

A collection of vintage items is arranged on a light-colored, textured surface. In the top left, a portion of a wooden chessboard with a checkered pattern and several chess pieces is visible. Below the chessboard, there are two medals: one with a red ribbon and a circular emblem, and another with a blue ribbon and a circular emblem. A pair of round, gold-rimmed glasses with thin temples is positioned in the center. In the bottom left corner, a small, round, silver-colored compass with a white face and black markings is visible. The text "Graph Shortest Path" is written in a large, serif font on the right side of the image.

Graph Shortest Path

Notes by Yan Yan



Contents

- ◆ Dijkstra's Algorithm
- ◆ Bellman-Ford's Algorithm

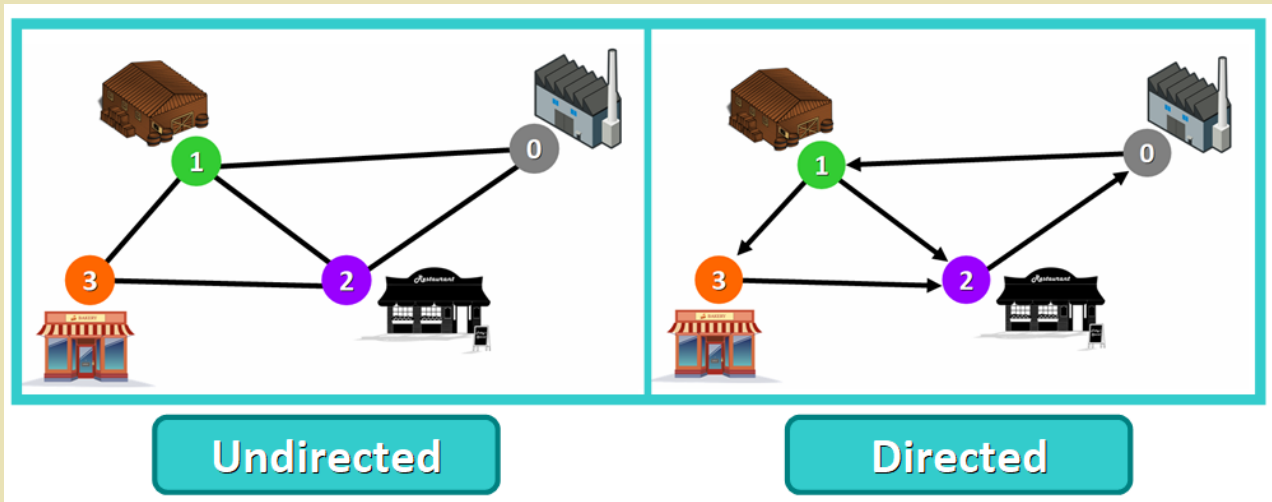


Learning Objectives

- ◆ Understand the process of Dijkstra's Algorithm and Bellman-Ford's Algorithm
- ◆ Implement the two algorithms to find the shortest path on a given graph
- ◆ Describe the differences of the two algorithms
- ◆ Remember the time complexities of the two algorithms

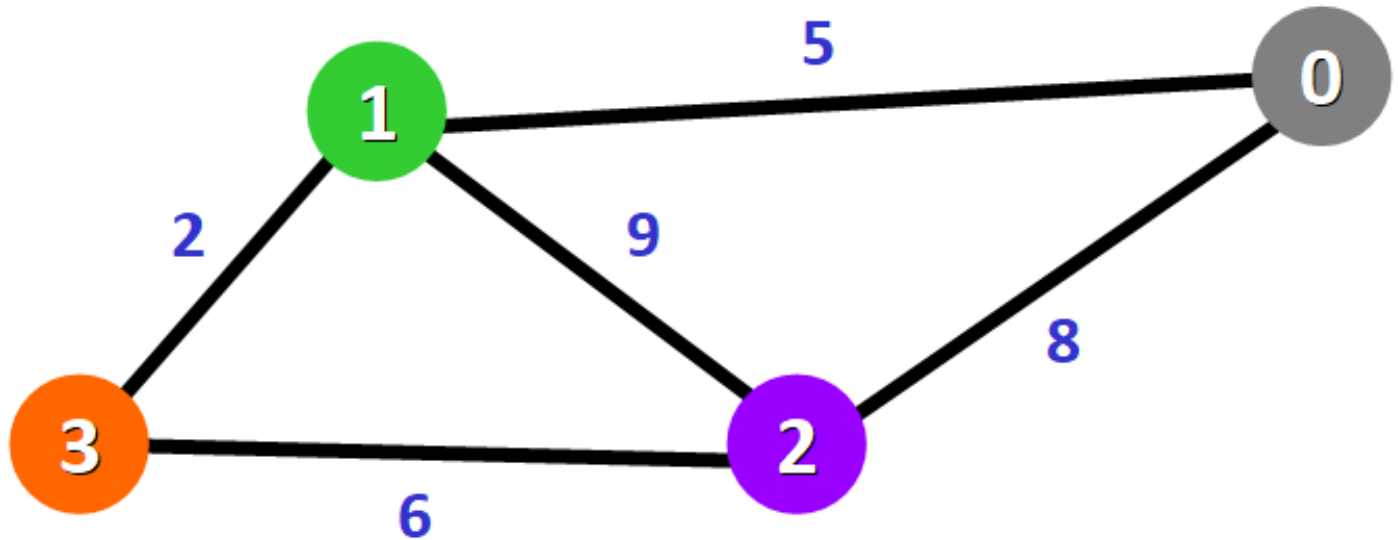
Dijkstra's algorithm

- ◆ Dijkstra's shortest path algorithm, created by Edsger Dijkstra
 - Dutch computer scientist and software engineer
- ◆ Can be directed or undirected graph with positive weights



Dijkstra's algorithm


- ◆ Example on a weighted graph
- ◆ What is the shortest path between 1 and 2?





Dijkstra's algorithm

- ◆ Dijkstra's shortest path algorithm determines the shortest path from a start vertex (source) to each vertex in a graph.
 - The algorithm keeps track of the currently known shortest distance from each vertex to the source vertex and it updates these values if it finds a shorter path.
 - Once the algorithm has found the shortest path between the source vertex and another vertex, that vertex is marked as "visited" and added to the path.
 - The process continues until all the vertices in the graph have been added to the path.

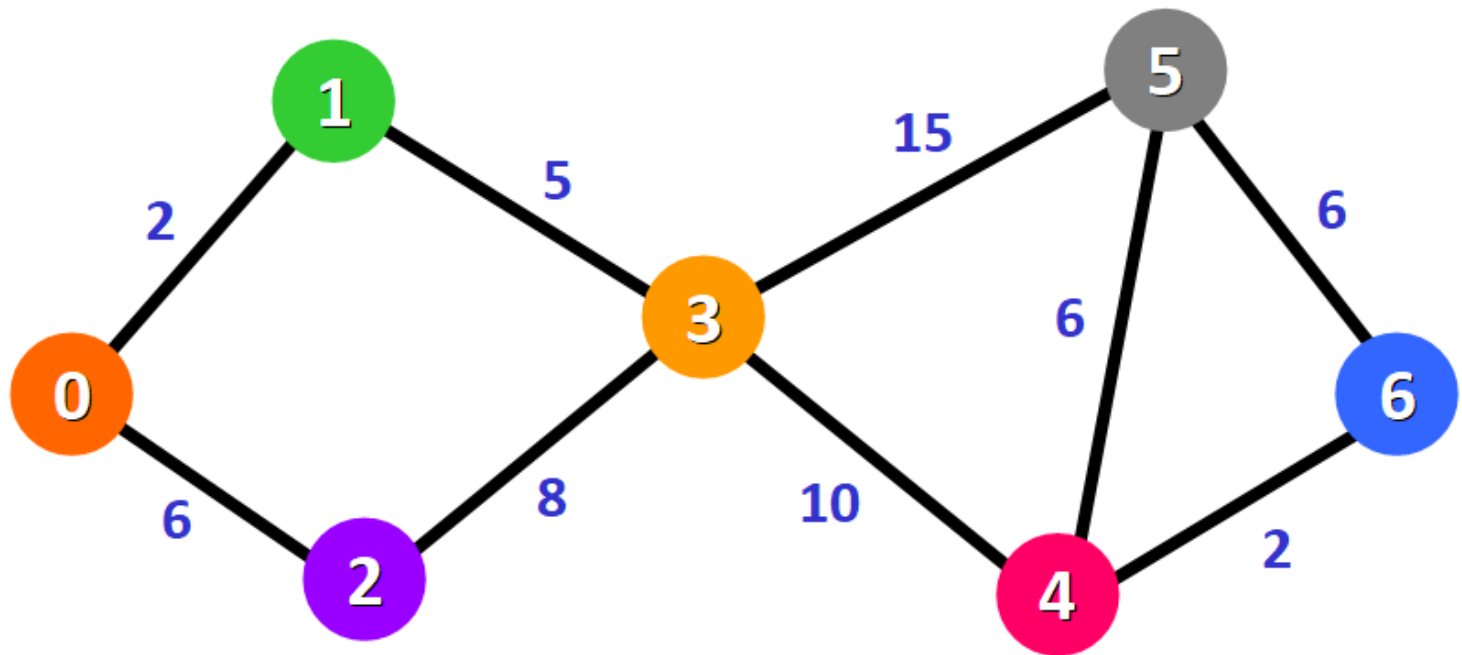


Dijkstra's algorithm

- ◆ For each vertex, Dijkstra's algorithm determines the vertex's distance and predecessor pointer.
- ◆ A vertex's **distance** is the shortest path distance from the start vertex.
- ◆ A vertex's **predecessor** pointer points to the previous vertex along the shortest path from the start vertex.

