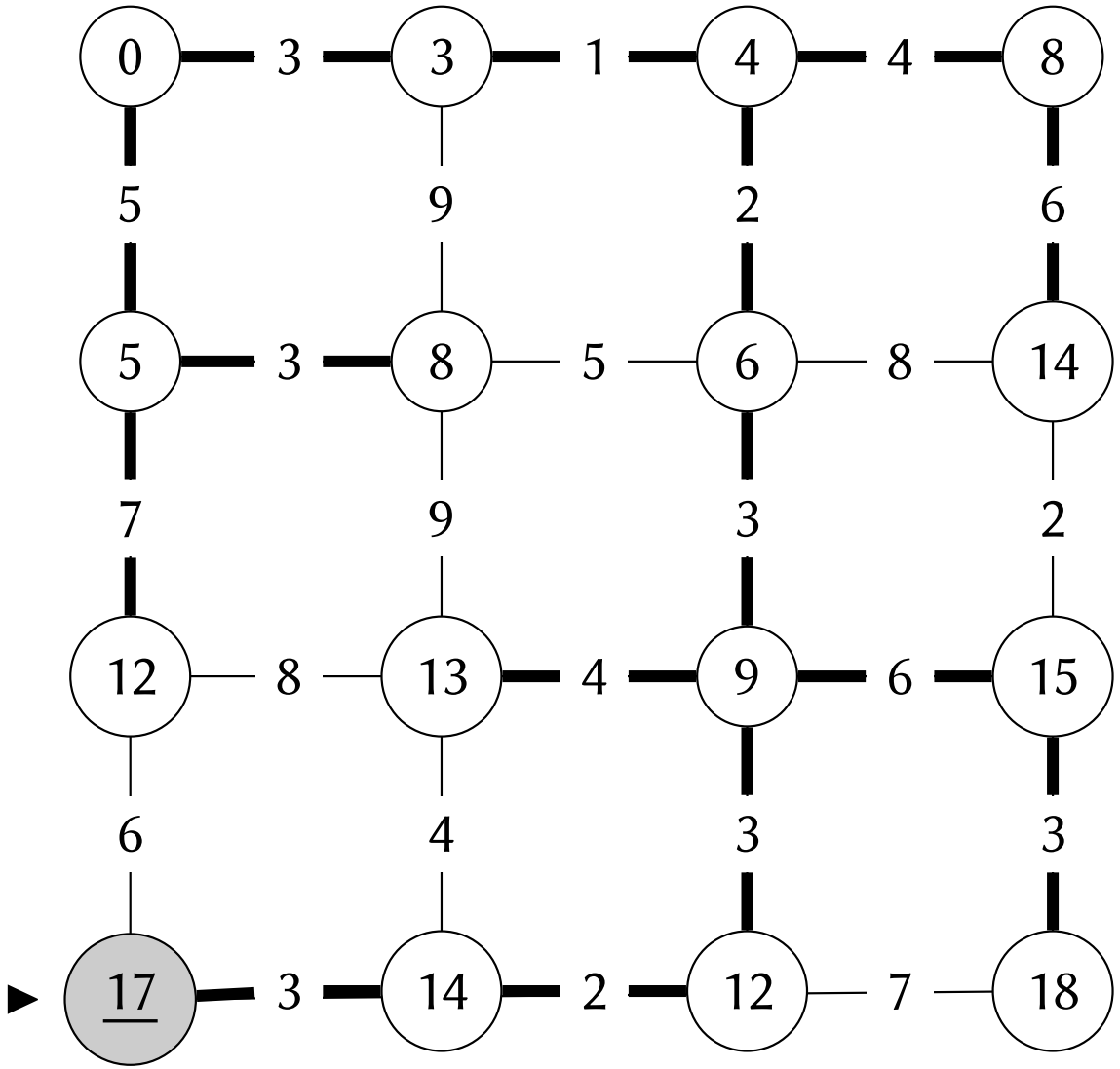



| (2, 3) | (2, 1) | (3, 2) |
|---|---|---|

# Bellman-Ford with Queue Example (7)

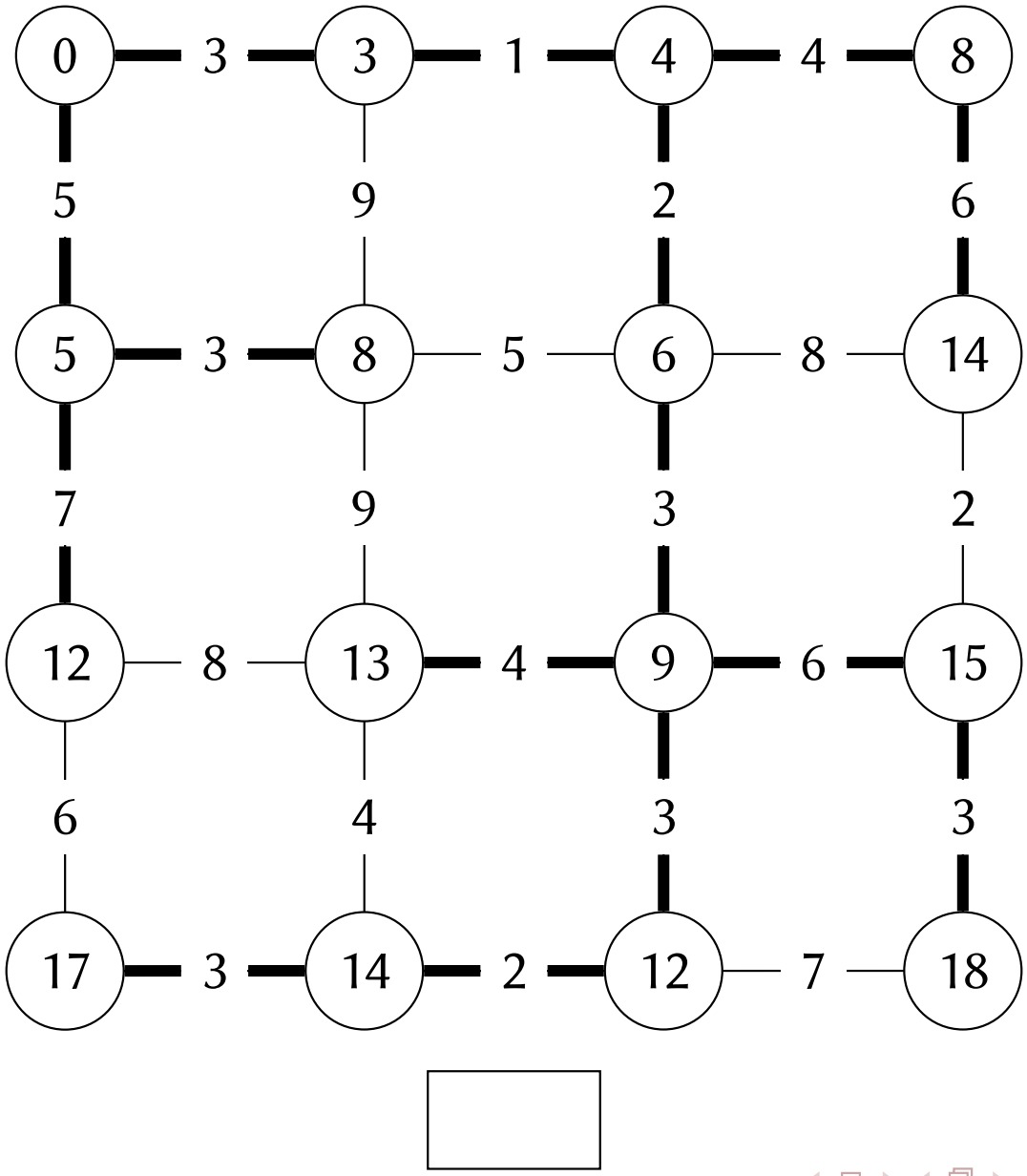


# Bellman-Ford with Queue Example (8)




(3, 0)

