SHORTEST PATH IN A GRAPH

- INTRODUCING
- WHAT IS DIJKSTRA’S ALGORITHM?
- EXAMPLE OF DIJKSTRA’S ALGORITHM
- DIJKSTRA’S TIME AND SPACE COMPLEXITY
- BELLMAN-FORD ALGORITHM
- EXAMPLE OF BELLMAN-FORD ALGORITHM
- BELLMAN-FORD ALGORITHM PSEUDOCODE
- BELLMAN-FORD TIME AND SPACE COMPLEXITY
- PRACTICE: SHORTEST PATH PROBLEM

Author: prof. Yevhenii Borodavka

INTRODUCING

Have you ever wondered how Google Maps can effortlessly calculate the best route to your destination?

The Dijkstra algorithm is one of the many key algorithms employed by Google Maps to find and optimize the best route available for the user.

The Dijkstra algorithm is a method for finding the shortest path among nodes in a weighted graph and in fact, is the best known algorithm for this class of problems.

WHAT IS DIJKSTRA'S ALGORITHM?

How it works:

- Define the starting node with a distance of 0.
- Set all the other nodes to a distance of ∞.
- Explore neighboring nodes and update their distances (if lower than its current value).
- Visit the unvisited node with the smallest distance.
- Repeat steps 3 and 4 until the shortest distance is found.

EXAMPLE OF DIJKSTRA'S ALGORITHM

Let's go through a step-by-step example and find the shortest distance between nodes **A** and **G**.

Note: The terms node and vertex are used interchangeably.

Note: A priority queue (min-heap) is a data structure that orders data in ascending order.

EXAMPLE OF DIJKSTRA'S ALGORITHM

How to understand the graphs:

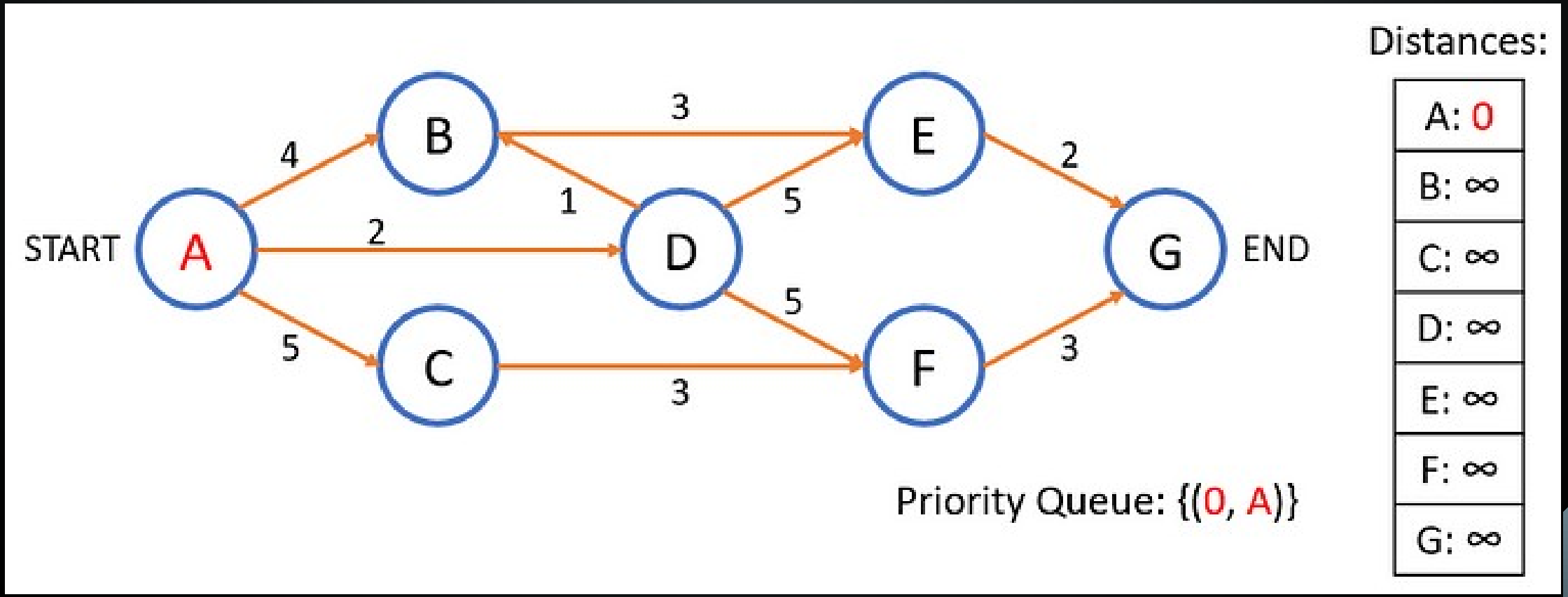
- **Red vertices and edges**: represent the vertices being explored.
- **Red numbers**: signify an update in the shortest distance.
- **Red tuples**: represents newly added tuples to the priority queue.
- **Blue tuples**: represent tuples being removed from the queue.

EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 1:

- Initialize the graph by setting the starting node at a distance of **0**.
- The distances to the other nodes are still unknown so are set to a worst-case distance of ∞.
- Since vertex **A** has been explored, vertex **A** and its corresponding distance are added to the priority queue.

EXAMPLE OF DIJKSTRA'S ALGORITHM

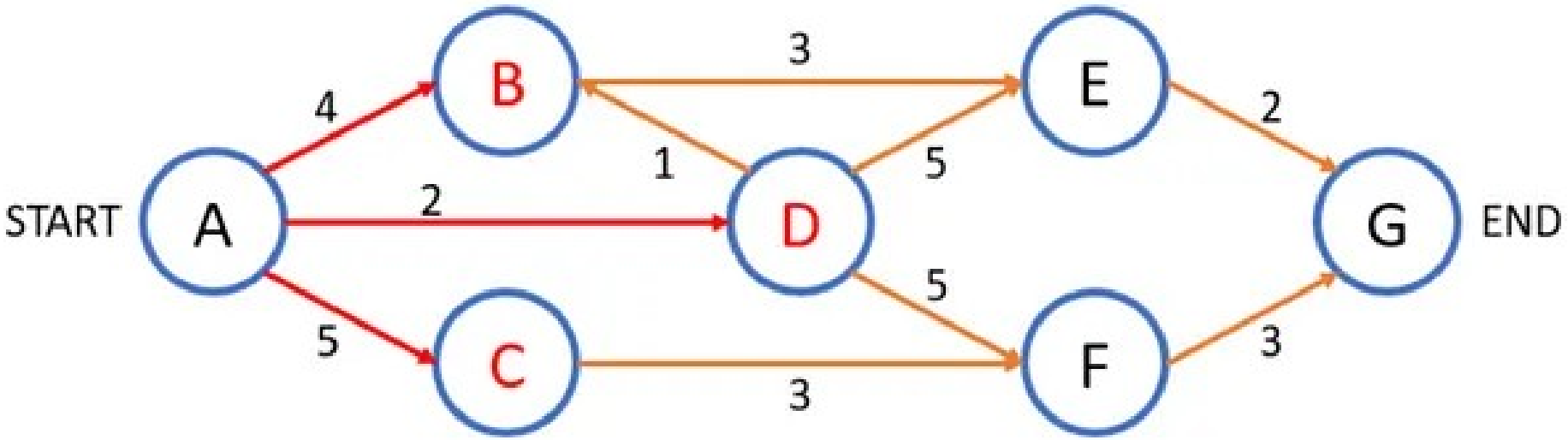


EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 2:

- Explore the edges stemming from the first tuple in the priority queue.
- Remove the tuple from the priority queue after it has been explored.
- Update the distances of the explored vertices if necessary.
- Add the tuples corresponding to each explored vertex to the priority queue.

EXAMPLE OF DIJKSTRA'S ALGORITHM



Distances:

A: 0
B: 4
C: 5
D: 2
E: ∞
F: ∞
G: ∞

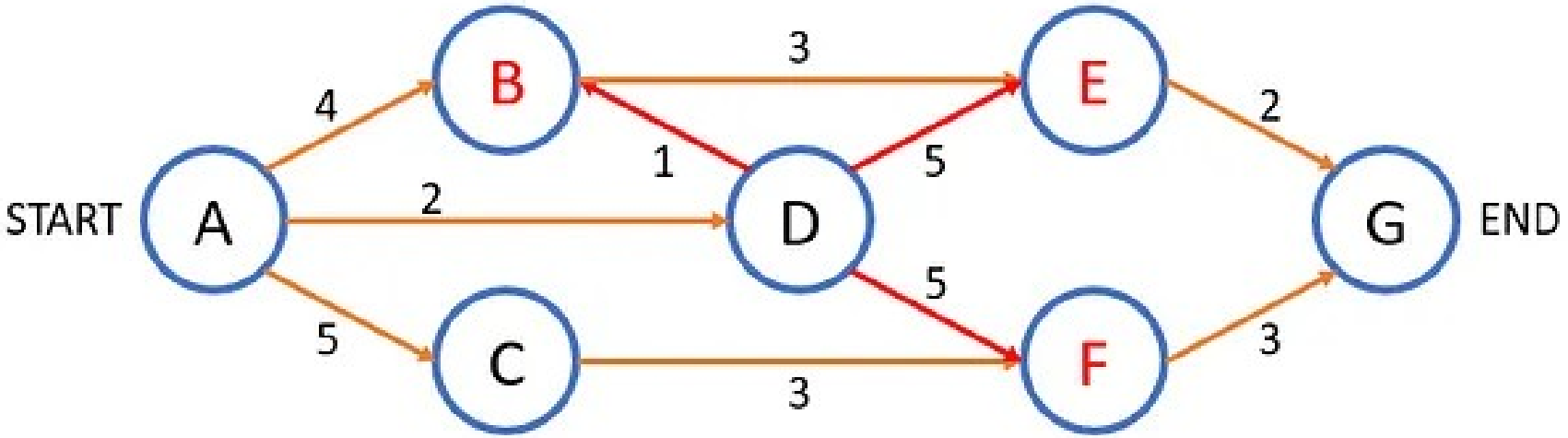
Priority Queue: {(0, A),
(2, D),
(4, B),
(5, C)}

EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 3:

- Explore the edges connecting to vertex D.
- Remove (2, D) from the priority queue.
- Update the shortest distances of the explored vertices.
- Add the corresponding tuples to the priority queue.