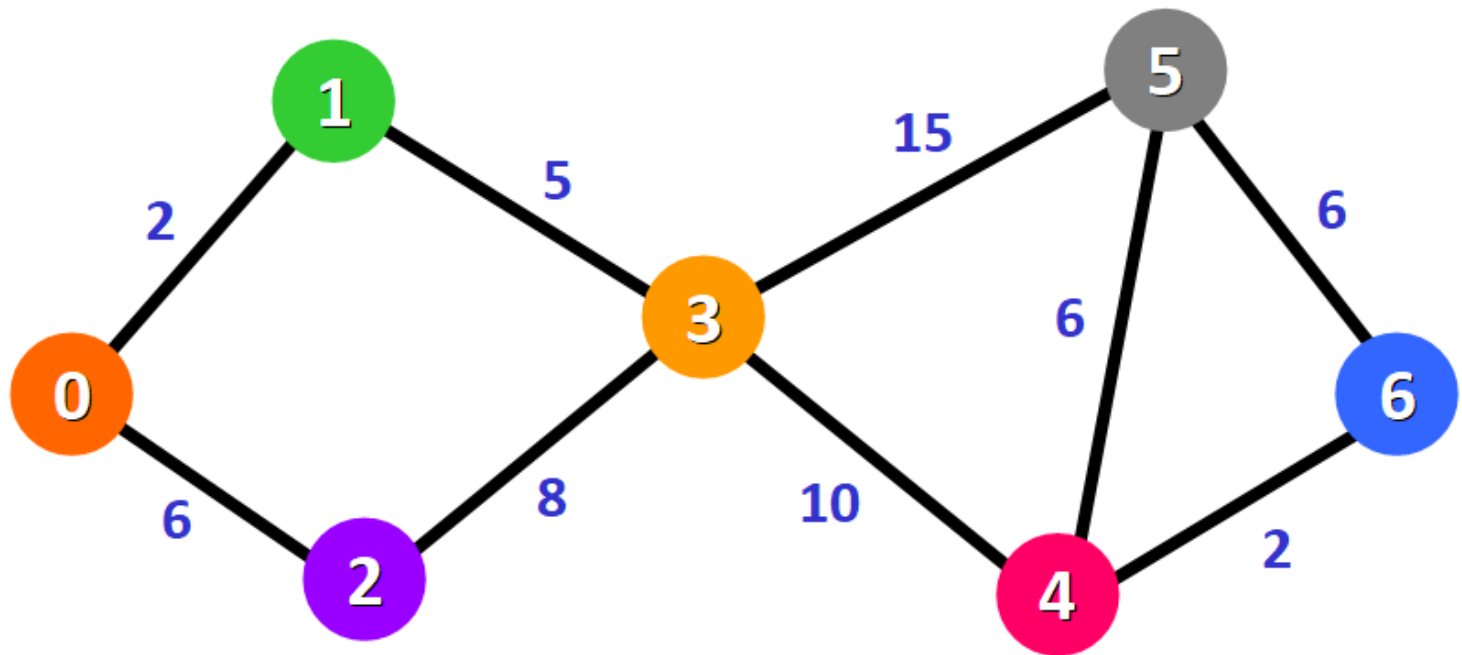
Dijkstra's algorithm

- ◆ Example procedure



Dijkstra's algorithm

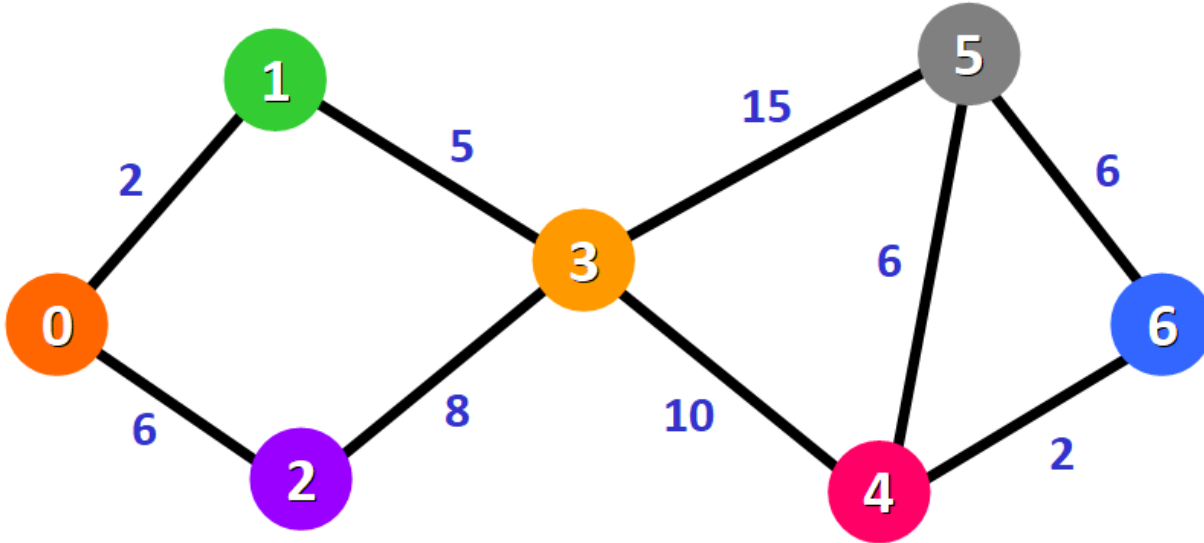
- ◆ Example procedure sources 0



- We will find the shortest path from node 0 to node 1, from node 0 to node 2, from node 0 to node 3, and so on for every node in the graph.

Dijkstra's algorithm

Step 1: The distance from the source node to itself is 0.



Unvisited Nodes: {0, 1, 2, 3, 4, 5, 6}

Distance:

0: 0

1: ∞

2: ∞

3: ∞

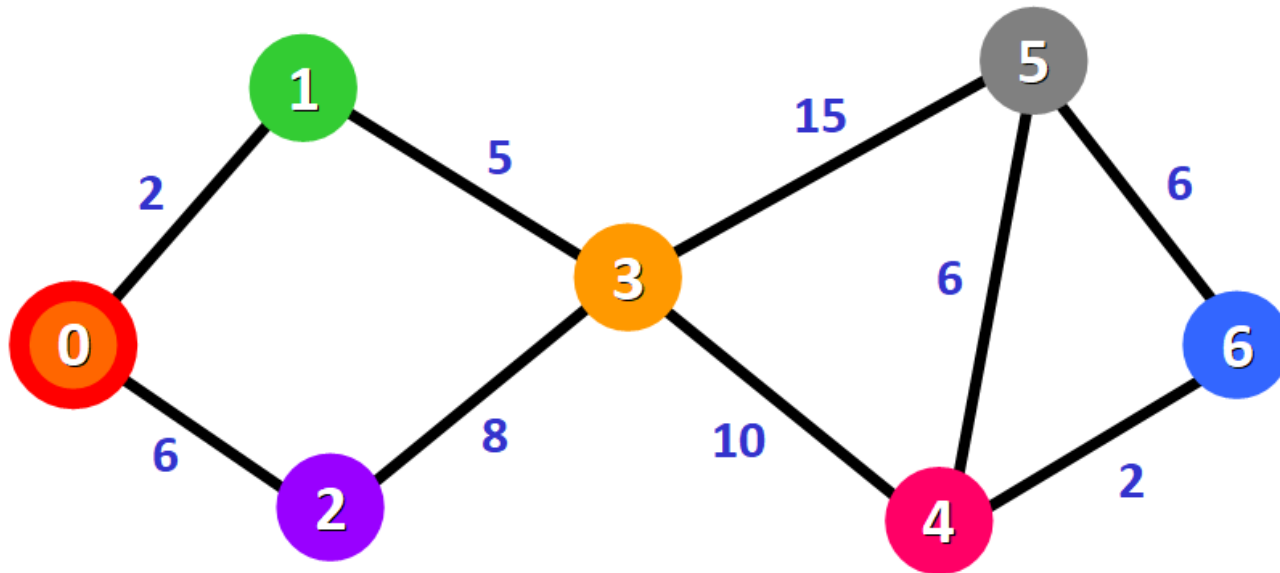
4: ∞

5: ∞

6: ∞

Dijkstra's algorithm

Step 1: Visit node 0. Add to path. Current: {0}.



Distance:

0: 0

1: ∞

2: ∞

3: ∞

4: ∞

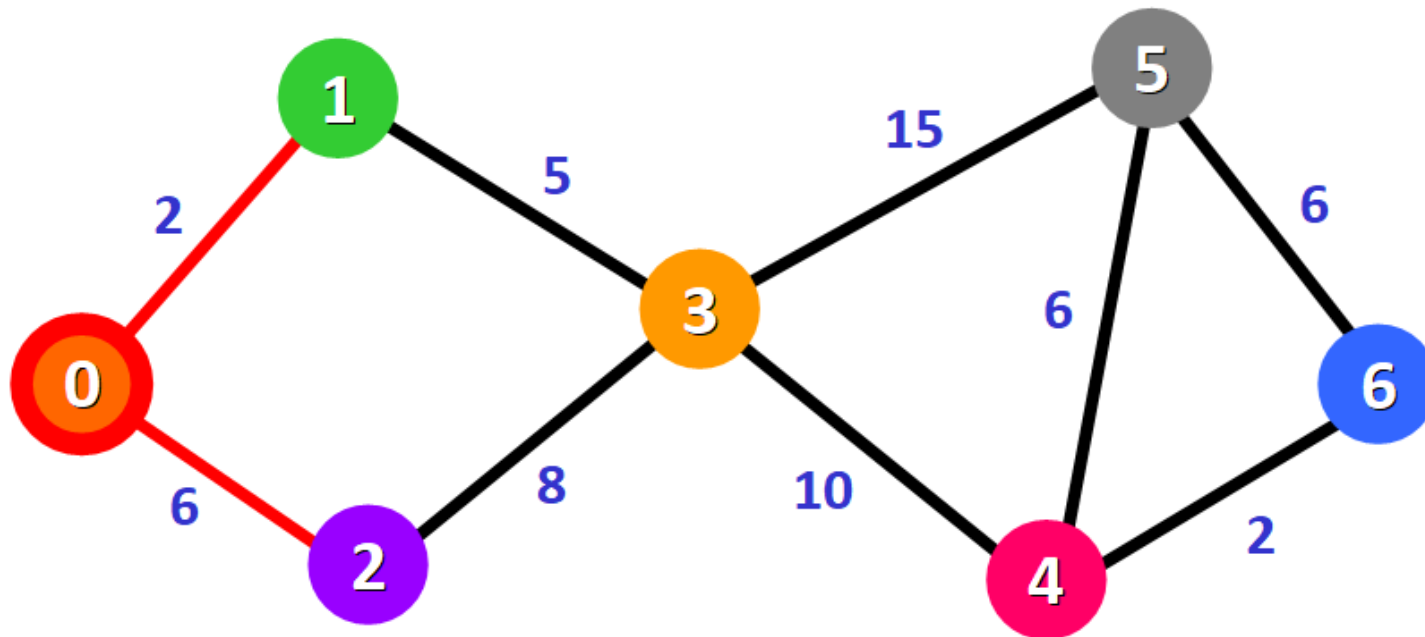
5: ∞

6: ∞

Unvisited Nodes: ~~0~~, 1, 2, 3, 4, 5, 6

Dijkstra's algorithm

Step 2: Start checking the distance from node 0 to its adjacent nodes (1 and 2), and update the distances from node 0 to them.



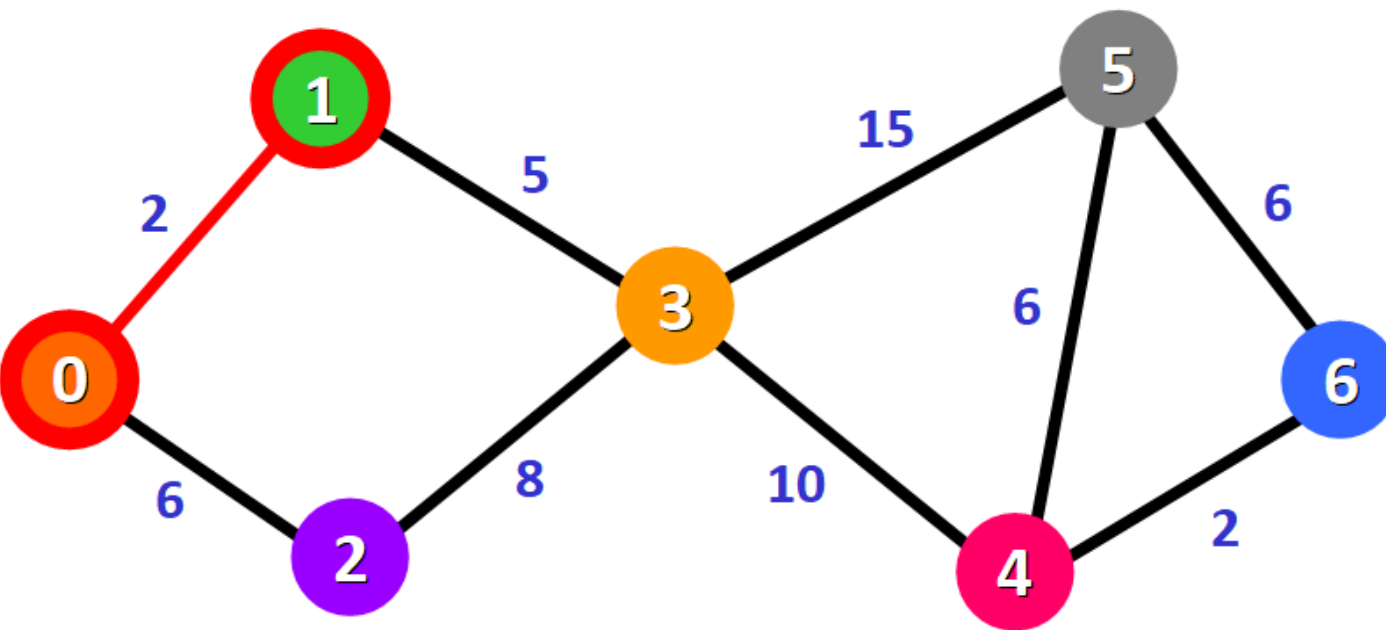
Distance:

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ∞
4: ∞
5: ∞
6: ∞

Unvisited Nodes: ~~0~~, 1, 2, 3, 4, 5, 6

Dijkstra's algorithm

Step 2: Select the node that is closest to the source node based on the current known distances. Mark it as visited. Add it to the path. Current: {0, 1}.