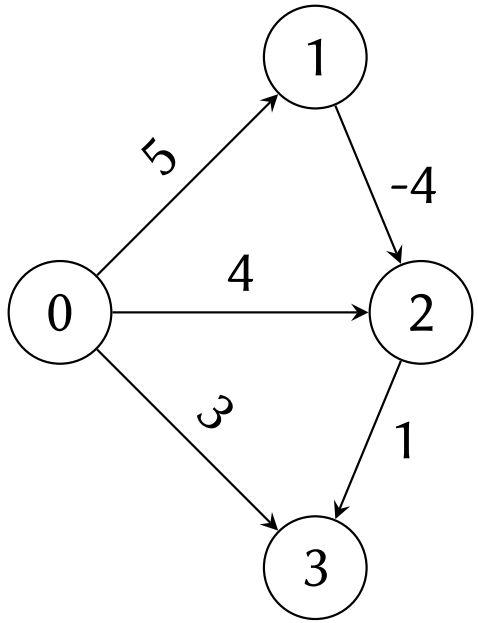
# Bellman-Ford with Queue Example (9)



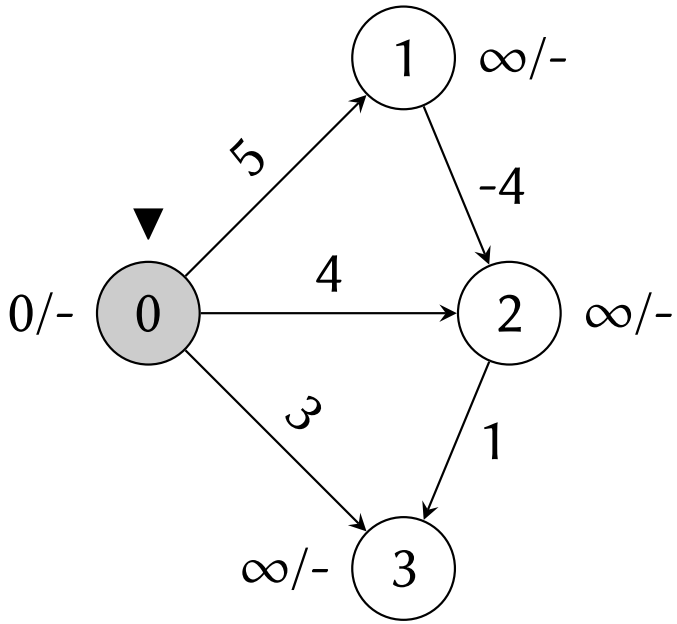
# Advantages of Bellman-Ford

- The use of the queue does not improve the worst case performance of Bellman-Ford, which remains $O(|V||E|)$.
- That is worse than the time required by Dijkstra's algorithm, which is $O(|E| \lg |V|)$.
- However, Bellman-Ford can be used with negative weights, while Dijkstra's algorithm cannot.

# Graph with Negative Weights

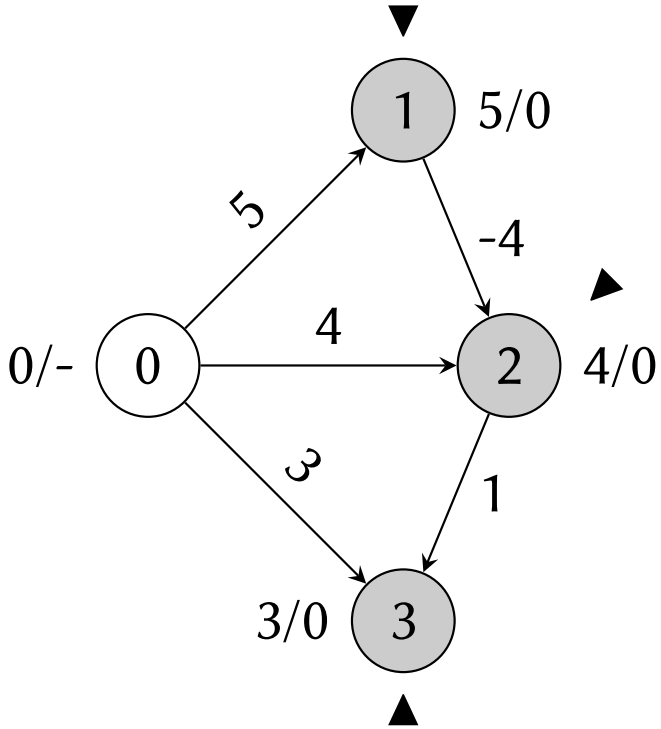


# Bellman-Ford with Negative Weights Example (1)



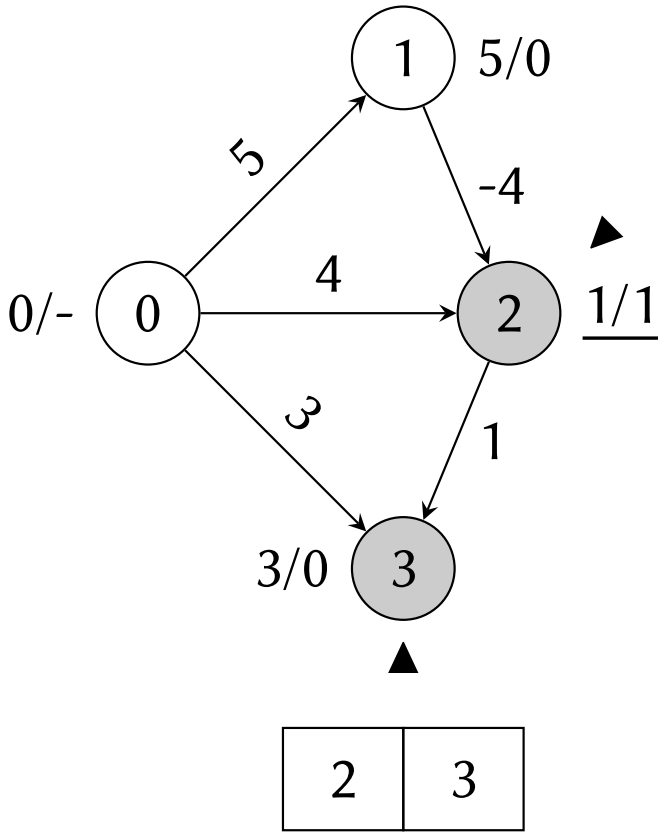

| 0 |
|---|

# Bellman Ford with Negative Weights Example (2)

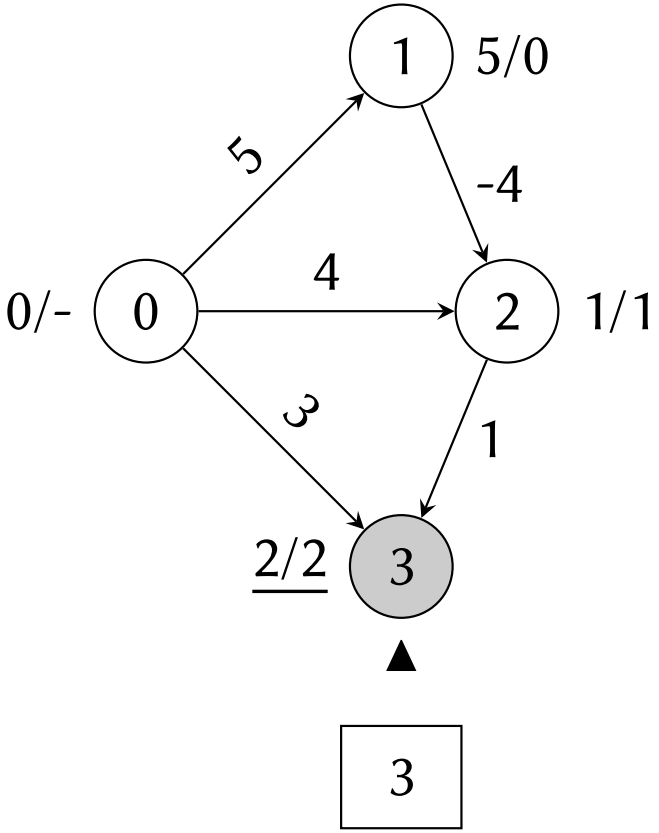



| 1 | 2 | 3 |
|---|---|---|

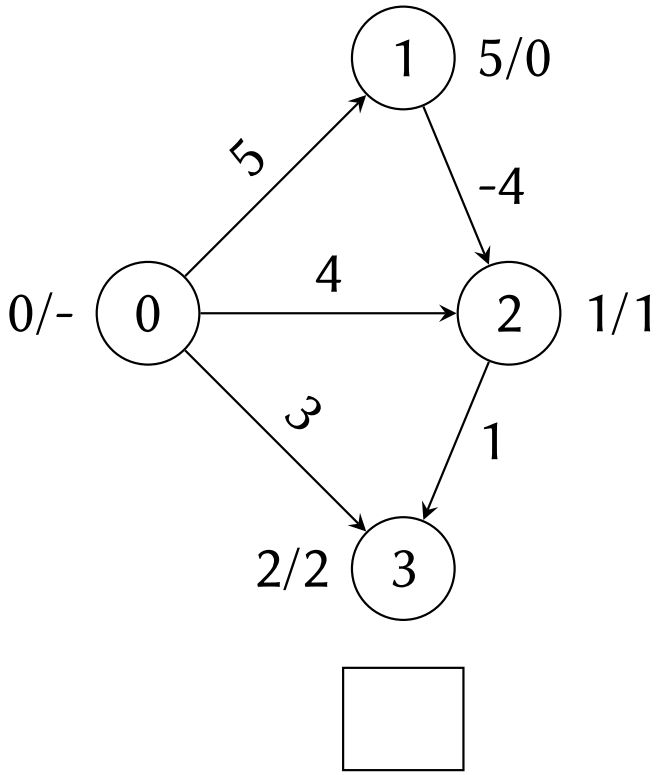
# Bellman-Ford with Negative Weights Example (3)