EXAMPLE OF DIJKSTRA'S ALGORITHM



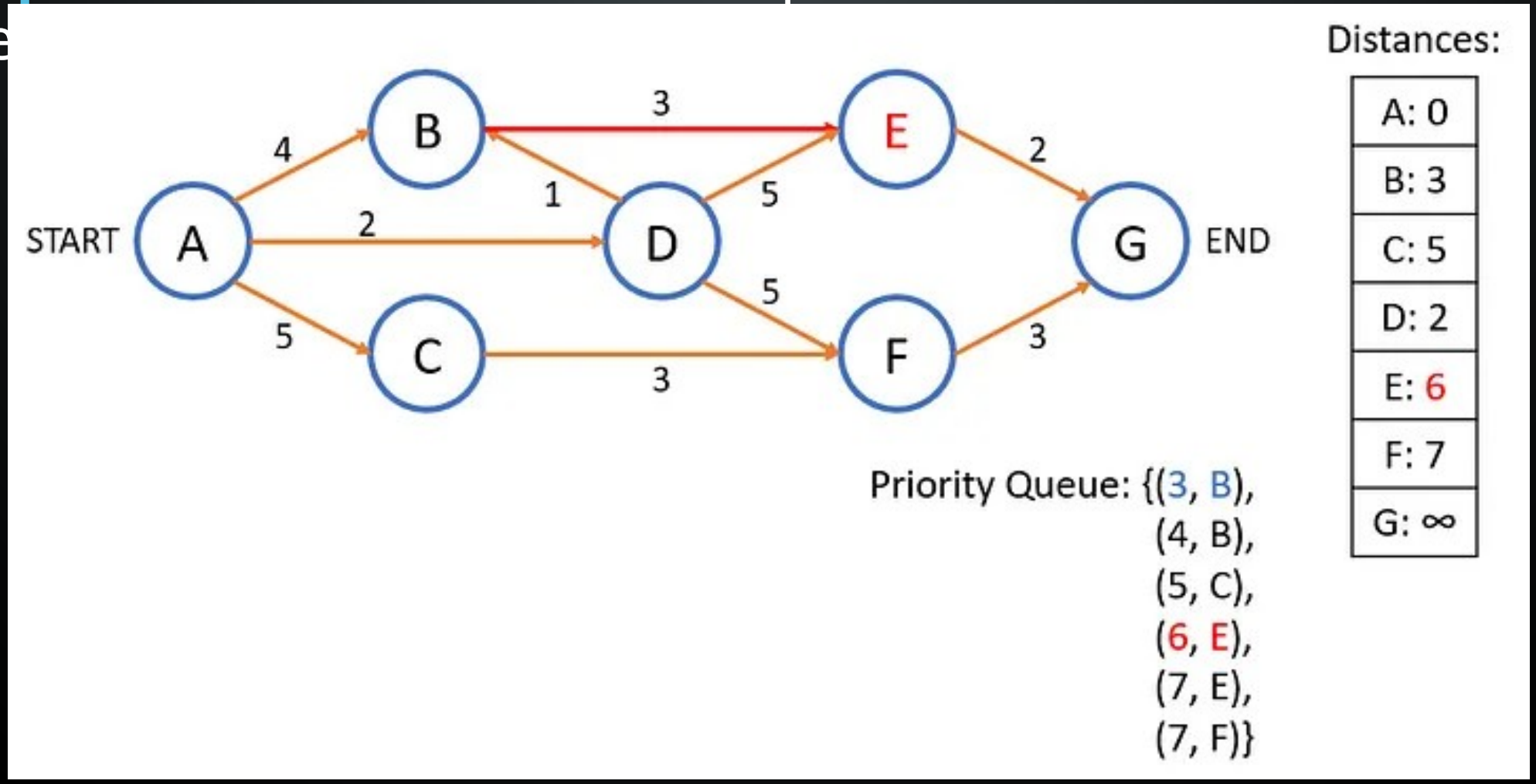
Distances:

A: 0
B: 3
C: 5
D: 2
E: 7
F: 7
G: ∞

Priority Queue: {(2, D),
(3, B),
(4, B),
(5, C),
(7, E),
(7, F)}

EXAMPLE OF THE DIJKSTRA'S ALGORITHM

Step 4: Continue the same process as described in the pre

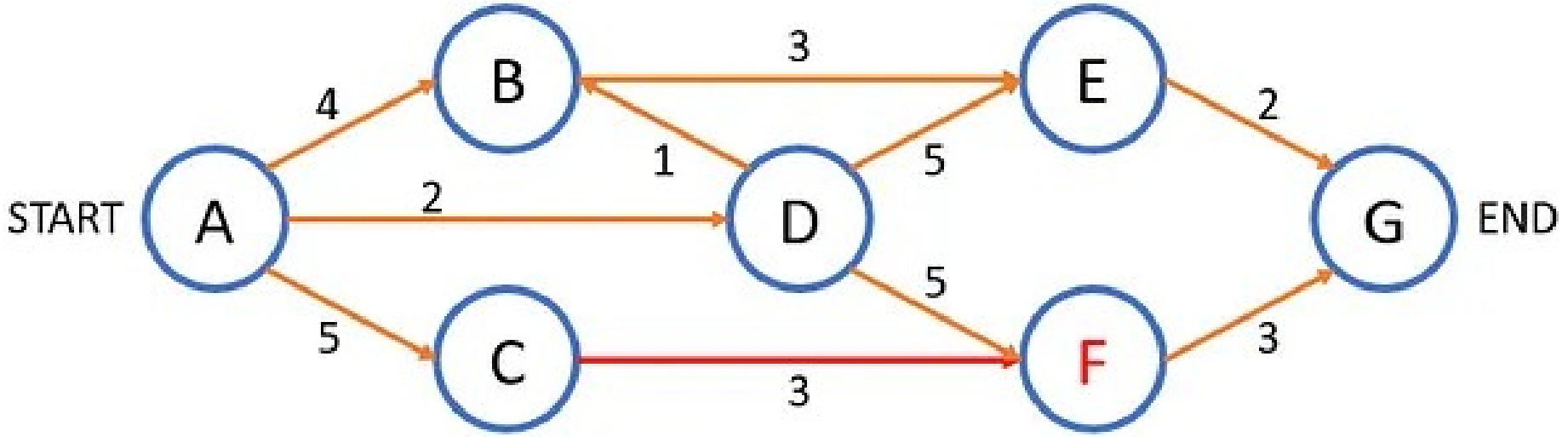


EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 5: This step is a little different

- Essentially, we only explore the top vertex from the priority queue if its distance is smaller than the value already stored in the distance list. 4 > 3, which means a shorter distance to vertex **B** has already been found. Therefore, it is unnecessary to visit it again.
- Due to the above reasons, we explore the edges from the next tuple, vertex **C**, and then discard both tuples from the priority queue.
- The distance to **F** is larger than previously found as a result is not updated in distances, but the corresponding tuple is still

EXAMPLE OF DIJKSTRA'S ALGORITHM



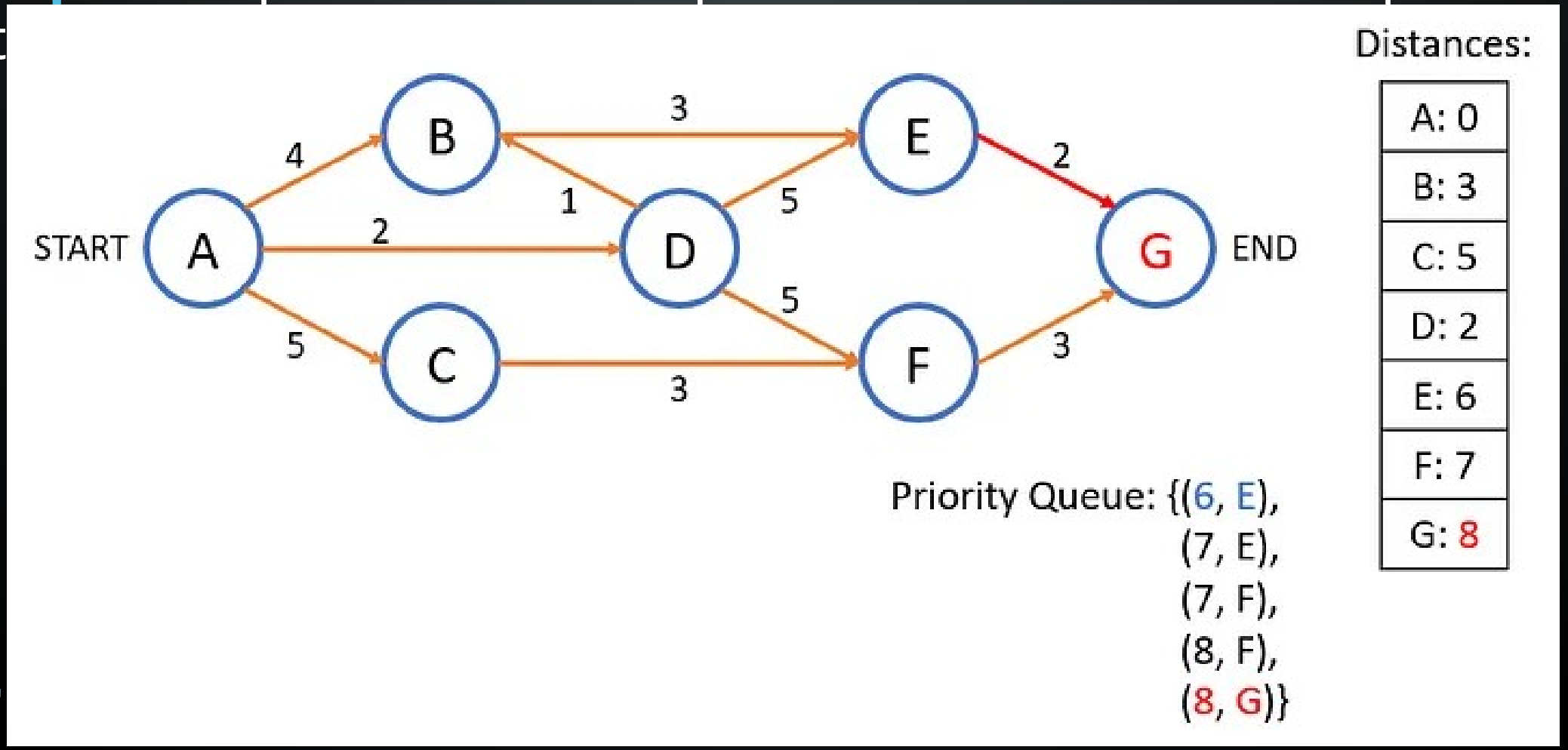
Distances:

A: 0
B: 3
C: 5
D: 2
E: 6
F: 7
G: ∞

Priority Queue: {(4, B),
(5, C),
(6, E),
(7, E),
(7, F),
(8, F)}

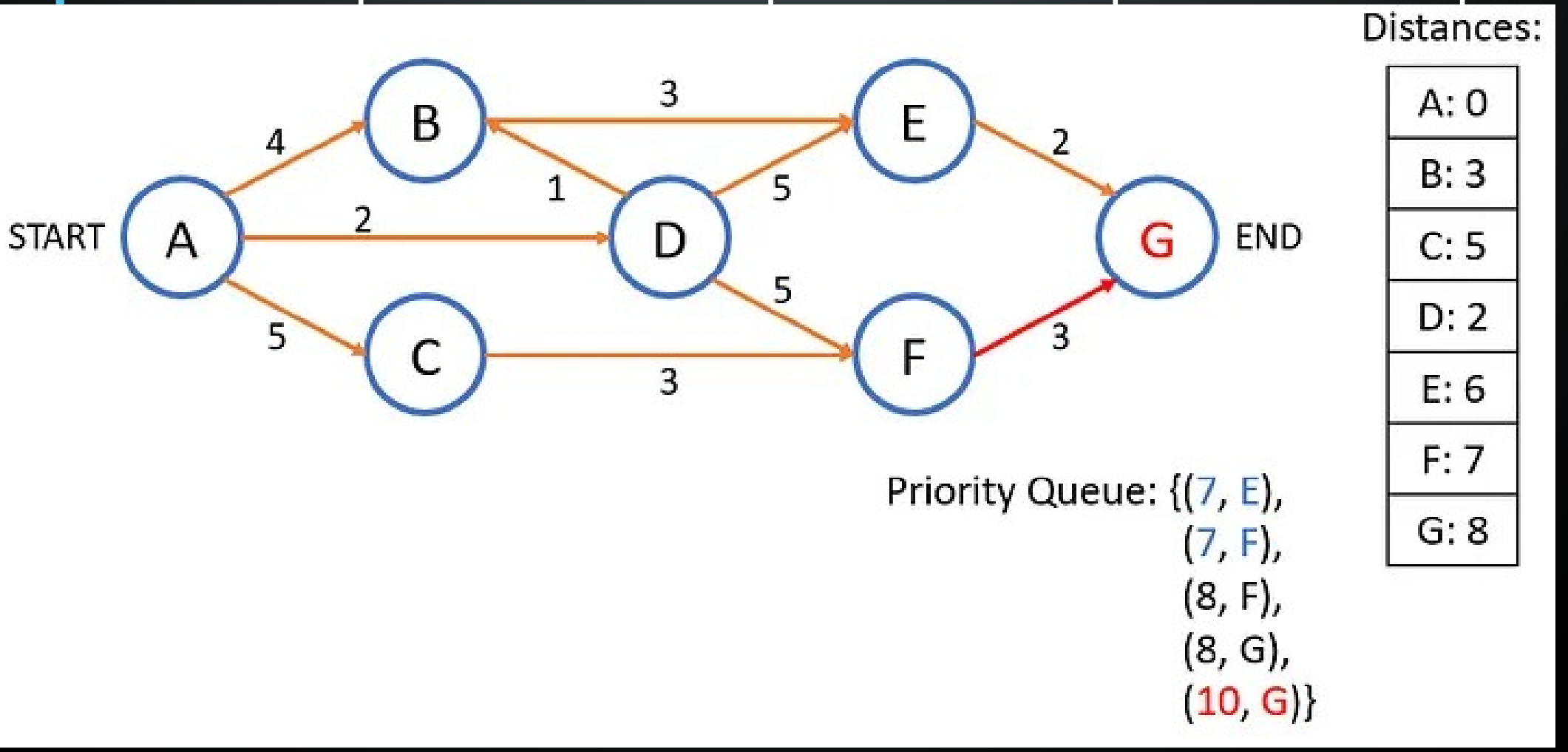
EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 6: Repeat the same process as described in previous st