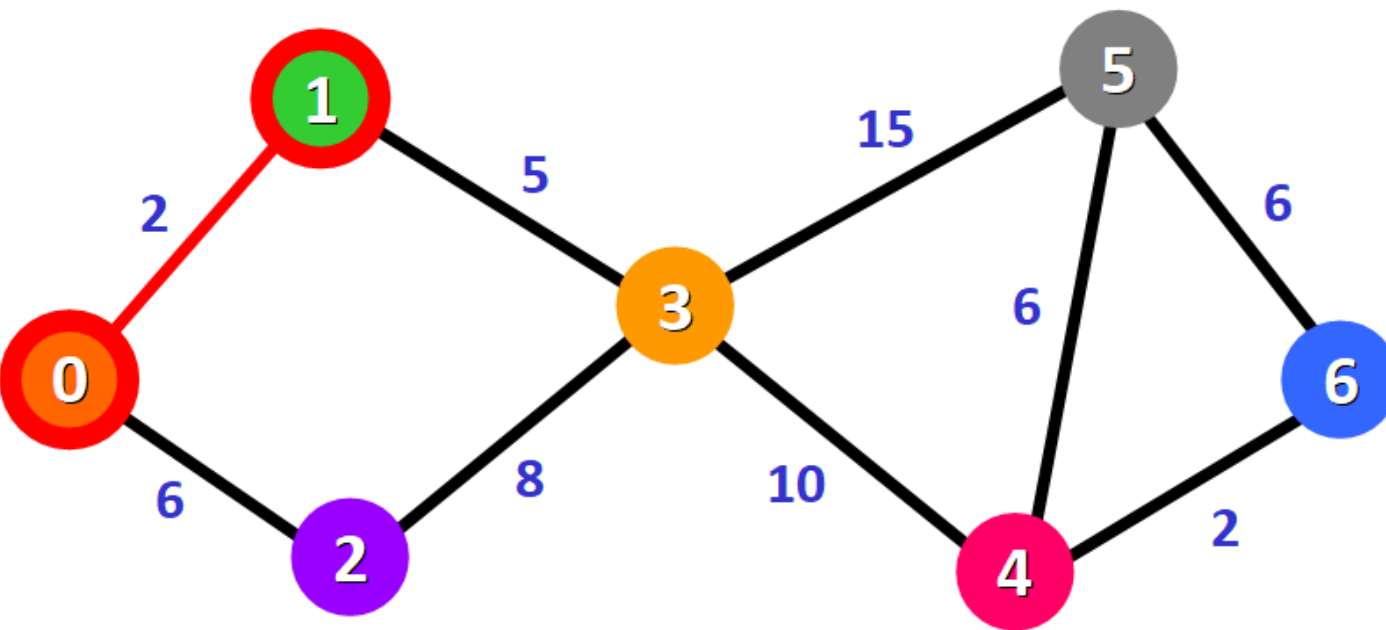
Distance:

0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ∞
4: ∞
5: ∞
6: ∞

Unvisited Nodes: ~~0~~, ~~1~~, 2, 3, 4, 5, 6

Dijkstra's algorithm

Step 3: Analyze nodes adjacent to nodes in the path. Here they are 2, 3.
Only need to update distance for 3.



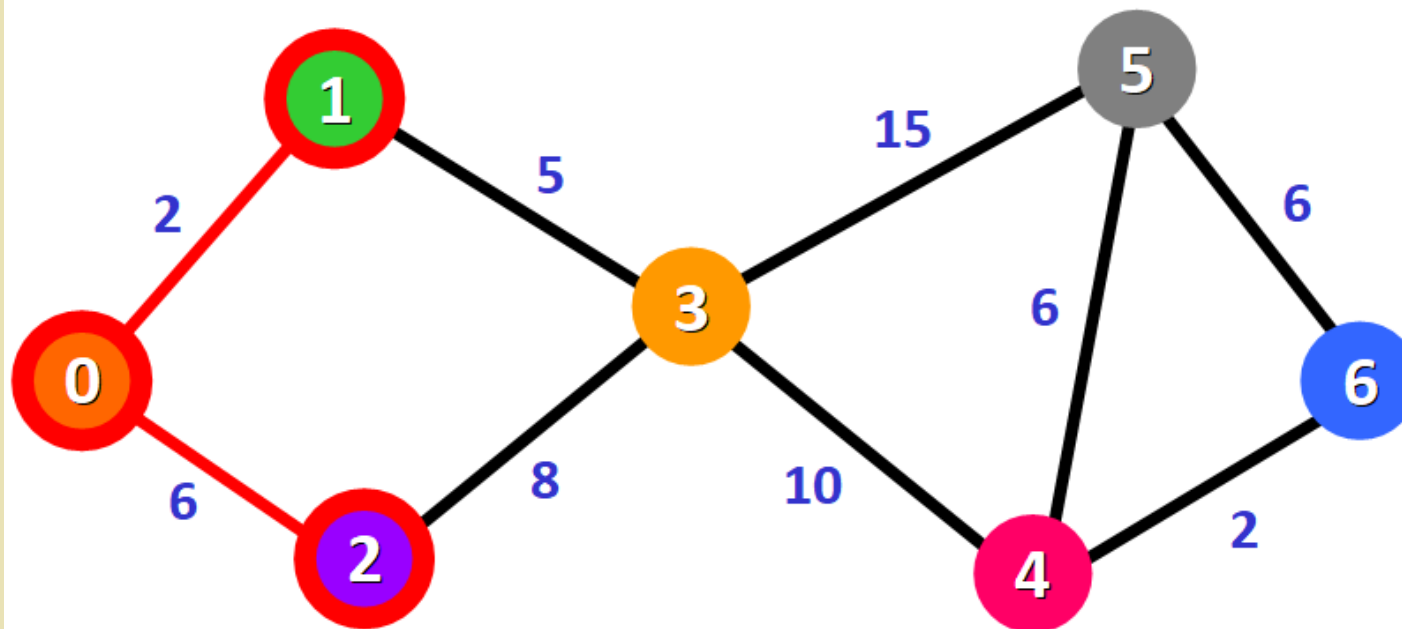
Distance:

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

Unvisited Nodes: ~~0~~, ~~1~~, 2, 3, 4, 5, 6

Dijkstra's algorithm

Step 3: select the **unvisited** node with the (currently known) shortest distance to the source node. Add to path. Current: {0, 1, 2}.



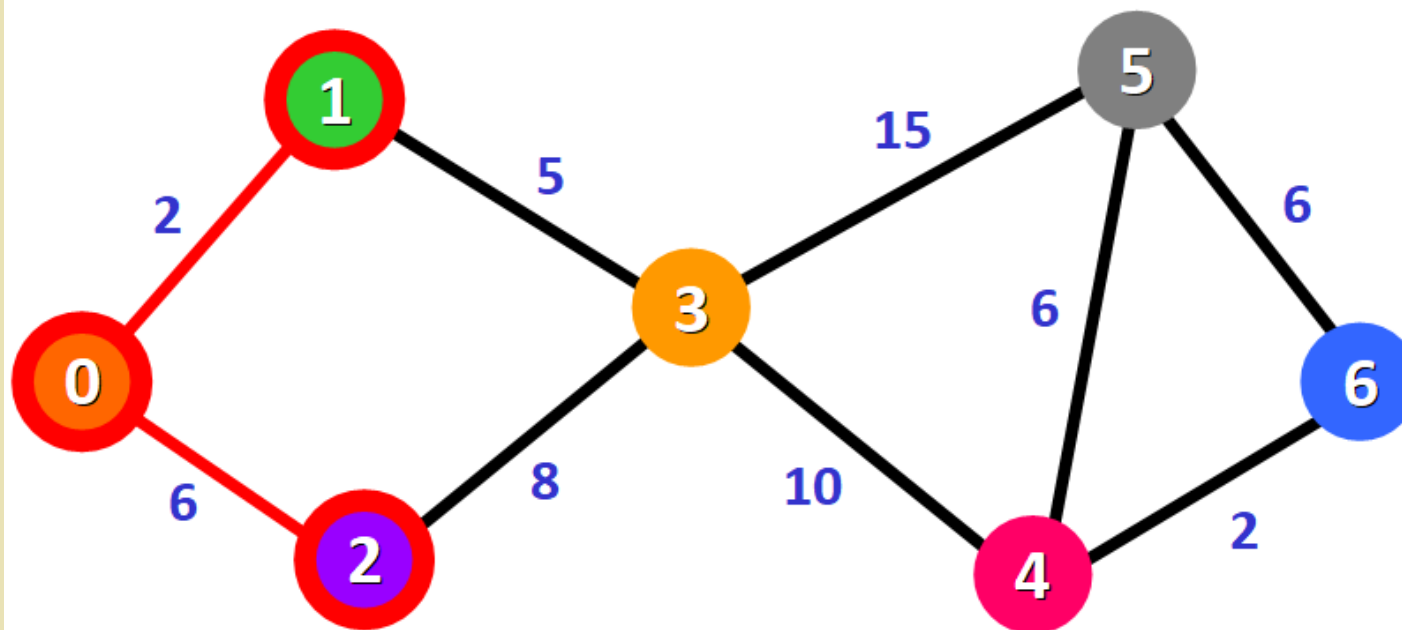
Distance:

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, ~~2~~, 3, 4, 5, 6}

Dijkstra's algorithm

Step 4: Repeat previous step. Analyze nodes adjacent to nodes in the path. Only 3. Check if updates needed. Here, $2+5=7 < (6+8)$. No updates.



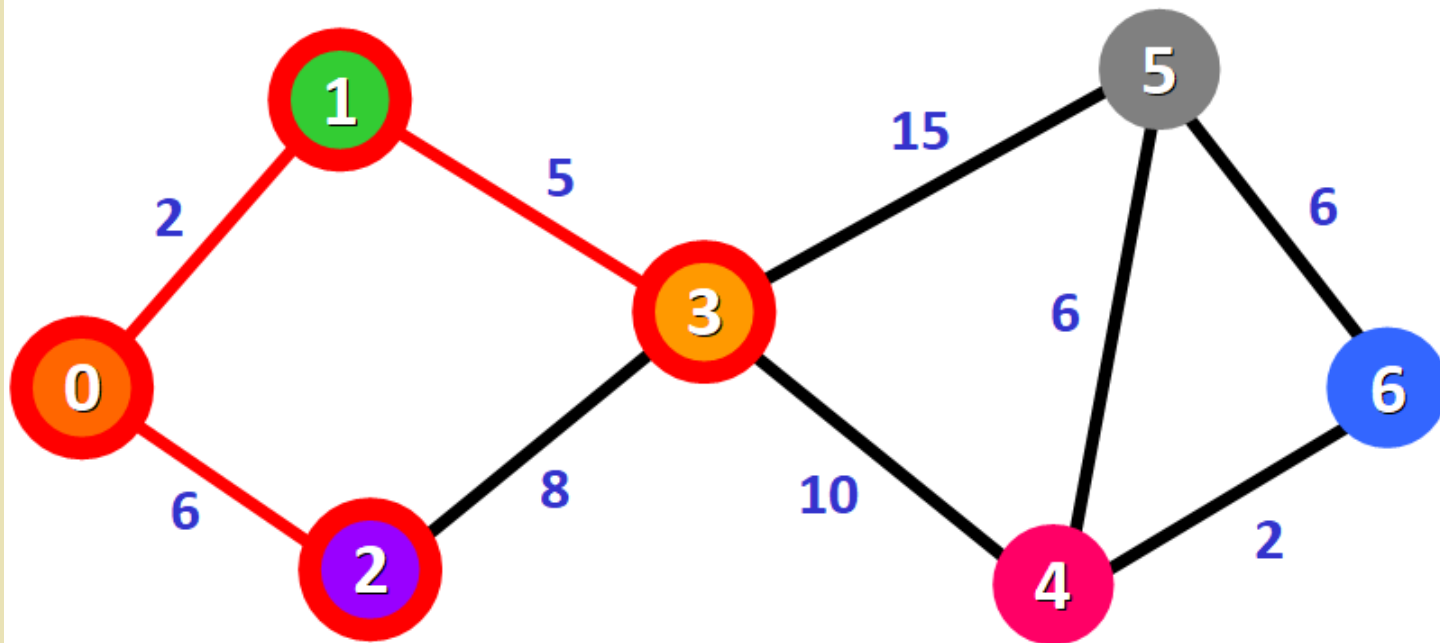
Distance:

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7
4: ∞
5: ∞
6: ∞

Unvisited Nodes: ~~0~~, ~~1~~, ~~2~~, 3, 4, 5, 6

Dijkstra's algorithm

Step 4: Add **unvisited** nodes with the shortest known distance to the source. Add to path. Current path: {0,1,2,3}.