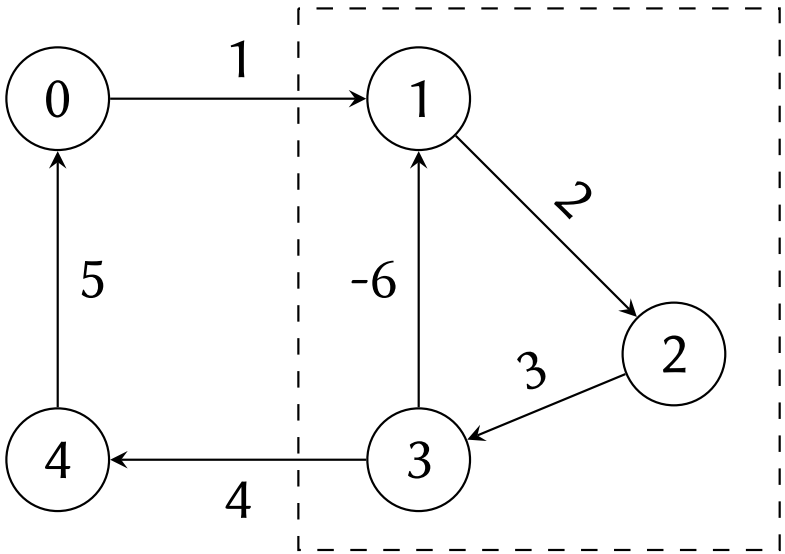
# Bellman-Ford with Negative Weights Example (4)



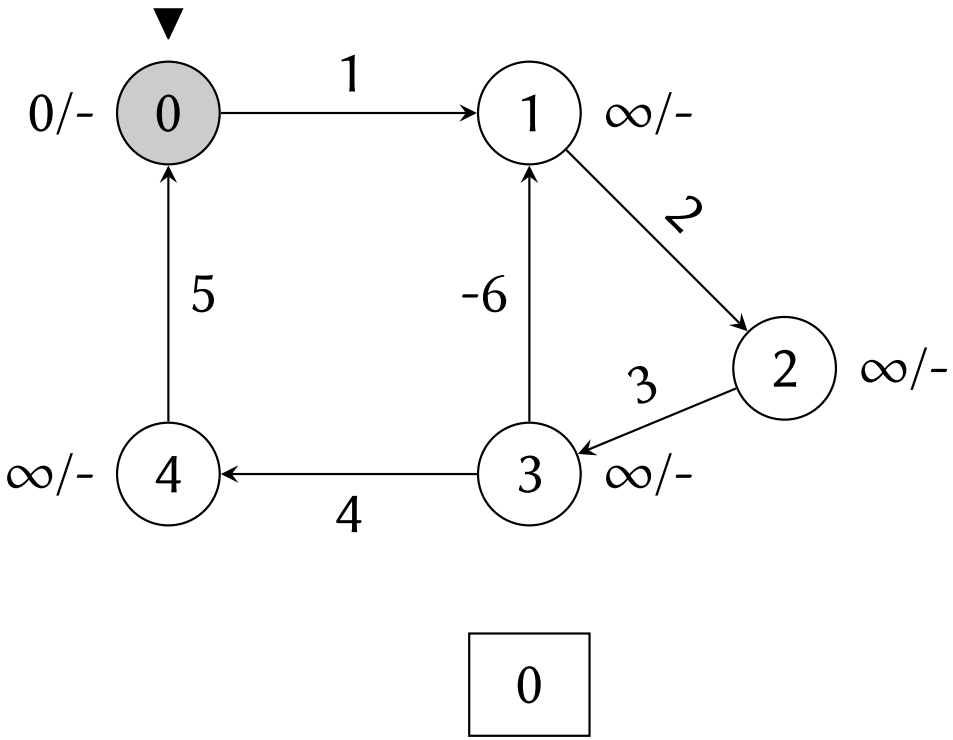
# Bellman-Ford with Negative Weights Example (5)



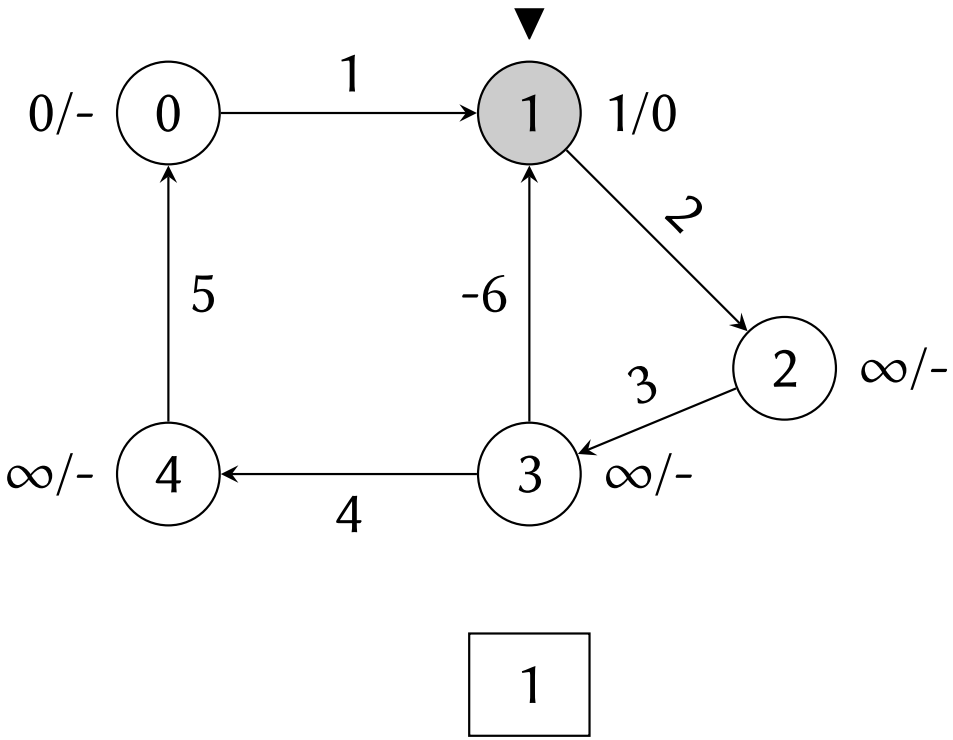
# Graphs with Negative Cycles



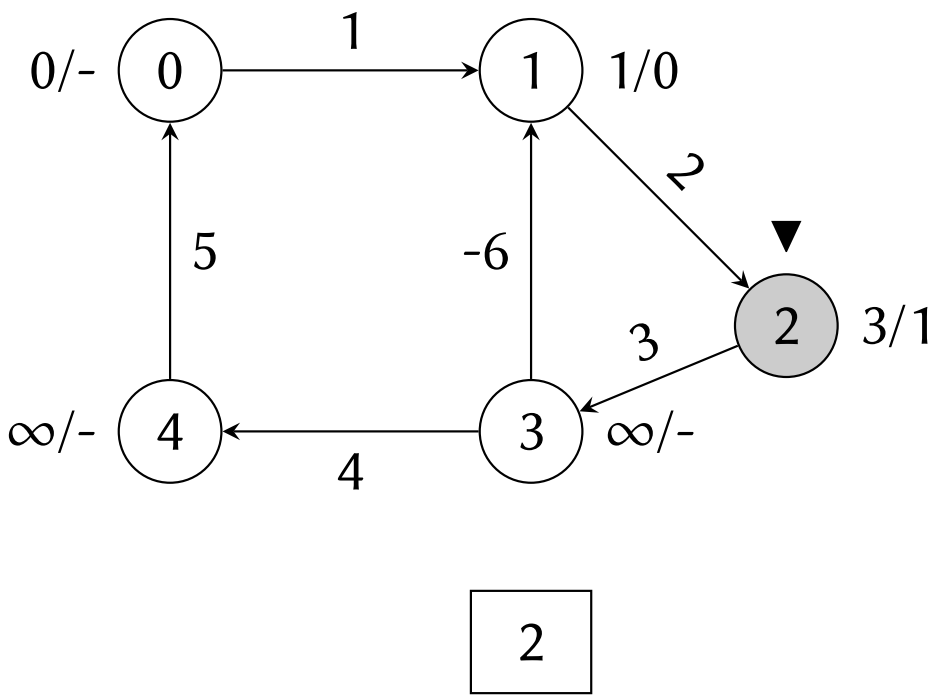
# Bellman-Ford with Negative Cycles Example (1)