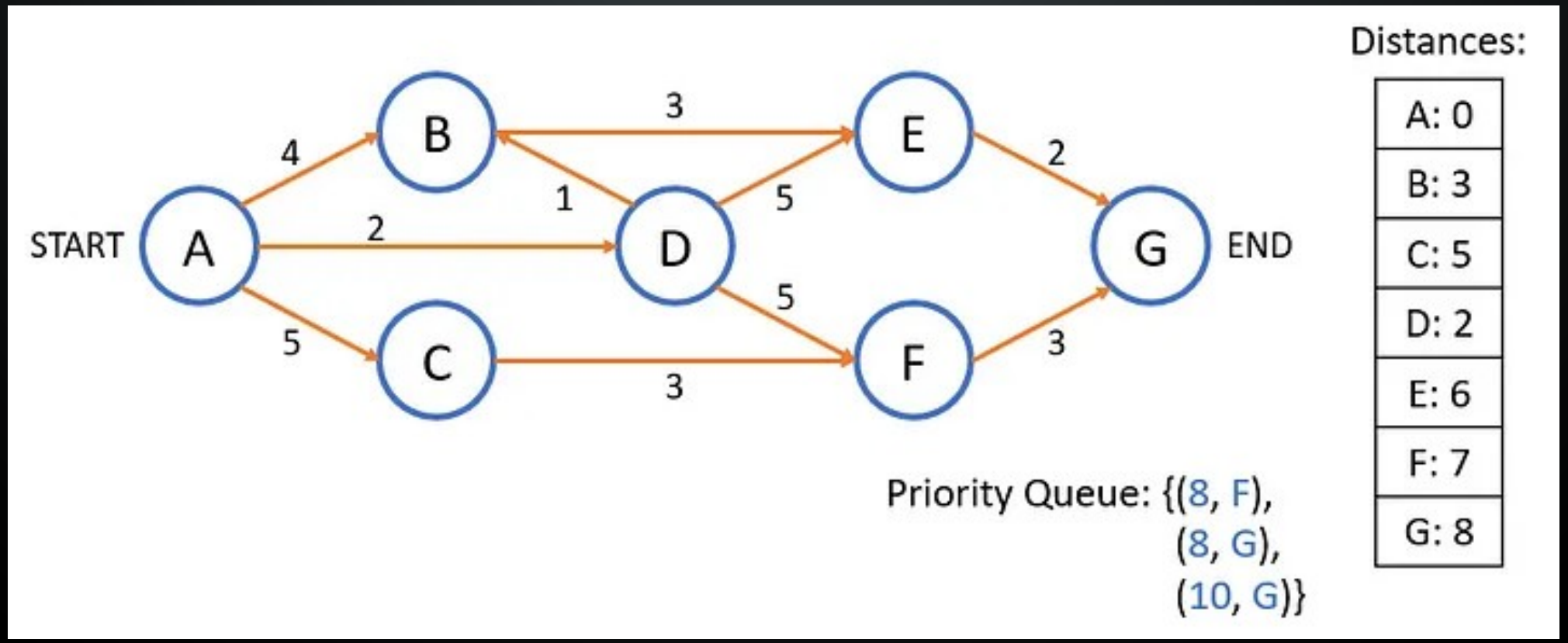
EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 7: This requires the same process as explained in step 5.



EXAMPLE OF DIJKSTRA'S ALGORITHM

Step 8: none of the tuples fit the condition of having a smaller distance value than those in distances. Therefore, they are all removed from the priority queue.



EXAMPLE OF DIJKSTRA'S ALGORITHM

This is it, there are no more tuples in the priority queue. We have just completed Dijkstra's Algorithm!

The shortest distance from the **START** node to all remaining nodes has been found. The shortest path from **START** to **END** has a distance of **8**.

To find the shortest path by nodes, **ADBEG**, an additional list must be created to store the temporary shortest paths.

DIJKSTRA'S TIME AND SPACE COMPLEXITY

There are multiple implementations of Dijkstra's algorithm, however, we will focus on the fastest and the most optimal implementation.

This is achieved by the use of priority queues (min-heap).

The min-heap stores the vertex along with its corresponding distance. During Dijkstra's algorithm, the vertex with the lowest distance is always found and explored. The best way of doing this is with a min-heap, which finds the minimum value in the heap. This requires an **O**(**log** **V**) time complexity, where **V** is the number of vertices in the graph.

DIJKSTRA'S TIME AND SPACE COMPLEXITY

In addition, throughout the algorithm, each edge is visited at most once while exploring vertices. This process requires a time complexity of **O**(**E**), where **E** is the number of edges in the graph.

When the minimum value vertex is extracted from the heap, the neighboring edges are then explored and their vertices are added to the heap. Therefore, combining these results yields an overall time complexity of **O**(**E log V**) for Dijkstra's algorithm.

To store the distances of each vertex, an additional space complexity of **O**(**V**) is required. Furthermore, the min-heap that stores the shortest distances during the worst-case will

BELLMAN-FORD ALGORITHM

Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs.

A Bellman-Ford algorithm is also guaranteed to find the shortest path in a graph, similar to Dijkstra’s algorithm. Although Bellman-Ford is slower than Dijkstra’s algorithm, it is capable of handling graphs with negative edge weights, which makes it more versatile.