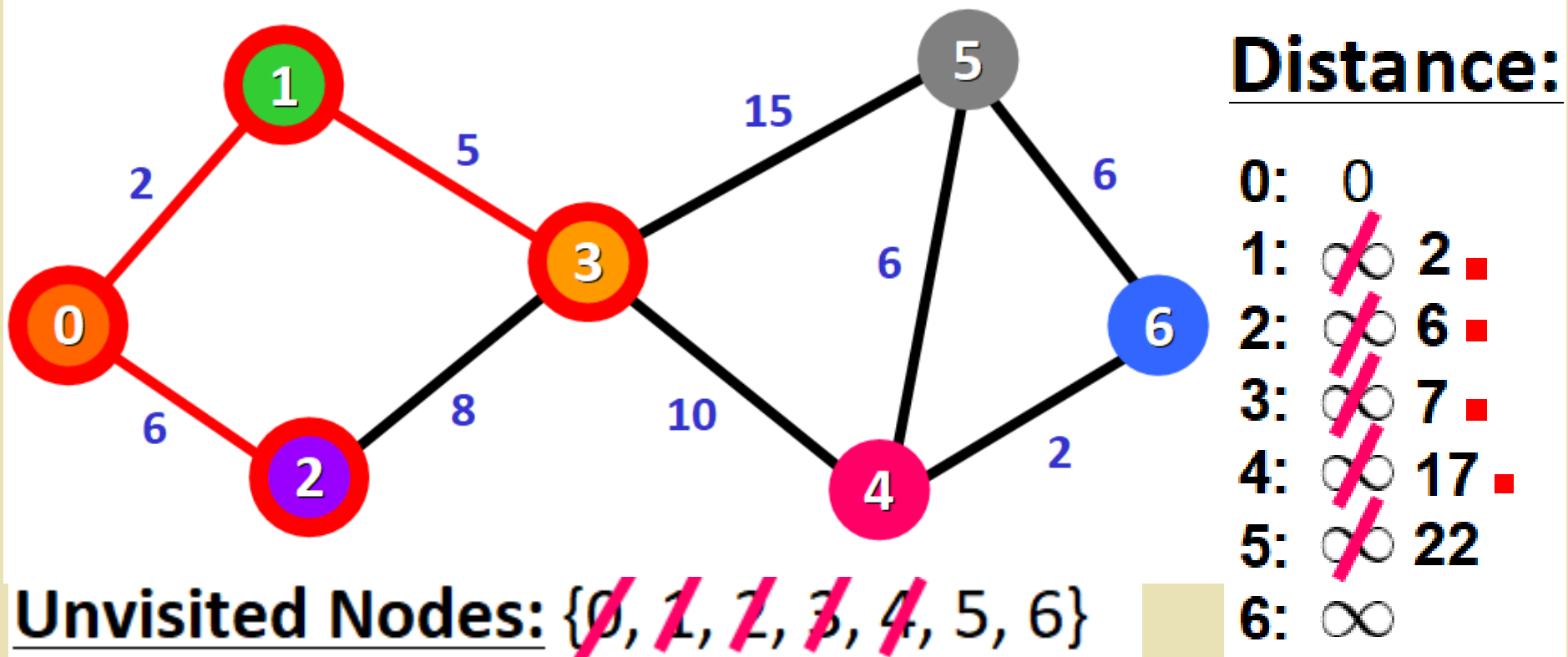
Distance:

0: 0
1: ~~∞~~ 2 ■
2: ~~∞~~ 6 ■
3: ~~∞~~ 7 ■
4: ∞
5: ∞
6: ∞

Unvisited Nodes: {~~0~~, ~~1~~, ~~2~~, ~~3~~, 4, 5, 6}

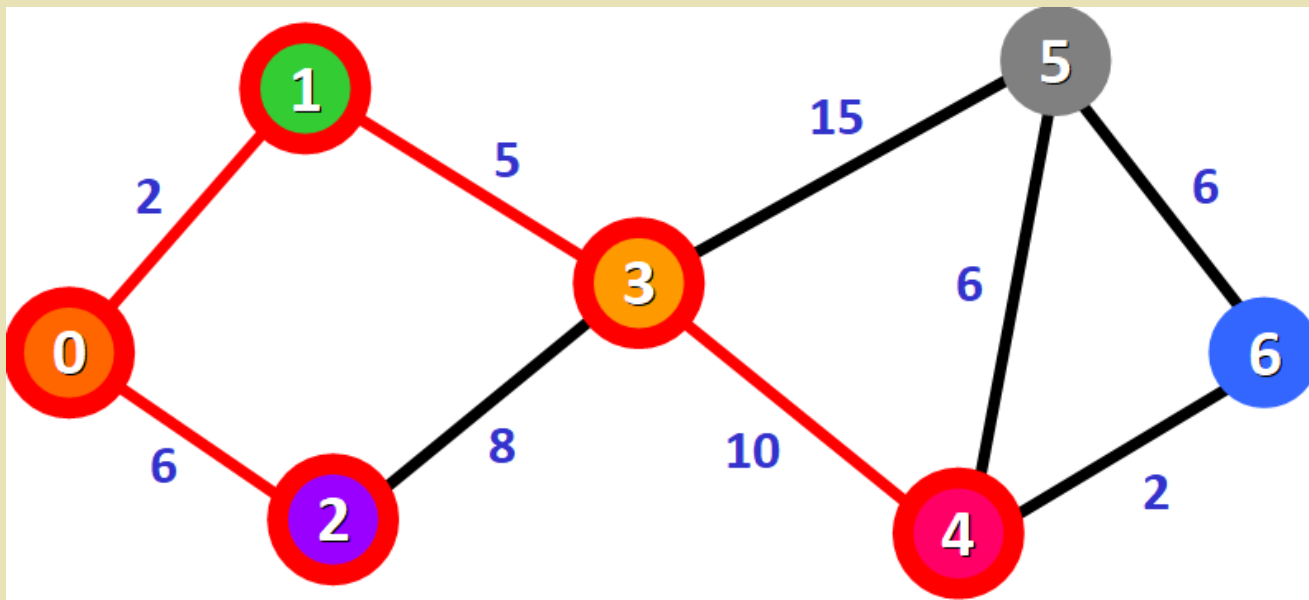
Dijkstra's algorithm

Step 5: check new adjacent nodes to nodes in the path. Update distance (4,5). Choose the unvisited node with shortest distance to add to path.



Dijkstra's algorithm

Step 6: check new adjacent nodes to nodes in the path. Update distance (not need for 5, needed for 6).



Distance:

0: 0
1: ~~0~~ 2 ■
2: ~~0~~ 6 ■
3: ~~0~~ 7 ■
4: ~~0~~ 17 ■
5: ~~0~~ 22
6: ~~0~~ 19 ■

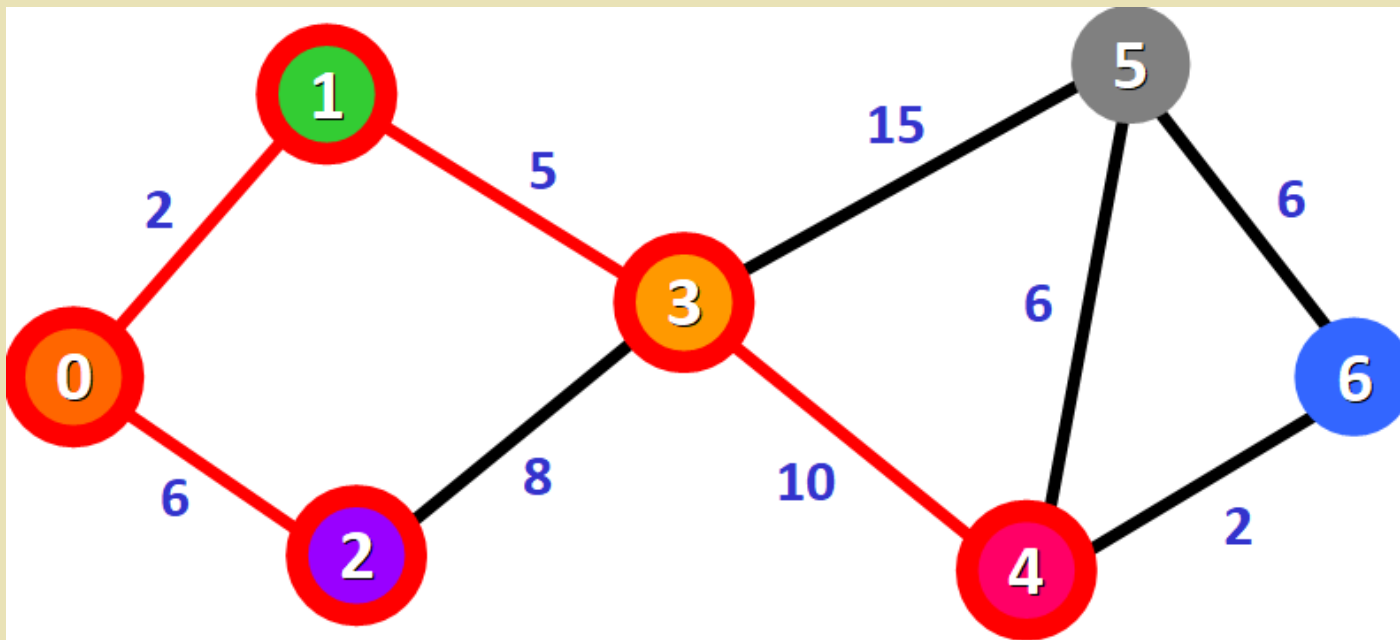
Two Options for reaching 5:

3 -> 5 (current) 22

4 -> 5 23 (17+ 6).

Dijkstra's algorithm

Step 6: Choose the unvisited node with shortest distance to the source(6).
Add to the path.



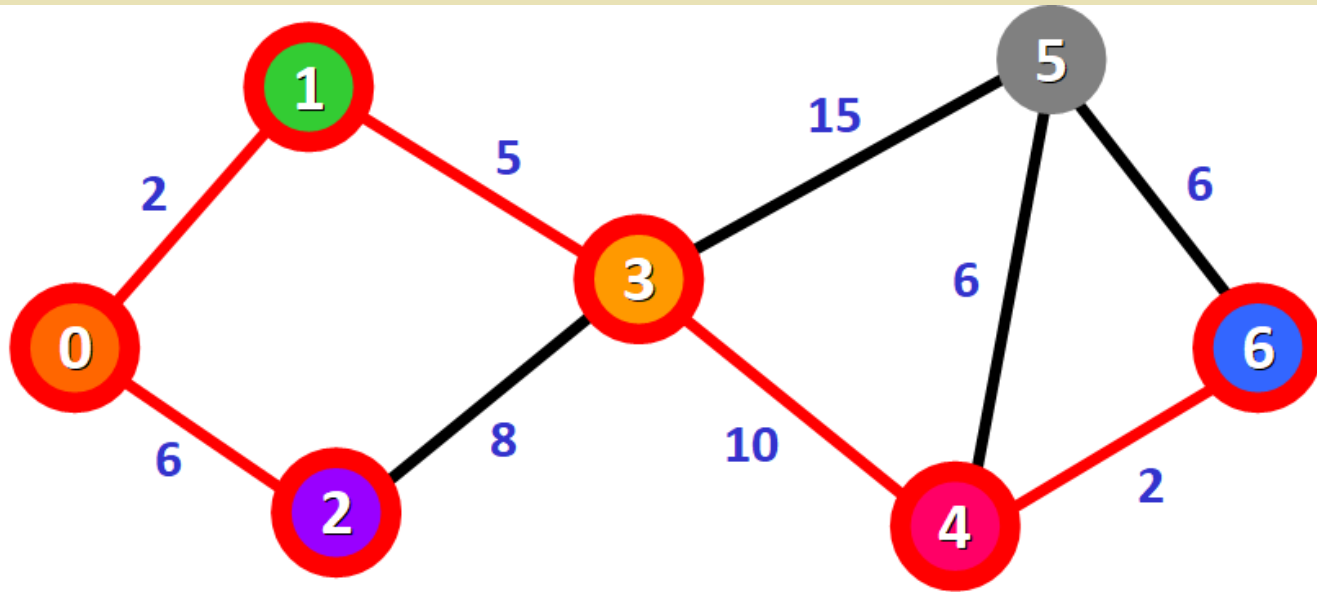
Unvisited Nodes: ~~{0, 1, 2, 3, 4, 5, 6}~~

Distance:

0: 0
1: ~~2~~ ■
2: ~~6~~ ■
3: ~~7~~ ■
4: ~~17~~ ■
5: ~~22~~
6: ~~19~~ ■

Dijkstra's algorithm

Step 7: check new adjacent nodes to nodes in the path. Update distance
Choose the unvisited node with shortest distance to add to path.