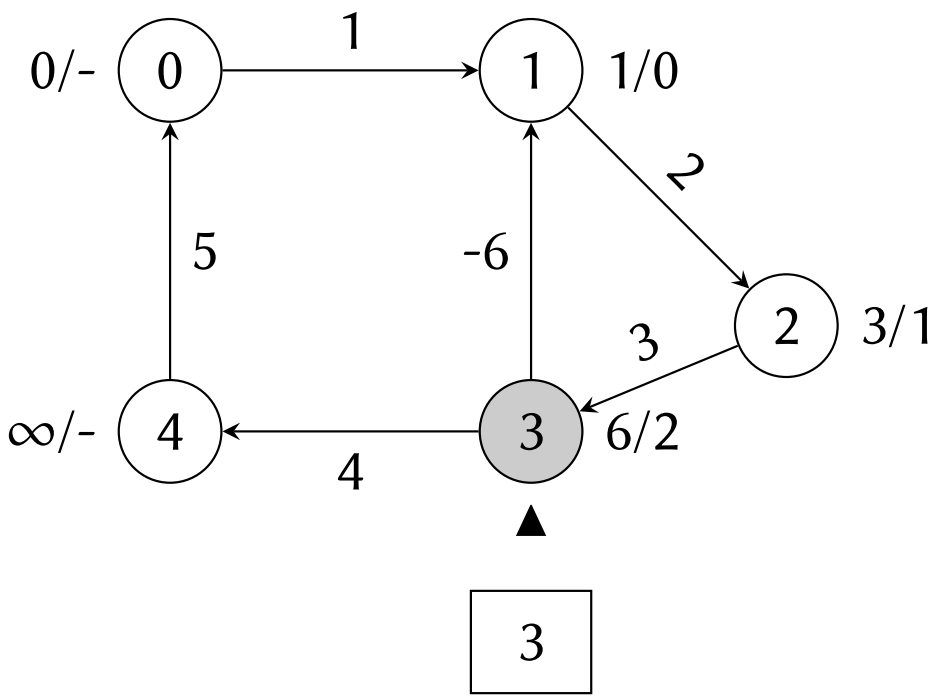
# Bellman-Ford with Negative Cycles Example (2)



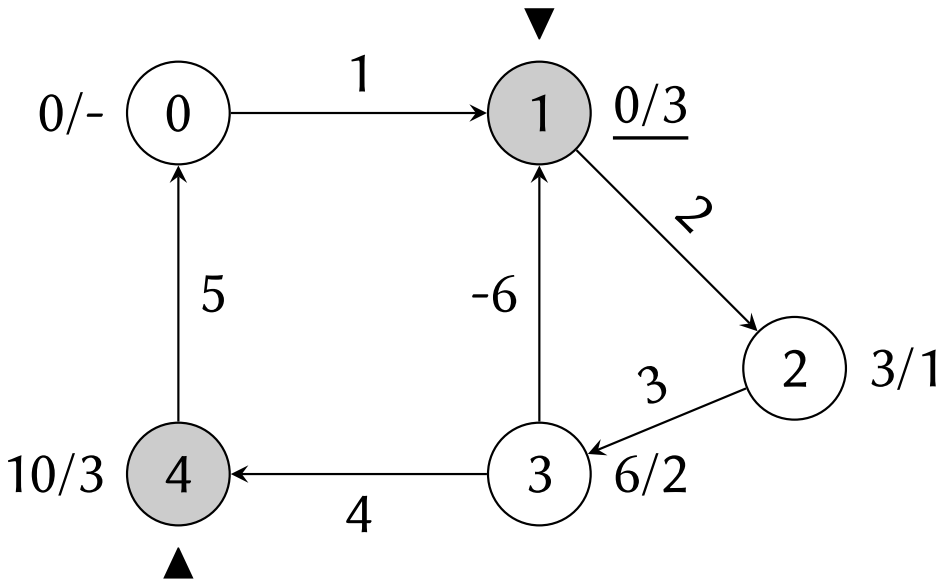
# Bellman-Ford with Negative Cycles Example (3)



# Bellman-Ford with Negative Cycles Example (4)

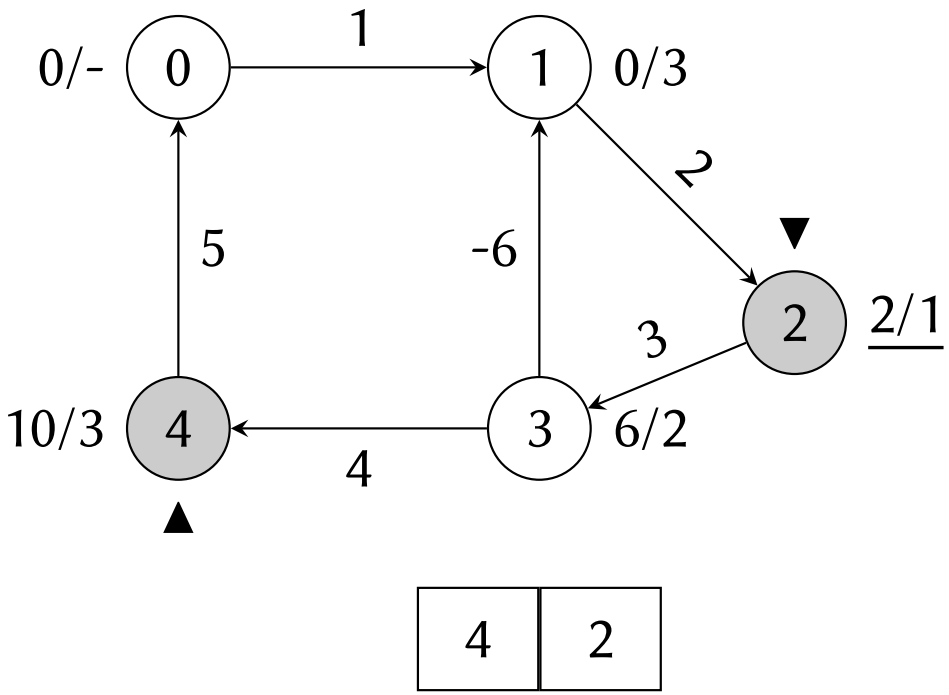


# Bellman-Ford with Negative Cycles Example (5)




| 1 | 4 |
|---|---|

# Bellman-Ford with Negative Cycles Example (6)



# Prevention

- The first Bellman-Ford implementation will stop anyway after $|V| - 1$ repetitions of the relaxation loop.
- To make sure that Bellman-Ford stops when implemented with a queue, we must add the proper check, otherwise it will never stop.