BELLMAN-FORD ALGORITHM

The shortest path cannot be found if there exists a negative cycle in the graph. If we continue to go around the negative cycle an infinite number of times, then the cost of the path will continue to decrease (even though the length of the path is increasing). As a result, Bellman-Ford is also capable of detecting negative cycles, which is an important feature.

The Bellman-Ford algorithm’s primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on the “Principle of Relaxation“.

BELLMAN-FORD ALGORITHM

Principle of Relaxation of Edges for Bellman-Ford:

- It states that for the graph having **N** vertices, all the edges should be relaxed **N-1** times to compute the single source's shortest path.
- To detect whether a negative cycle exists or not, relax all the edges one more time, and if the shortest distance for any node reduces then we can say that a negative cycle exists. In short, if we relax the edges **N** times, and there is any change in the shortest distance of any node between the **N-1th** and **Nth** relaxation then a negative cycle exists, otherwise not exist.

BELLMAN-FORD ALGORITHM

Why Relaxing Edges N-1 times, gives us Single Source Shortest Path?

In the worst-case scenario, the shortest path between two vertices can have at most **N-1** edges, where **N** is the number of vertices. This is because a simple path in a graph with **N** vertices can have at most **N-1** edges, as it's impossible to form a closed loop without revisiting a vertex.

By relaxing edges **N-1** times, the Bellman-Ford algorithm ensures that the distance estimates for all vertices have been updated to their optimal values, assuming the graph doesn't contain any negative-weight cycles reachable from the source vertex. If a graph contains a negative-weight cycle reachable

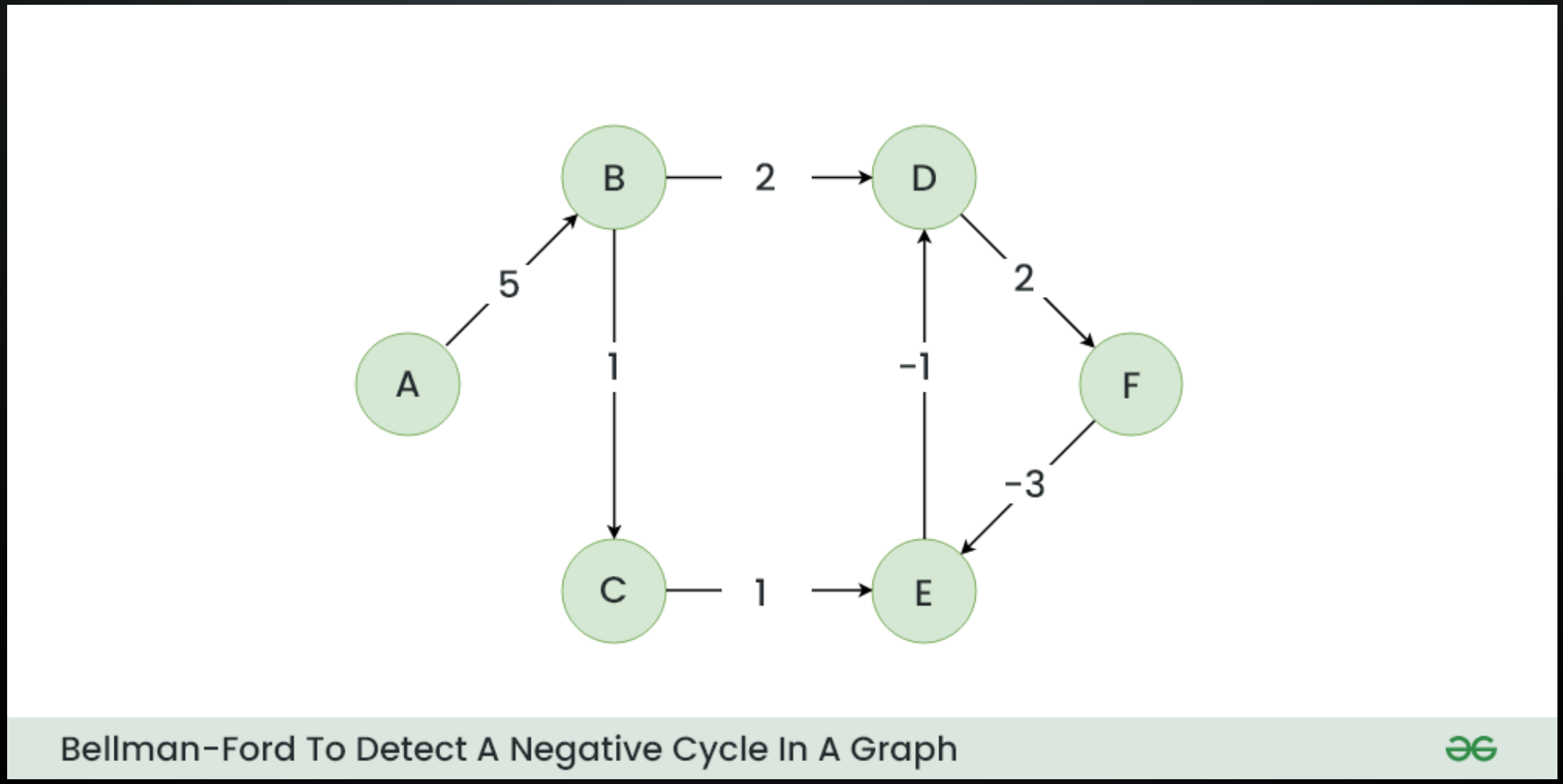
BELLMAN-FORD ALGORITHM

Why Does the Reduction of Distance in the Nth Relaxation Indicates the Existence of a Negative Cycle?