Distance:

0:	0
1:	2 ■
2:	6 ■
3:	7 ■
4:	17 ■
5:	22
6:	19 ■

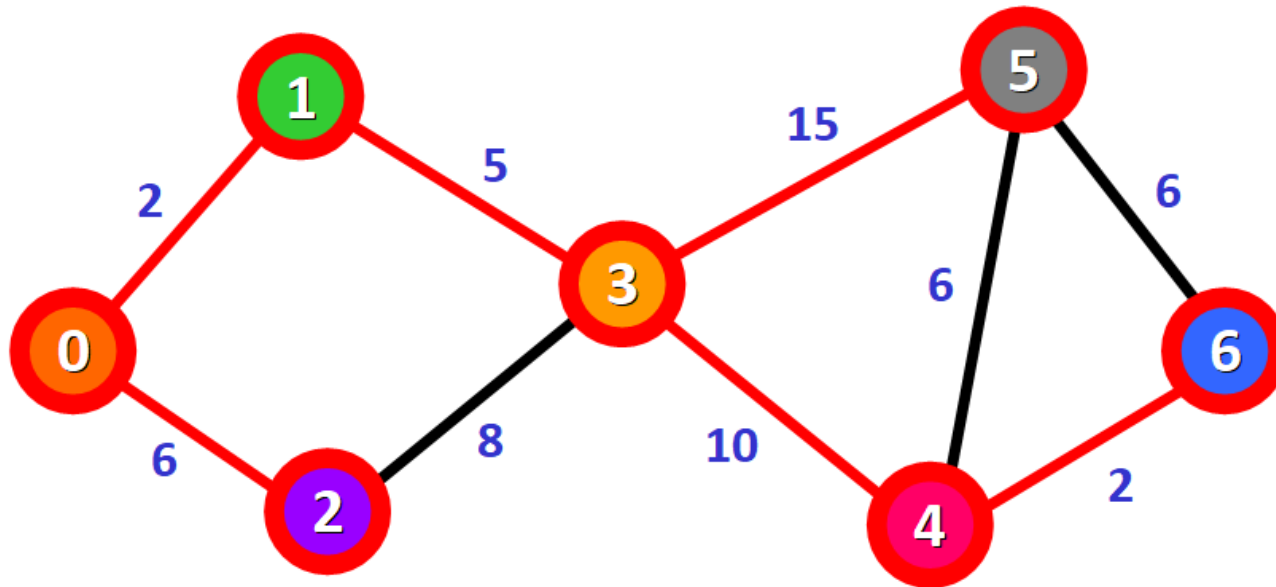
Two Options for reaching 5:

3 -> 5 (current) 22

6 -> 5 25 (19+ 6)

Dijkstra's algorithm

Step 7: All nodes visited. The red lines mark the edges that belong to the shortest path.



Unvisited Nodes: ~~{0, 1, 2, 3, 4, 5, 6}~~

Distance:

0: 0
1: ~~0~~ 2 ■
2: ~~0~~ 6 ■
3: ~~0~~ 7 ■
4: ~~0~~ 17 ■
5: ~~0~~ 22 ■
6: ~~0~~ 19 ■

Dijkstra(Graph, source):

Initialize:

$Q \leftarrow$ set of all vertices in Graph

$\text{distance}[v] \leftarrow \infty$ for each vertex v in Graph

$\text{previous}[v] \leftarrow \text{UNDEFINED}$ for each vertex v in Graph

$\text{distance}[\text{source}] \leftarrow 0$

While Q is not empty:

$u \leftarrow$ vertex in Q with smallest $\text{distance}[u]$

Remove u from Q

For each neighbor v of u :

$\text{alt} \leftarrow \text{distance}[u] + \text{length}(u, v)$

If $\text{alt} < \text{distance}[v]$:

$\text{distance}[v] \leftarrow \text{alt}$

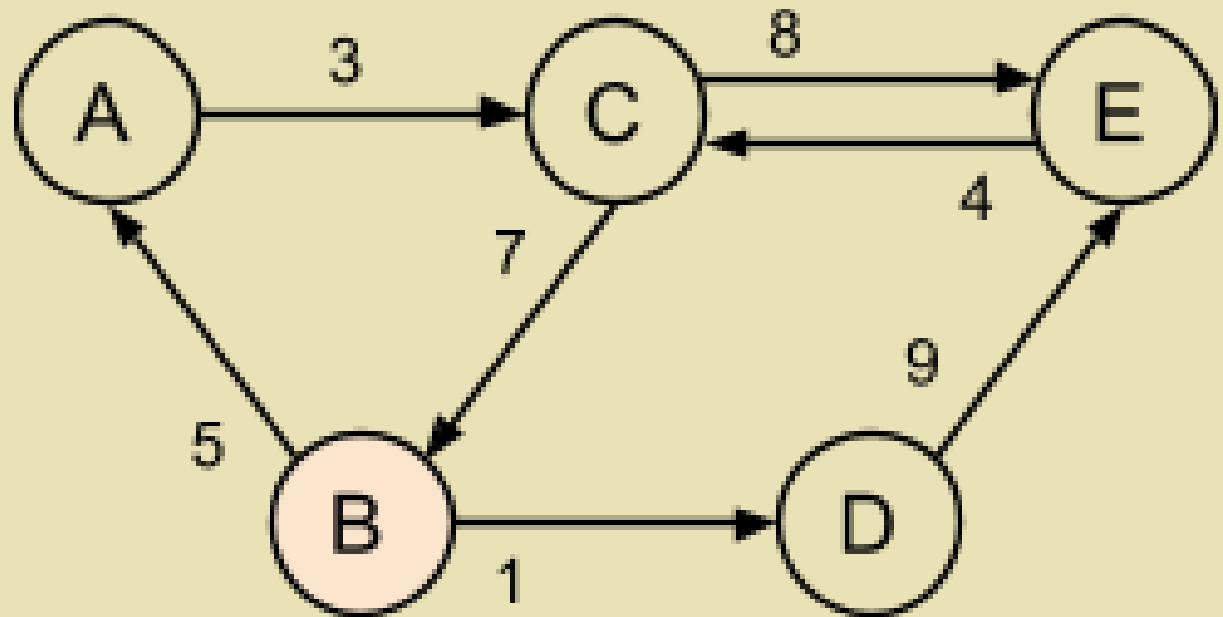
$\text{previous}[v] \leftarrow u$

Return $\text{distance}[], \text{previous}[]$

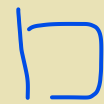
Dijkstra's algorithm

Dijkstra's Algorithm -- Example

- ◆ Perform Dijkstra's shortest path algorithm on the graph below with starting vertex B.



- ◆ Which vertex is visited after B?





Algorithm efficiency

- ◆ If the unvisited vertex queue is implemented using a list, the runtime for Dijkstra's shortest path algorithm is $O(V^2)$.
- ◆ The outer loop executes V times to visit all vertices. In each outer loop execution, dequeuing the vertex from the queue requires searching all vertices in the list, which has a runtime of $O(V)$.



Algorithm efficiency

- ◆ For each vertex, the algorithm follows the subset of edges to adjacent vertices; following a total of E edges across all loop executions. Given $E < V^2$, the runtime is $O(V * V + E) = O(V^2 + E) = O(V^2)$.
- ◆ Implementing the **unvisited vertex queue** using a standard binary heap reduces the runtime to $O((E + V) \log V)$, and using a Fibonacci heap data structure (not discussed in this material) reduces the runtime to $O(E + V \log V)$.



Bellman-Ford's Algorithm

- ◆ The Bellman-Ford shortest path algorithm, created by Richard Bellman and Lester Ford, Jr., determines the shortest path from a start vertex to each vertex in a graph.
 - Can work on graphs with negative weights, but **no negative weight cycles**.
- ◆ A vertex's **distance** is the shortest path distance from the start vertex.
- ◆ A vertex's **predecessor pointer** points to the previous vertex along the shortest path from the start vertex.

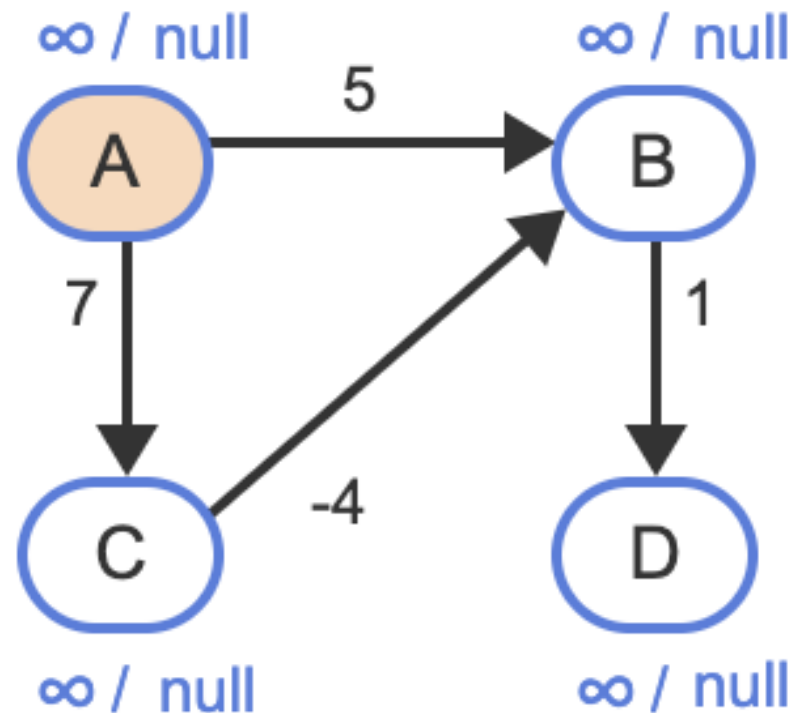
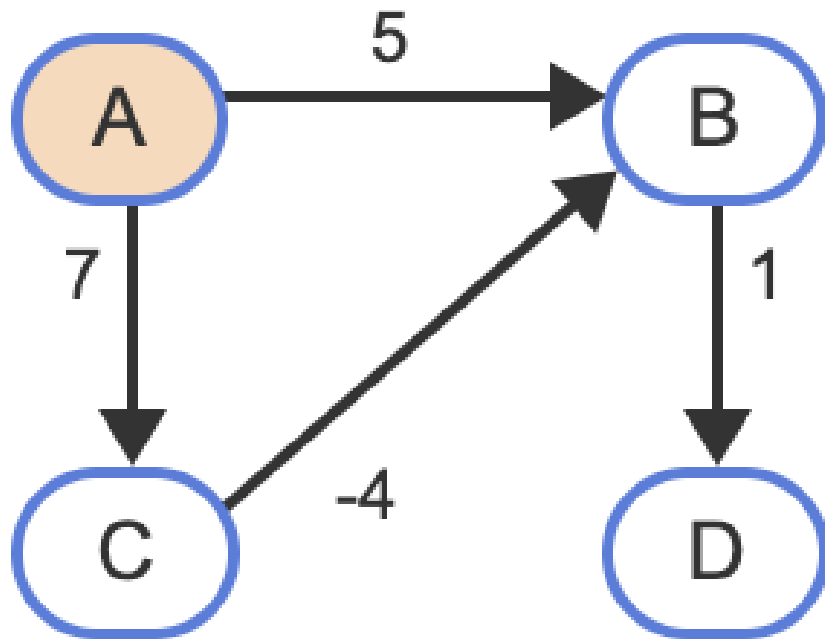


Bellman-Ford's Algorithm

- ◆ Initialize all vertices' current distances to infinity (∞) and predecessors to null, and assigns the start vertex with a distance of 0.
- ◆ Perform $V-1$ main iterations, visiting all vertices in the graph in each iteration by checking all **edges**.
 - For each edge (u, v) with weight w : If going through u gives a shorter path to v from the source (i.e., $\text{distance}[v] > \text{distance}[u] + w$), we update the $\text{distance}[v]$ as $\text{distance}[u] + w$.
 - This is also called “**Relaxation of Edges**”.