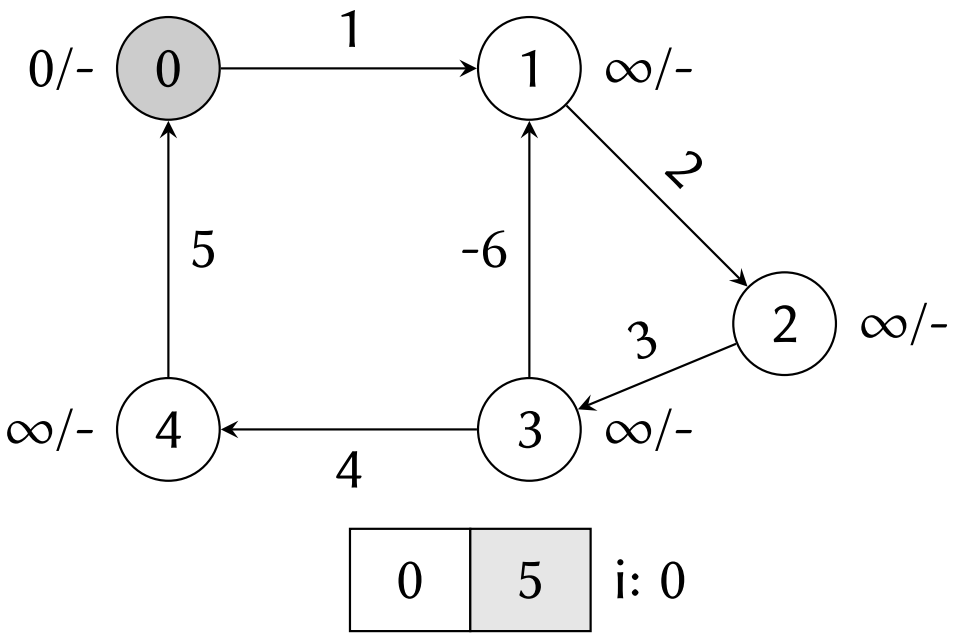
# Bellman-Ford with Negative Cycles

- We can detect a cycle if we find a path with more than $|V| - 1$ edges.
- In Bellman-Ford, we start with paths of zero edges from $s$.
- Then we find the paths including the neighbors of $s$.
- And then the paths including the neighbors of the neighbors of $s$, and so on.
- So we can stop after $|V| - 1$ iterations, in each of which we process one set of neighbors.

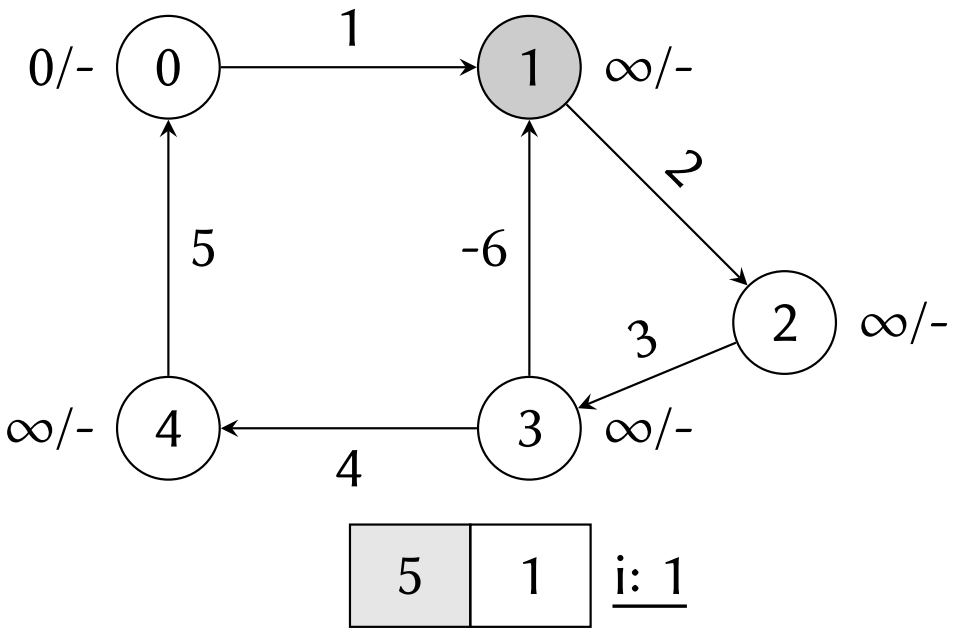
# Sentinel Value

- We need to know when we finish processing one set of neighbors.
- To do that we can use a special *sentinel* value in the queue.
- In general, sentinel values are invalid values that signal some specific event.
- In our case, we will use the number $|V|$ as the sentinel value.

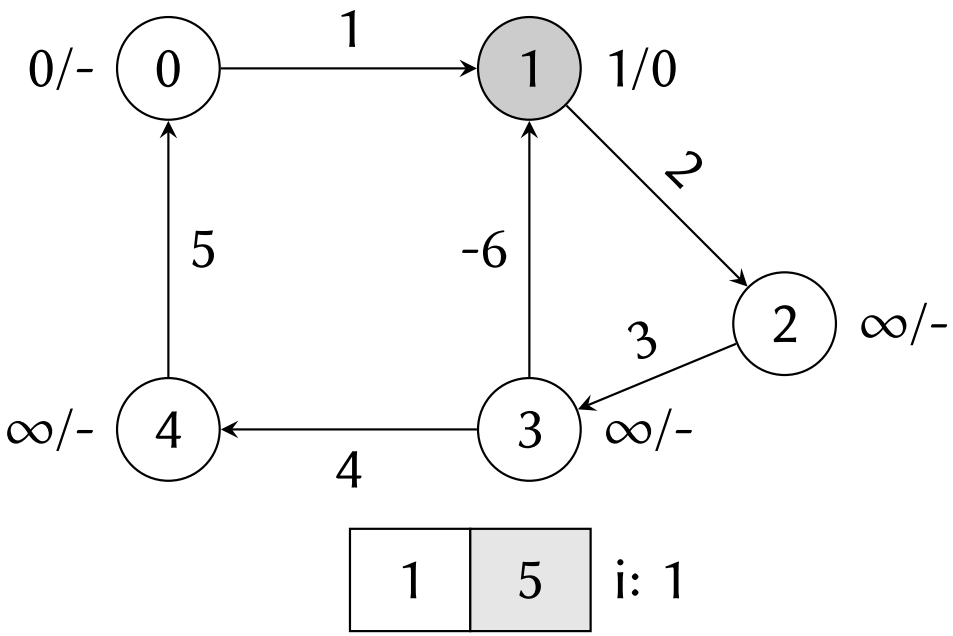
# Bellman-Ford with Sentinel Example (1)



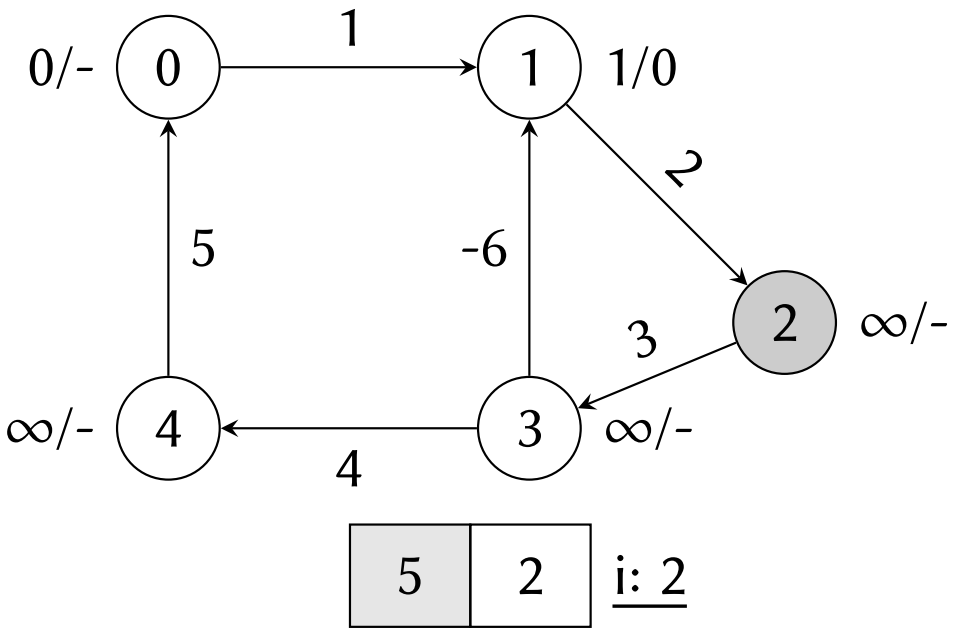
# Bellman-Ford with Sentinel Example (2)



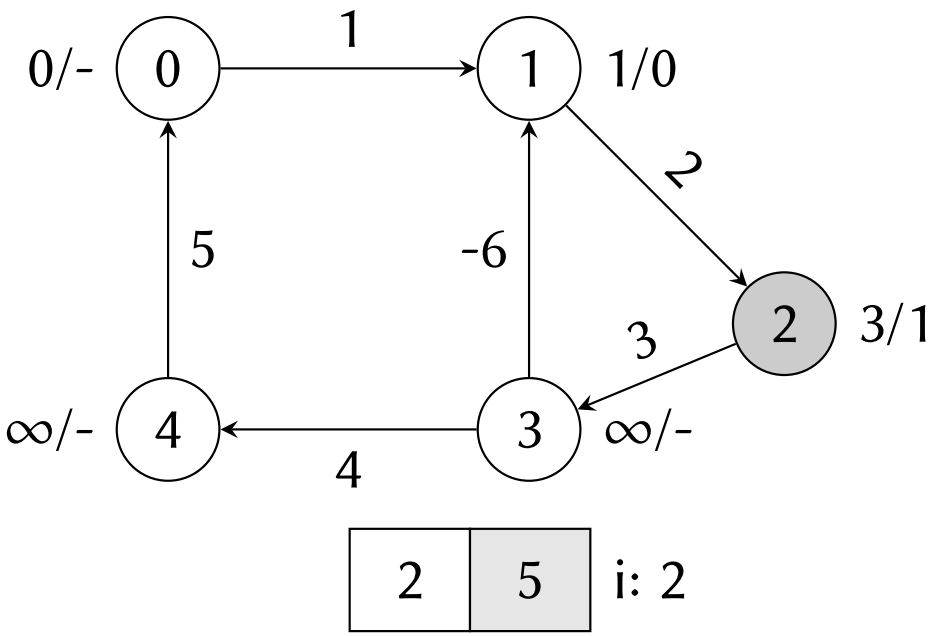
# Bellman-Ford with Sentinel Example (3)



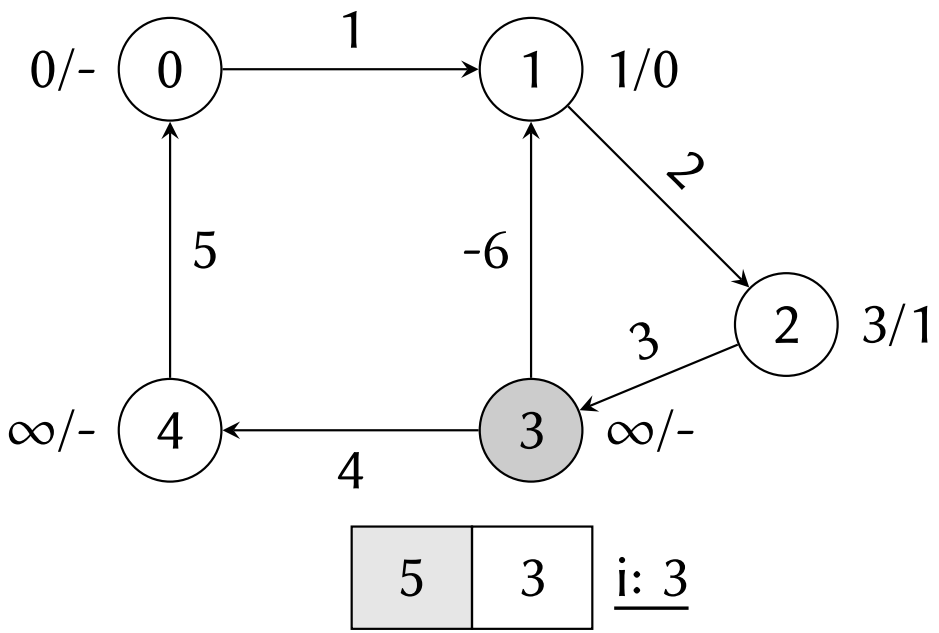
# Bellman-Ford with Sentinel Example (4)



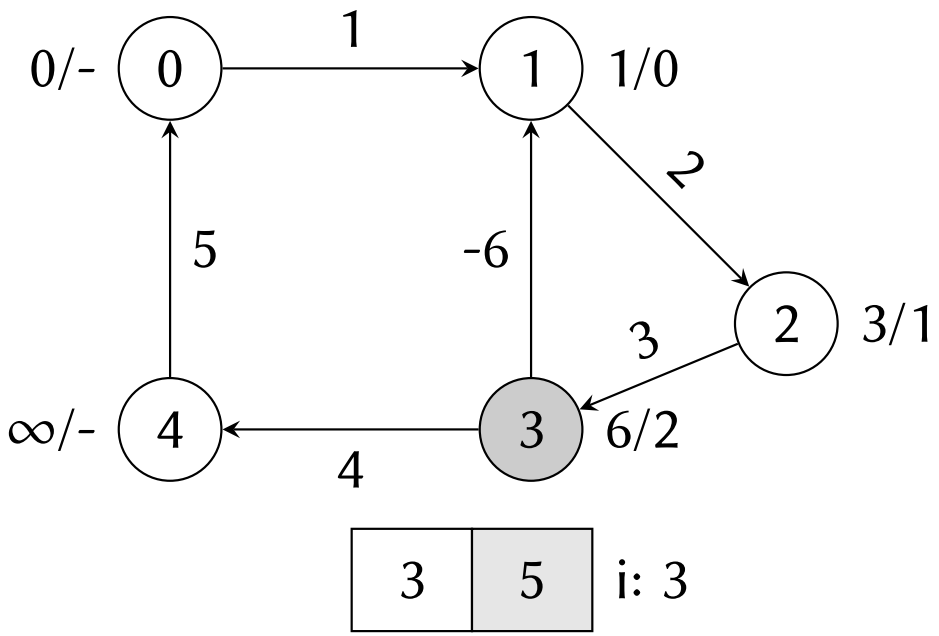
# Bellman-Ford with Sentinel Example (5)



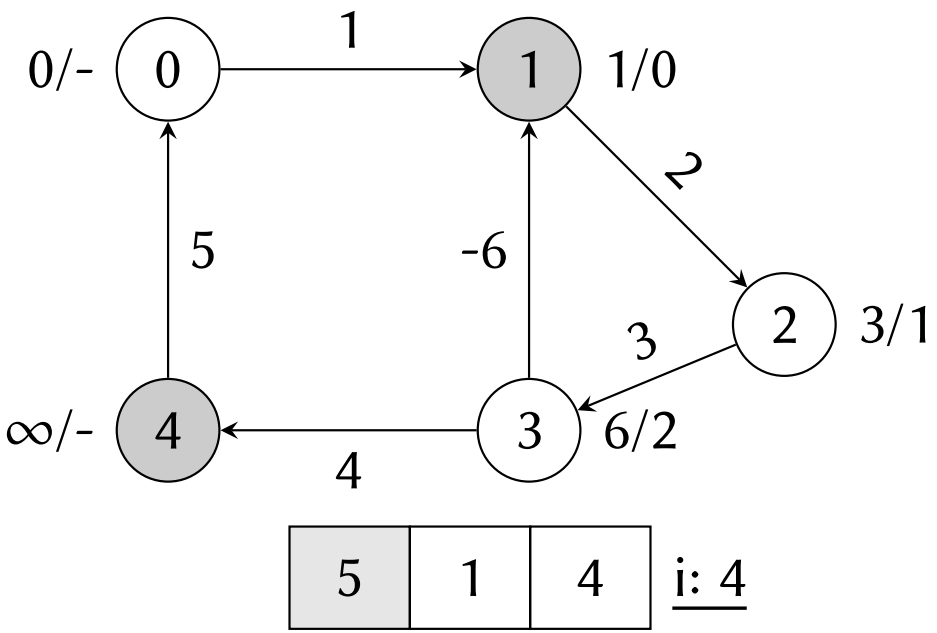
# Bellman-Ford with Sentinel Example (6)