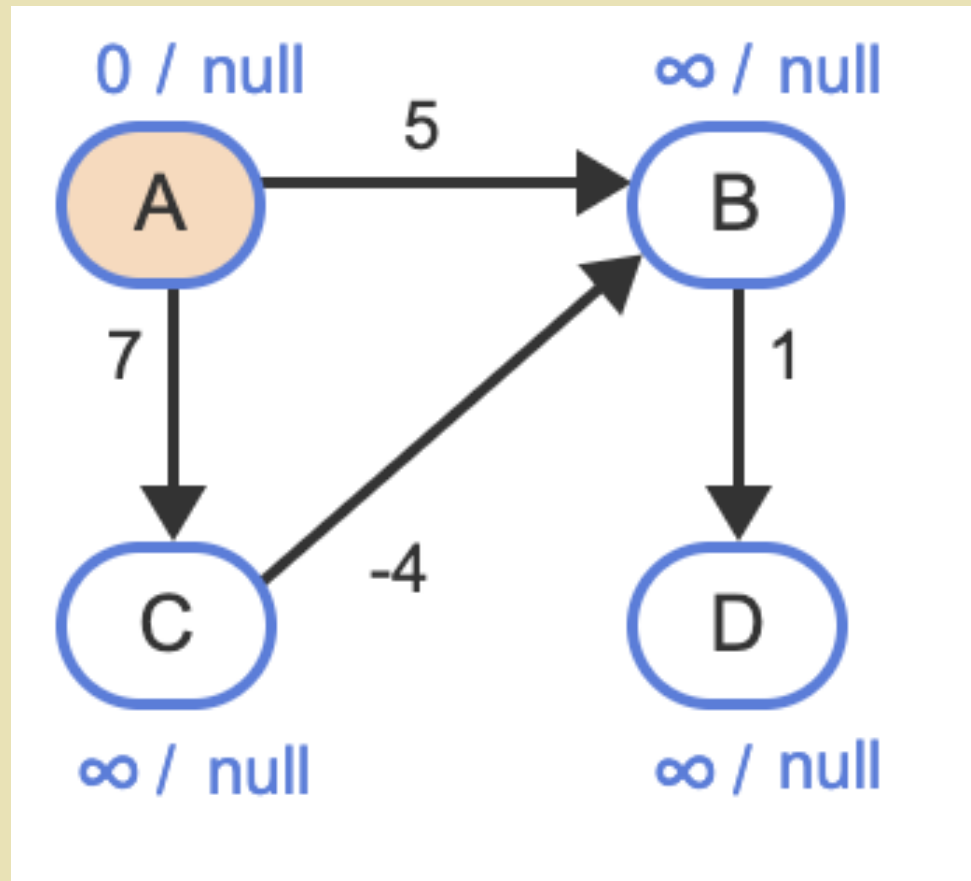
Bellman-Ford's Algorithm

Initialization



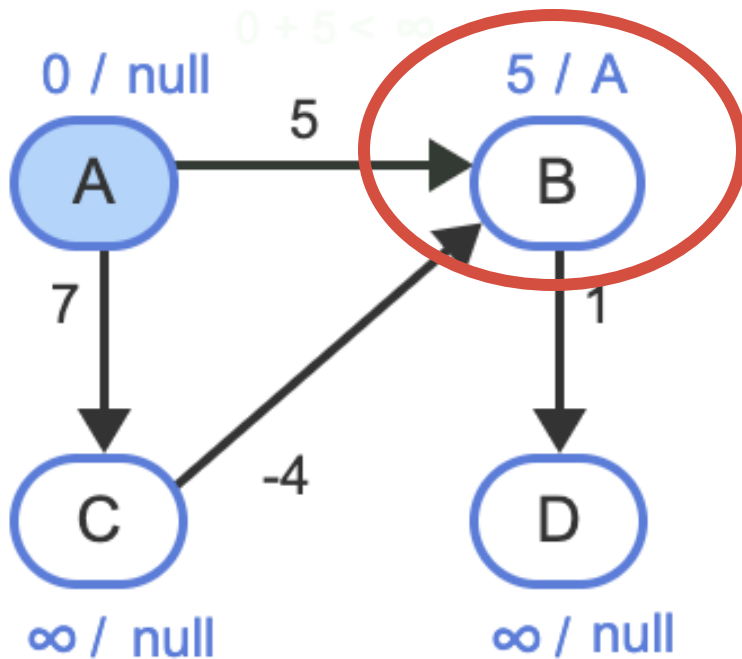
Bellman-Ford's Algorithm

Initialization



Bellman-Ford's Algorithm

Iteration 1



Edges

A \rightarrow B

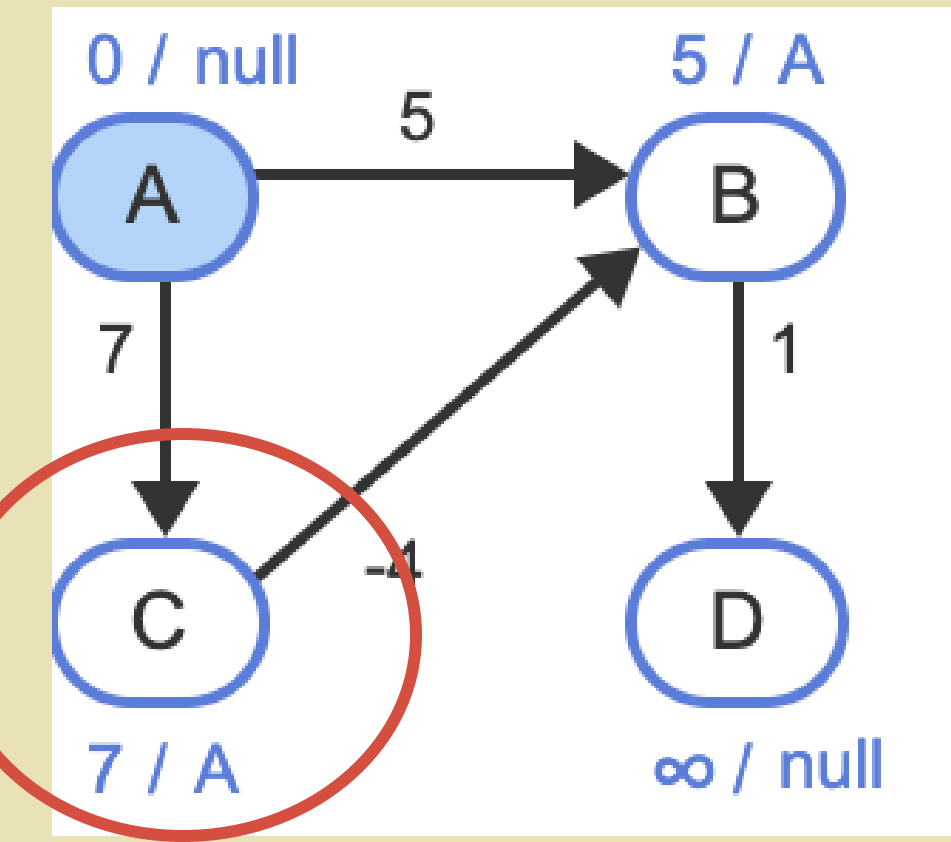
A \rightarrow C

B \rightarrow D

C \rightarrow B

Bellman-Ford's Algorithm

Iteration 1



Edges

A \rightarrow B

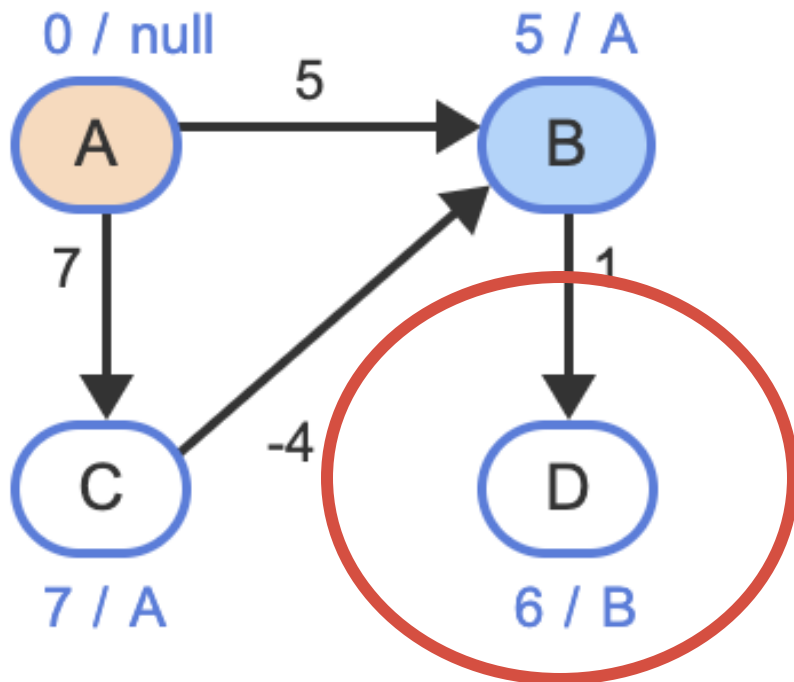
A \rightarrow C

B \rightarrow D

C \rightarrow B

Bellman-Ford's Algorithm

Iteration 1



Edges

$A \rightarrow B$

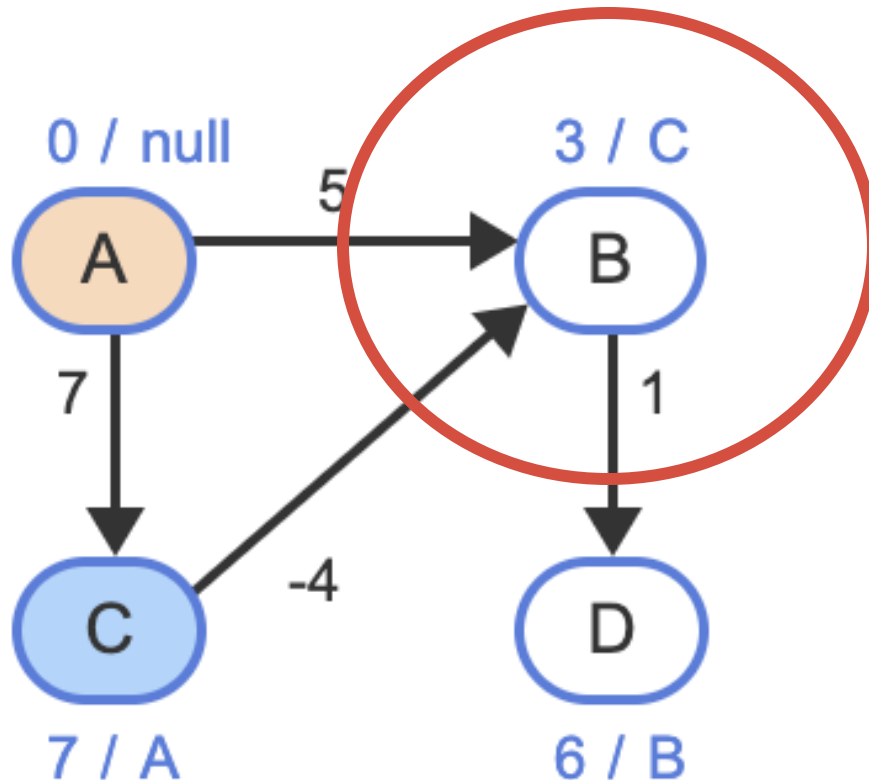
$A \rightarrow C$

$B \rightarrow D$

$C \rightarrow B$

Bellman-Ford's Algorithm

Iteration 1



Edges

A → B

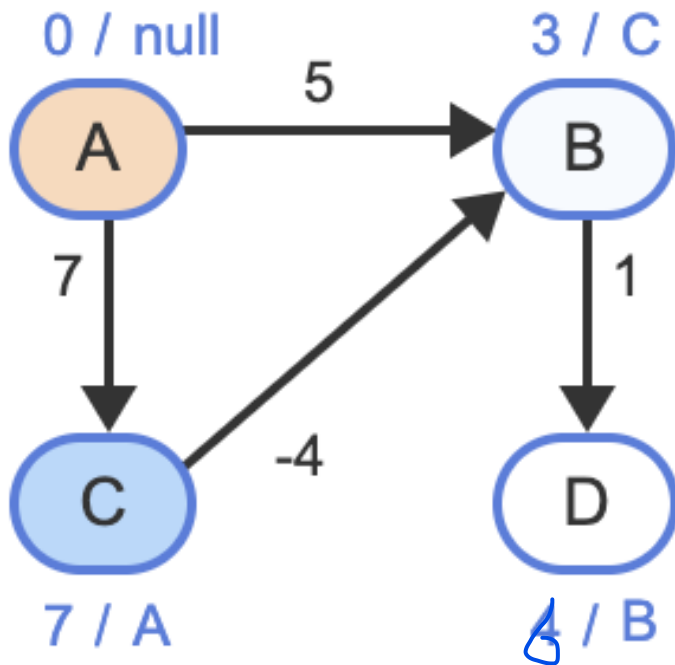
A → C

B → D

C → B

Bellman-Ford's Algorithm

Iteration 2



Edges

$A \rightarrow B$

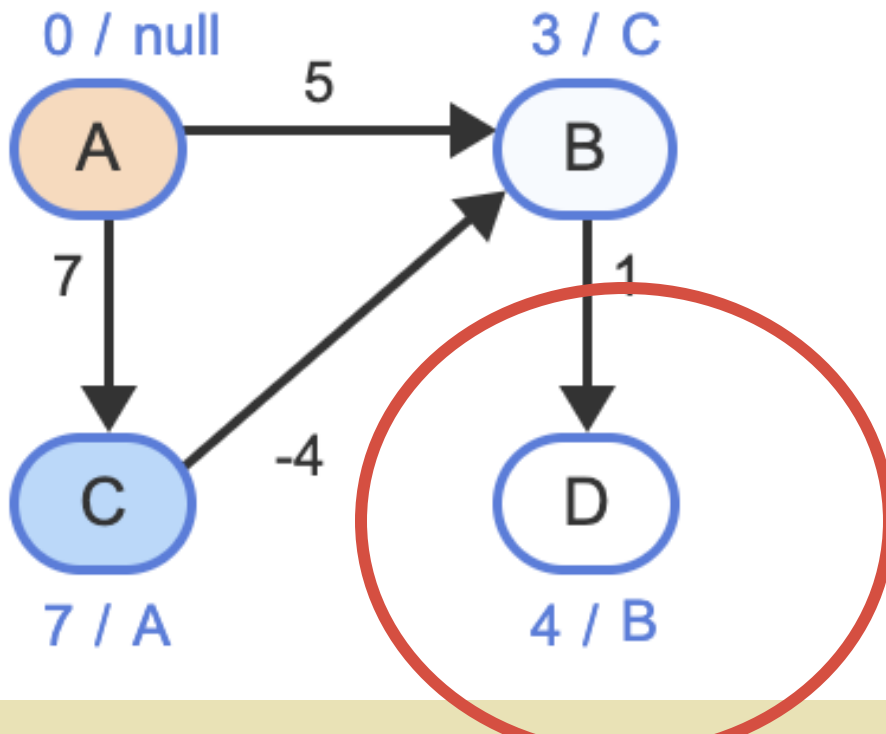
$A \rightarrow C$

$B \rightarrow D$

$C \rightarrow B$

Bellman-Ford's Algorithm

Iteration 2



Edges

A → B

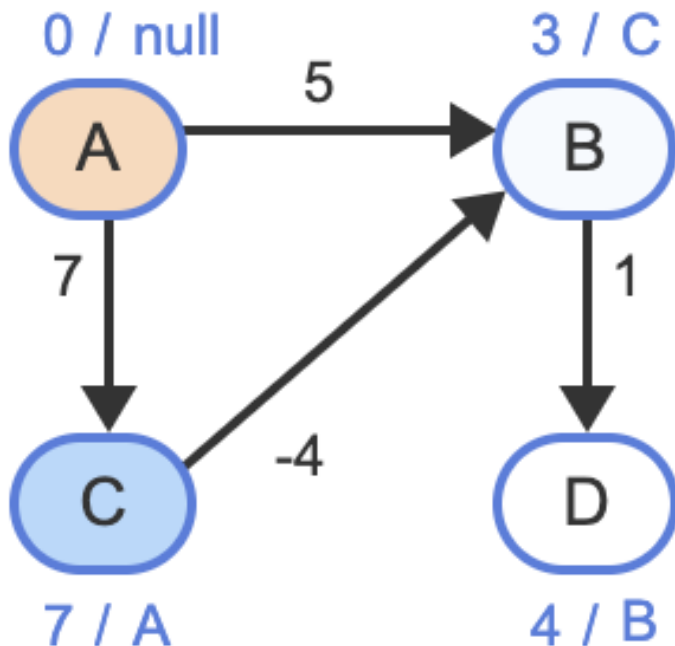
A → C

B → D

C → B

Bellman-Ford's Algorithm

Iteration 3 (last one)



Edges

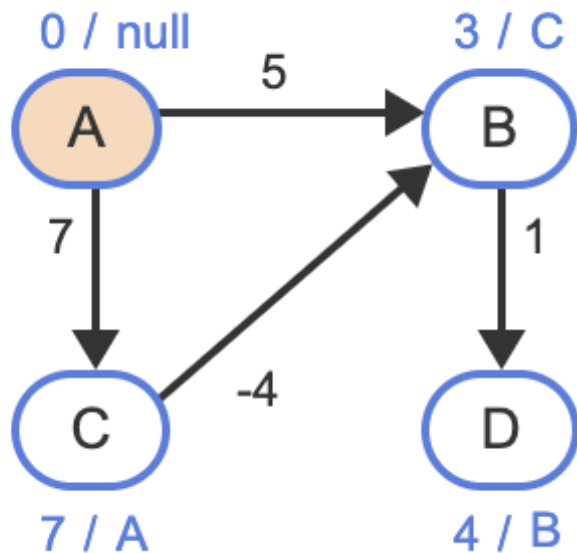
$A \rightarrow B$

$A \rightarrow C$