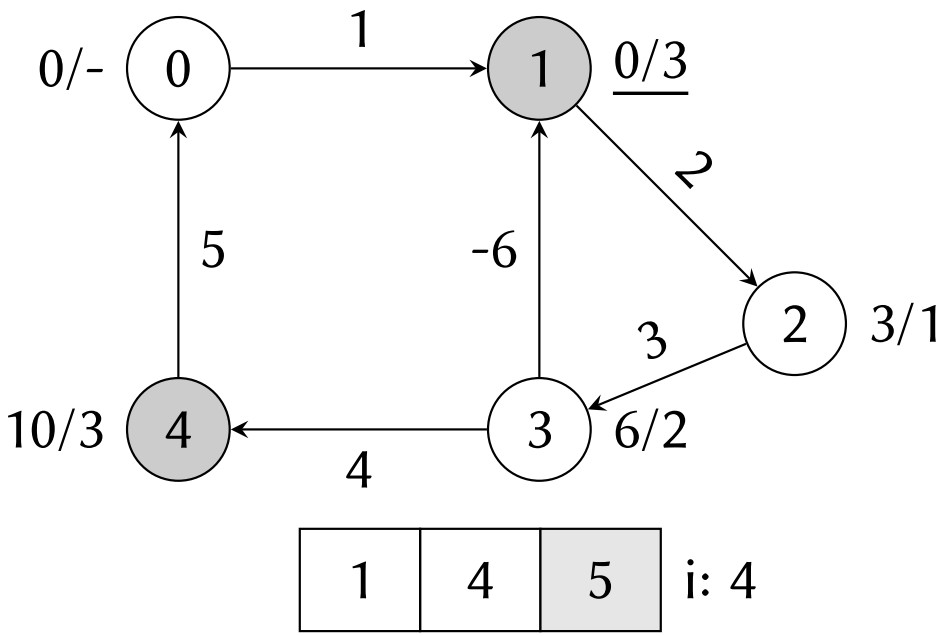
# Bellman-Ford with Sentinel Example (7)



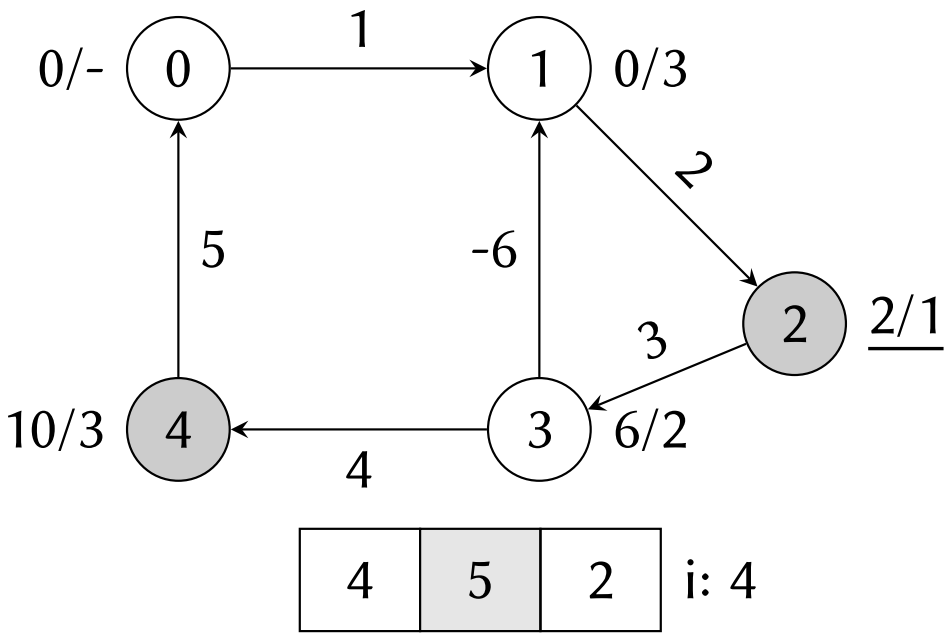
# Bellman-Ford with Sentinel Example (8)



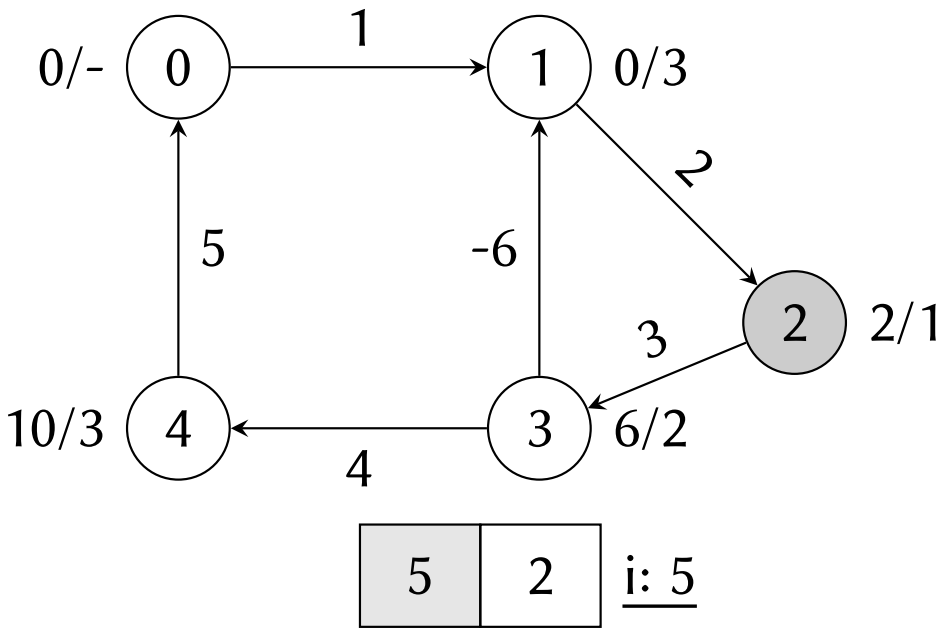
# Bellman-Ford with Sentinel Example (9)



# Bellman-Ford with Sentinel Example (10)



# Bellman-Ford with Sentinel Example (11)



# Queue-based Bellman-Ford with Negative Cycles Algorithm

**Algorithm:** Queue-based Bellman-Ford with negative cycles.

`BellmanFordQueueNC`$(G, s) \rightarrow (pred, dist, ncc)$

**Input:** $G = (V, E)$, a graph
$s$, the starting node

**Output:** $pred$, an array of size $|V|$ such that $pred[i]$ is the predecessor of node $i$ in the shortest path from $s$
$dist$, an array of size $|V|$ such that $dist[i]$ is the length of the shortest path calculated from node $s$ to $i$
$nnc$, TRUE if there is no negative cycle, FALSE otherwise.

```
1  inqueue ← CreateArray(|V|)
2  Q ← CreateQueue()
3  foreach v in V do
4      pred[v] ← |V|
5      if v ≠ s then
6          dist[v] ← ∞
7          inqueue[v] ← FALSE
8      else
9          dist[v] ← 0
10 Enqueue(Q, s)
11 inqueue[s] ← TRUE
12 Enqueue(Q, |V|)
13 i ← 0
14 while Size(Q) ≠ 1 and i < |V| do
15     u ← Dequeue(Q)
16     if u = |V| then
17         i ← i + 1
18         Enqueue(Q, |V|)
19     else
20         inqueue[u] ← FALSE
21         foreach v in AdjacencyList(G, u) do
22             if dist[v] > dist[u] + Weight(G, u, v) then
23                 dist[v] ← dist[u] + Weight(G, u, v)
24                 pred[v] ← u
25                 if not inqueue[v] then
26                     Enqueue(Q, v)
27                     inqueue[v] ← TRUE
28 return (pred, dist, i < |V|)
```

# Outline

# Top Ten Most Traded Currencies by Value, April 2013

| Rank | Currency | Code | % daily share |
|---|---|---|---|
| 1 | United States dollar | USD | 87.0% |
| 2 | European Union euro | EUR | 33.4% |
| 3 | Japanese yen | JPY | 23.0% |
| 4 | United Kingdom pound sterling | GPB | 11.8% |
| 5 | Australian dollar | AUD | 8.6% |
| 6 | Swiss franc | CHF | 5.2% |
| 7 | Canadian dollar | CAD | 4.6% |
| 8 | Mexican peso | MXN | 2.5% |
| 9 | Chinese yuan | CNY | 2.2% |
| 10 | New Zealand dollar | NZD | 2.0% |

The total percentage (including currencies below the top ten) adds up to 200% as trades are between pairs of currencies.

# Arbitrage Example

- Suppose that in London the exchange rate of euros and U.S. dollars is:

$$€1 = \$1.37$$

- Suppose also that in New York City the exchange rate of U.S. dollars and euros is:

$$\$1 = €0.74$$

- Then a trader can buy in London using €1,000,000 a total of $1,370,000 and send them to New York City.
- There the trader converts them back to euros:

$$\$1{,}370{,}000 \times 0.74 = €1{,}013{,}800$$

- The trader made a risk-free profit of €13,800.

# How is Arbitrage Resolved

- If the exchange rate between U.S. dollars and euros in New York city is:

$$\$\,1 = €\,0.73$$

- Then with the same trades, the trader will have:

$$\$\,1{,}370{,}000 \times 0.73 = €\,1{,}000{,}100$$

- So the profit will be just €100.
- As soon as arbirtage is detected, traders will rush to take advantage of it and the exchange rates will move towards equilibrium.

# General Scheme

- From U.S. dollars to Australian dollars, to Canadian dollars, and then back to U.S. Dollars:

$$1 \times (\text{USD} \rightarrow \text{AUD}) \times (\text{AUD} \rightarrow \text{CAD}) \times (\text{CAD} \rightarrow \text{USD})$$

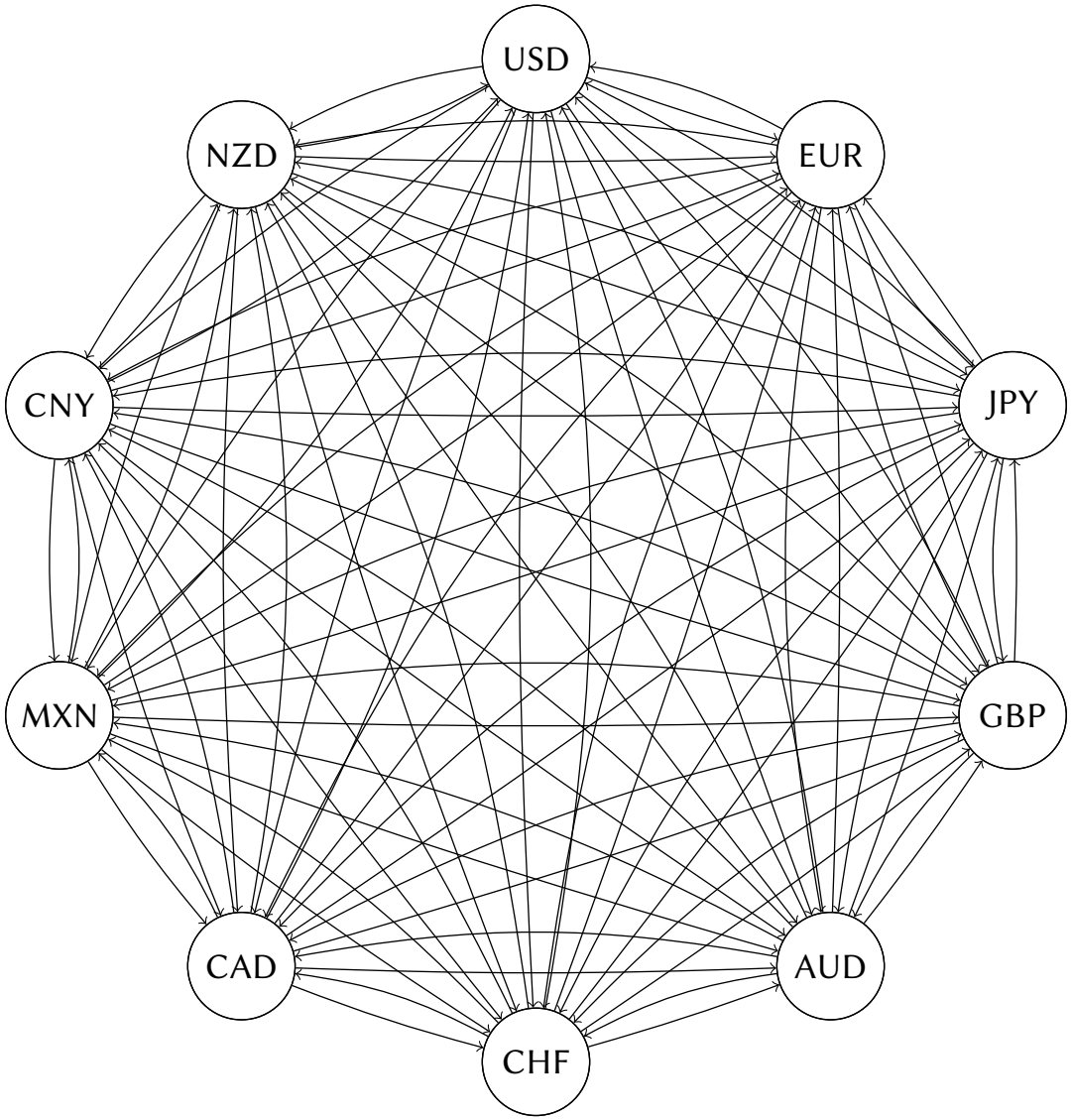
- In general, we want to find a sequence of currency conversions:

$$a = 1 \times (c_1 \rightarrow c_2) \times (c_2 \rightarrow c_3) \times \cdots \times (c_n \rightarrow c_1)$$

where:

$$a > 1$$

# Cross-currency Rates Graph



# Cross-currency Graph Adjacency Matrix

| | USD | EUR | JPY | GBP | AUD | CHF | CAD | MXN | CNY | NZD |
|---|---|---|---|---|---|---|---|---|---|---|
| USD | 1 | 1.3744 | 0.009766 | 1.6625 | 0.9262 | 1.1275 | 0.9066 | 0.07652 | 0.1623 | 0.8676 |
| EUR | 0.7276 | 1 | 0.007106 | 1.2097 | 0.6739 | 0.8204 | 0.6596 | 0.05568 | 0.1181 | 0.6313 |
| JPY | 102.405 | 140.743 | 1 | 170.248 | 94.8421 | 115.455 | 92.8369 | 7.836 | 16.6178 | 88.8463 |
| GBP | 0.6016 | 0.8268 | 0.005875 | 1 | 0.5572 | 0.6782 | 0.5454 | 0.04603 | 0.09762 | 0.5219 |
| AUD | 1.0799 | 1.4842 | 0.010546 | 1.7953 | 1 | 1.2176 | 0.979 | 0.08263 | 0.1752 | 0.9369 |
| CHF | 0.8871 | 1.2192 | 0.008663 | 1.4748 | 0.8216 | 1 | 0.8042 | 0.06788 | 0.144 | 0.7696 |
| CAD | 1.1033 | 1.5163 | 0.010775 | 1.8342 | 1.0218 | 1.2439 | 1 | 0.08442 | 0.179 | 0.9572 |
| MXN | 13.0763 | 17.9724 | 0.1277 | 21.7397 | 12.1111 | 14.7435 | 11.8545 | 1 | 2.122 | 11.345 |
| CNY | 6.167 | 8.4761 | 0.06023 | 10.2528 | 5.7118 | 6.9533 | 5.5908 | 0.4719 | 1 | 5.3505 |
| NZD | 1.153 | 1.5846 | 0.01126 | 1.9168 | 1.0678 | 1.2999 | 1.0452 | 0.08822 | 0.1871 | 1 |

# Arbitrage Detection

## Problem

*To spot an arbitrage opportunity we must find in the cross-currency rates graph a cycle with weights:*

$$w_1, w_2, \ldots, w_n$$

*such that:*

$$w_1 w_2 \ldots w_n > 1$$

# Arbitrage Detection Method

1. If the weight between two vertices $u$ and $v$ is $w(u, v)$, we substitute it by the value $w'(u, v) = -\log w(u, v)$.
2. We execute the amended Bellman-Ford to detect a negative cycle in the resulting graph.
3. If the algorithm reports the existence of a negative cycle, the cycle corresponds to an arbitrage opportunity.

# Explanation (1)

- Suppose that the cycle consists of $n$ edges, then the sum of the cycle weights will be:

$$w'_1 + w'_2 + \ldots + w'_n < 0$$

where:

$$w'_1, w'_2, \ldots, w'_n$$

are the weights.

- We have:

$$w'_1 + w'_2 + \ldots + w'_n = -\log w_1 - \log w_2 - \cdots - \log w_n$$

which means that in the negative cycle:

$$-\log w_1 - \log w_2 - \cdots - \log w_n < 0$$

- A basic property of logarithms is that $\log(xy)$ is equal to $\log x + \log y$ and $\log(1/x\ 1/y)$ equal to $\log(1/x) + \log(1/y) = -\log x - \log y$.

# Explanation (2)

- So the last inequality becomes:

$$\log(\frac{1}{w_1}\frac{1}{w_2}\cdots\frac{1}{w_n}) < 0$$

- Exponentiating to remove the logarithm we get:

$$\frac{1}{w_1}\frac{1}{w_2}\cdots\frac{1}{w_n} < 10^0 = 1$$

- But this is equivalent to:

$$w_1 w_2 \ldots w_n > 1$$

That is what we wanted to find in the first place; a cyclical path with weights whose product is greater than one.