$B \rightarrow D$

$C \rightarrow B$

Bellman-Ford's Algorithm



Shortest path and length

Path from A to D:

A C B D

Path length: 4

BellmanFord(startVertex):

Initialize all vertices with infinite distance and null predecessor
for each vertex vertex in graph:

vertex.distance = Infinity

vertex.predecessor = null

Set the starting vertex distance to 0

startVertex.distance = 0

Relax all edges for (number of vertices - 1) times

for iteration = 1 to (number of vertices - 1):

for each vertex currentVertex in graph:

for each adjacent vertex adjacentVertex of currentVertex:

edgeWeight = weight of edge(currentVertex,

adjacentVertex)

newDistance = currentVertex.distance + edgeWeight

If a shorter path is found, update distance and
predecessor

if newDistance < adjacentVertex.distance:

adjacentVertex.distance = newDistance

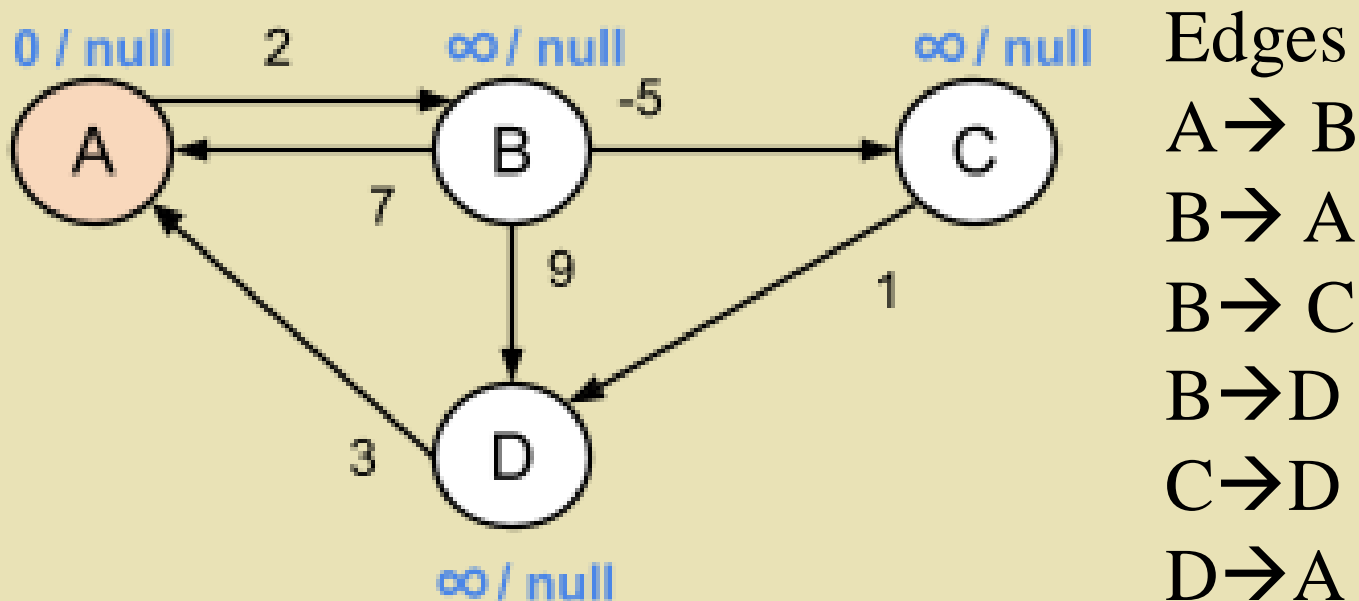
adjacentVertex.predecessor = currentVertex

Check for negative weight cycle (later slides)

Bellman- Ford's algorithm

Bellman-Ford's Algorithm -- Example

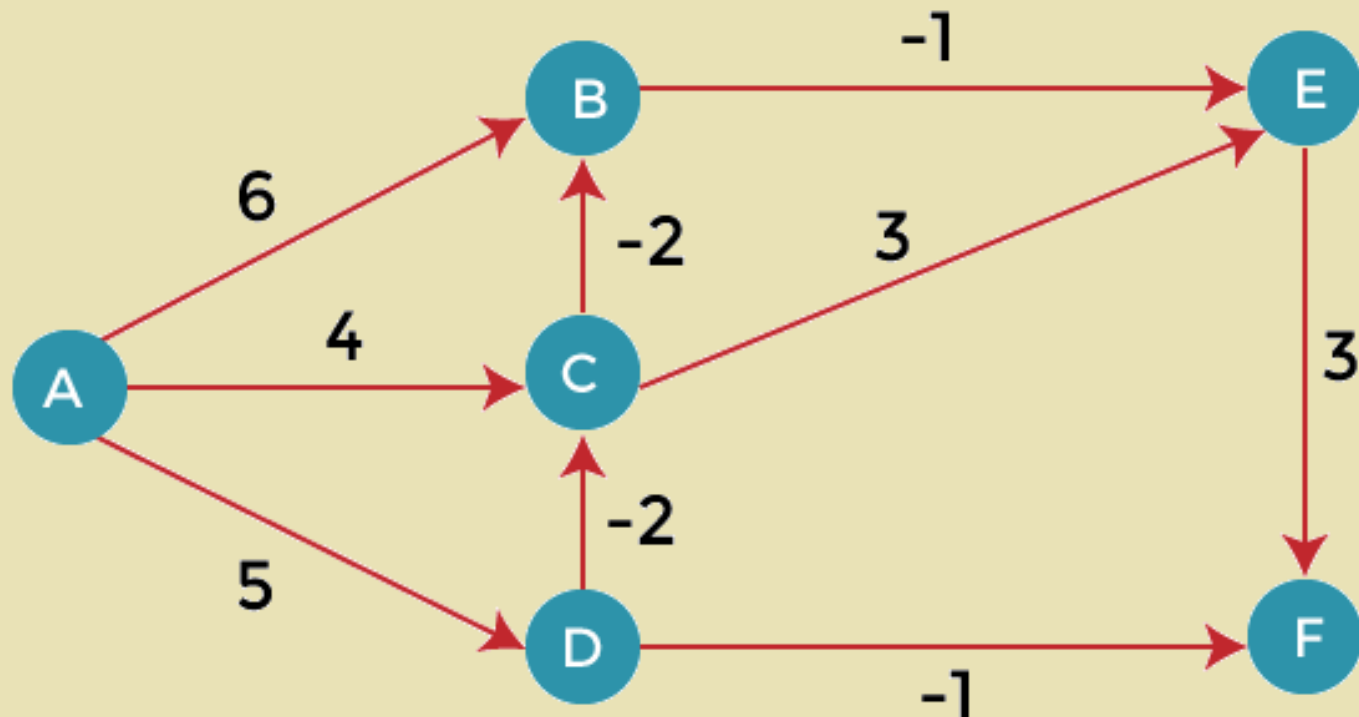
- ◆ Perform Bellman-Ford's shortest path algorithm on the graph below with starting vertex A. Order of edge list is given.



Bellman-Ford's Algorithm -- Example

- ◆ Another good example

<https://www.javatpoint.com/bellman-ford-algorithm>



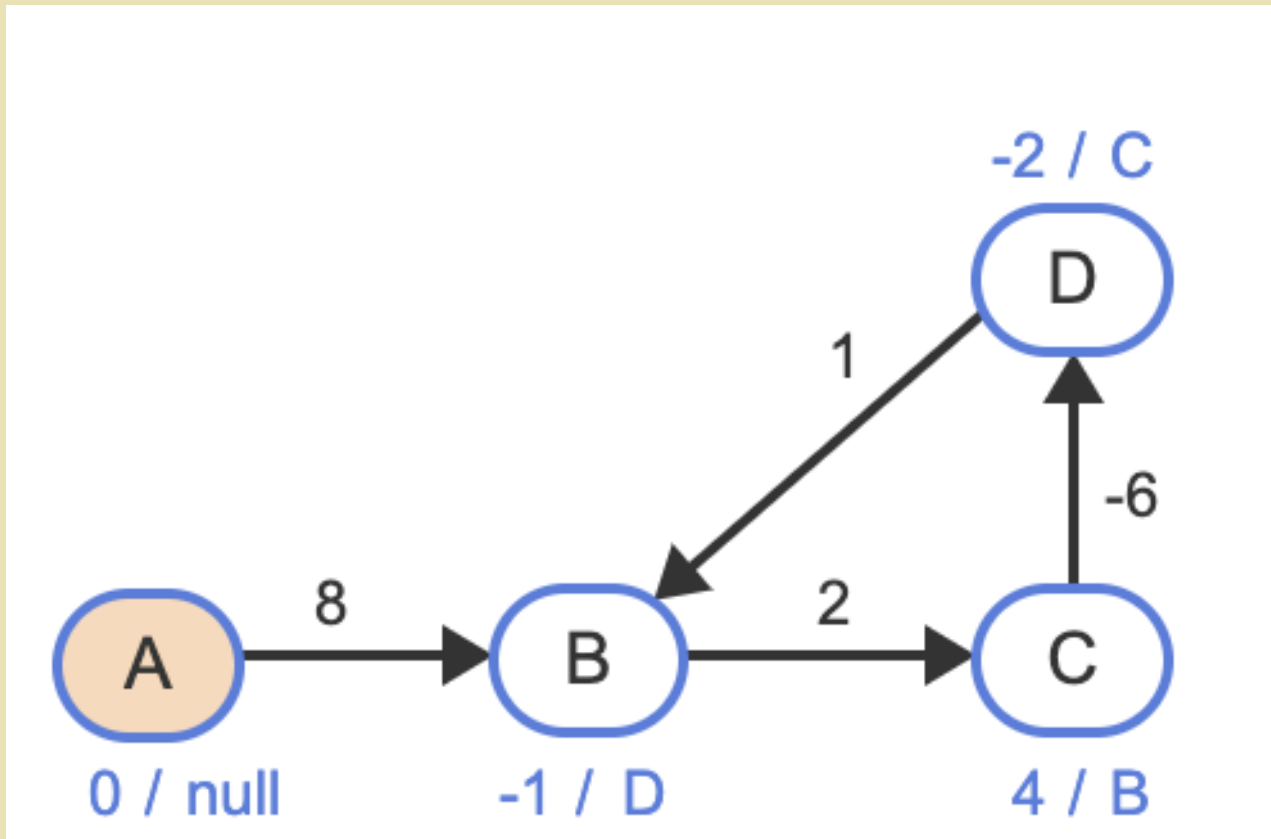


Bellman-Ford's Algorithm

- ◆ Negative weight cycle check (after $V-1$ iterations)
 - For each vertex in the graph, adjacent vertices are checked for a shorter path.
 - If shorter path from startV to adjV is still found, a negative edge weight cycle exists

Bellman-Ford's Algorithm

- ◆ Negative weight cycle check (after $V-1$ iterations)



Edges

$A \rightarrow B$

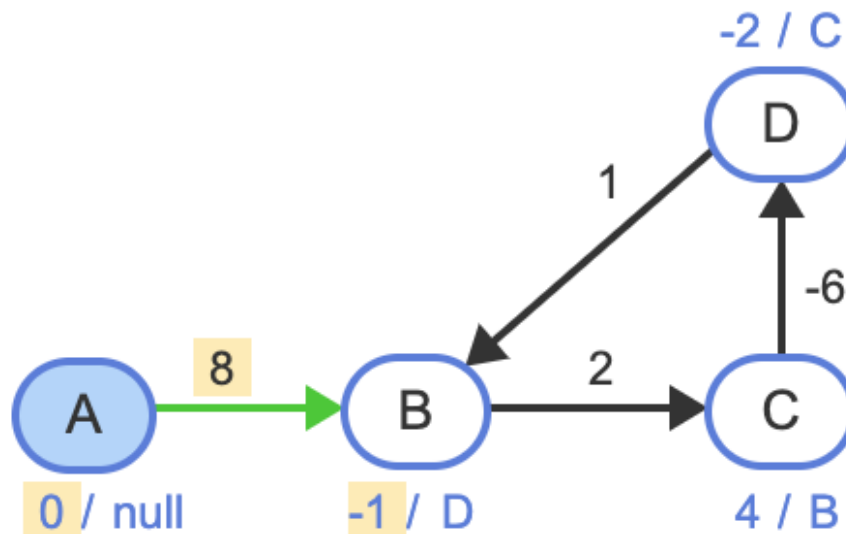
$B \rightarrow C$

$C \rightarrow D$

$D \rightarrow B$

Bellman-Ford's Algorithm

- ◆ Negative weight cycle check (after V-1 iterations)



Shorter path still found?

$$0 + 8 < -1 \quad \text{X}$$

Edges

A → B

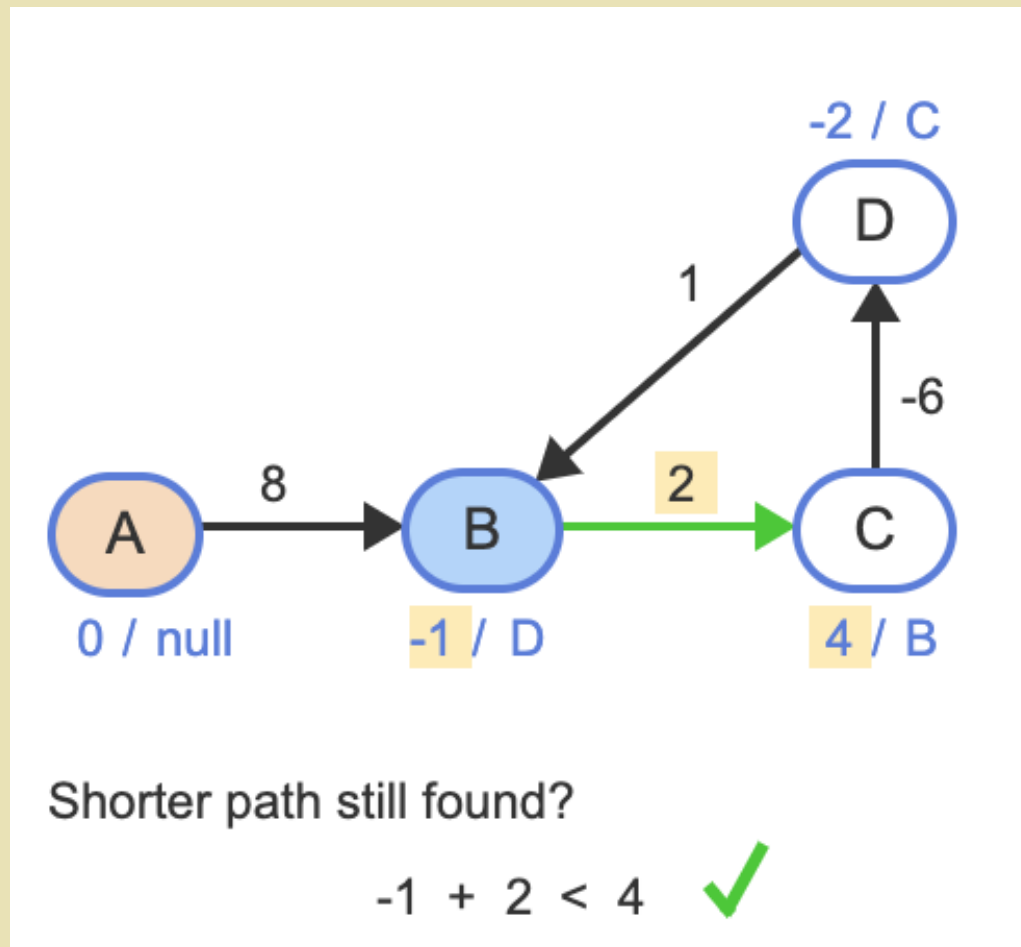
B → C

C → D

D → B

Bellman-Ford's Algorithm

- ◆ Negative weight cycle check (after $V-1$ iterations)



Edges

$A \rightarrow B$

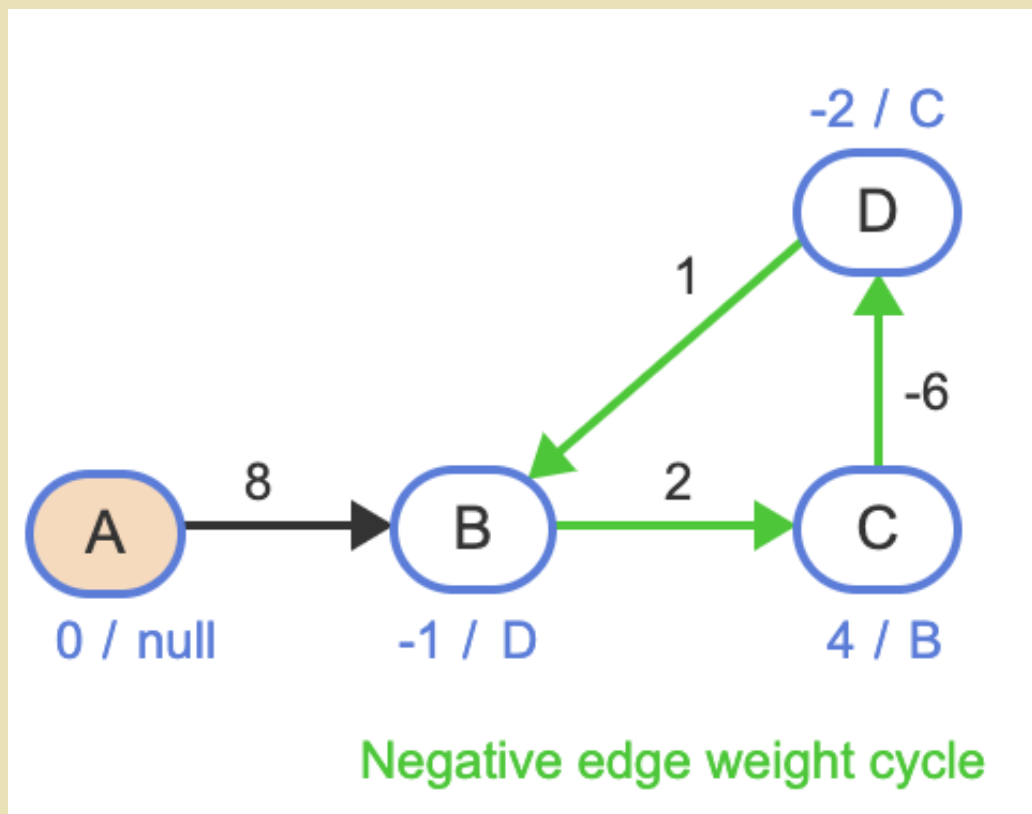
$B \rightarrow C$

$C \rightarrow D$

$D \rightarrow B$

Bellman-Ford's Algorithm

- ◆ Negative weight cycle check (after V-1 iterations)



Edges

A → B

B → C

C → D

D → B

Bellman-Ford's algorithm negative weight cycle

```
# Check for negative weight cycle
for each vertex currentVertex in graph:
    for each adjacent vertex adjacentVertex of currentVertex:
        edgeWeight = weight of edge(currentVertex,
adjacentVertex)
        if currentVertex.distance + edgeWeight <
adjacentVertex.distance:
            print("Graph contains a negative weight cycle")
return
```




Bellman-Ford's Algorithm

- ◆ Time complexity

- Best Case: $O(E)$, when distance array after 1st and 2nd relaxation are same , we can simply stop further processing.
- Worst Case: $O(V * E)$



References and Useful Resources

- ◆ Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction <https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/#:~:text=Dijkstra's%20Algorithm%20finds%20the%20shortest,node%20and%20all%20other%20nodes.>
- ◆ Bellman-Ford's Algorithm <https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>
- ◆ Dijkstra's vs Bellman-Ford Algorithm <https://medium.com/@brianpatrao1996/dijkstras-vs-bellman-ford-algorithm-383e4771c2cb#:~:text=One%20of%20the%20main%20benefits,is%20the%20number%20of%20vertices.>

That's
about this
lecture!