As previously discussed, achieving the single source shortest paths to all other nodes takes **N-1** relaxations. If the **Nth** relaxation further reduces the shortest distance for any node, it implies that a certain edge with negative weight has been traversed once more. It is important to note that during the **N-1** relaxations, we presumed that each vertex is traversed only once. However, the reduction of distance during the **Nth** relaxation indicates revisiting a vertex.

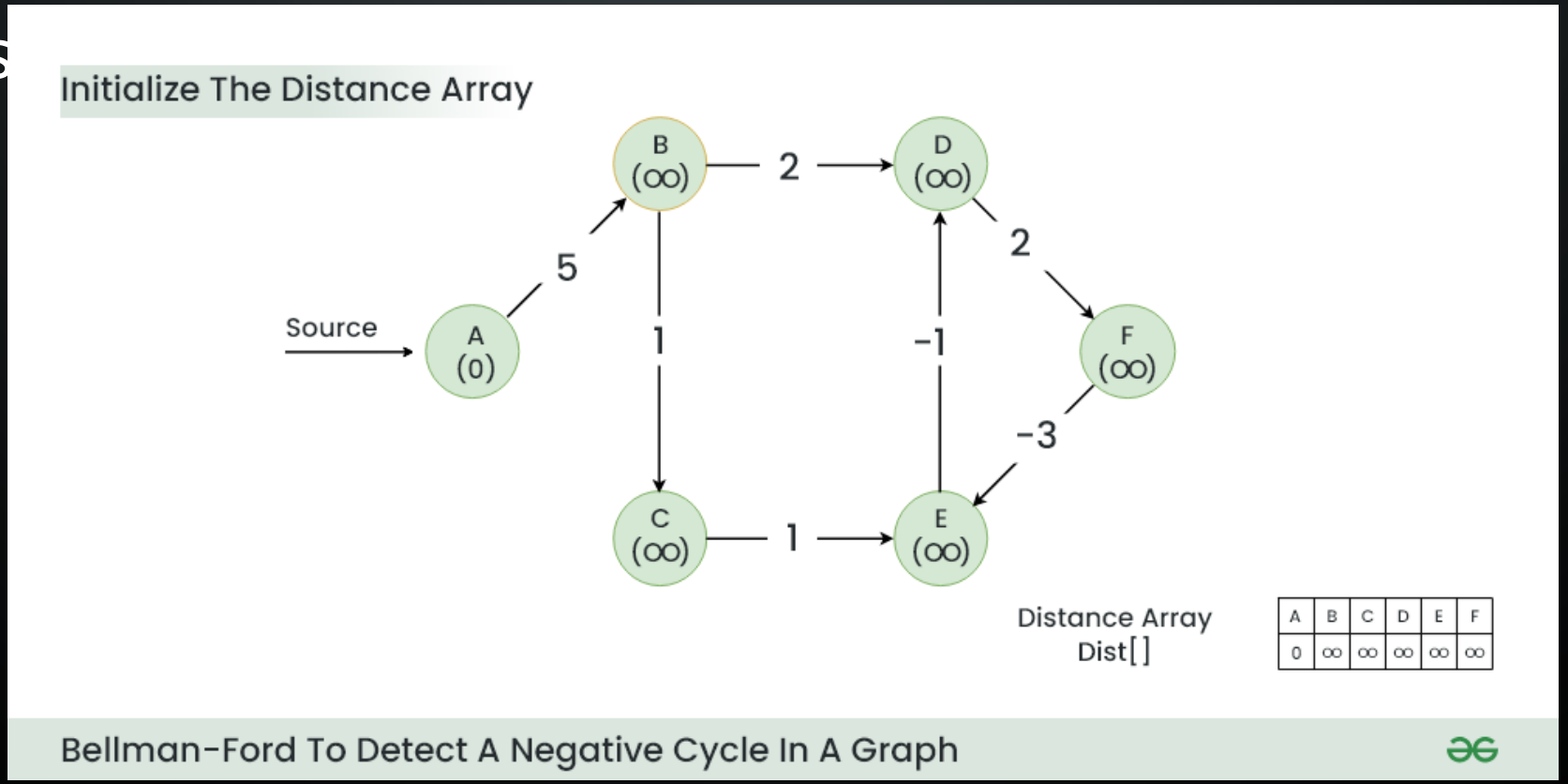
EXAMPLE OF BELLMAN-FORD ALGORITHM

Let’s suppose we have a graph which is given below.



EXAMPLE OF BELLMAN-FORD ALGORITHM

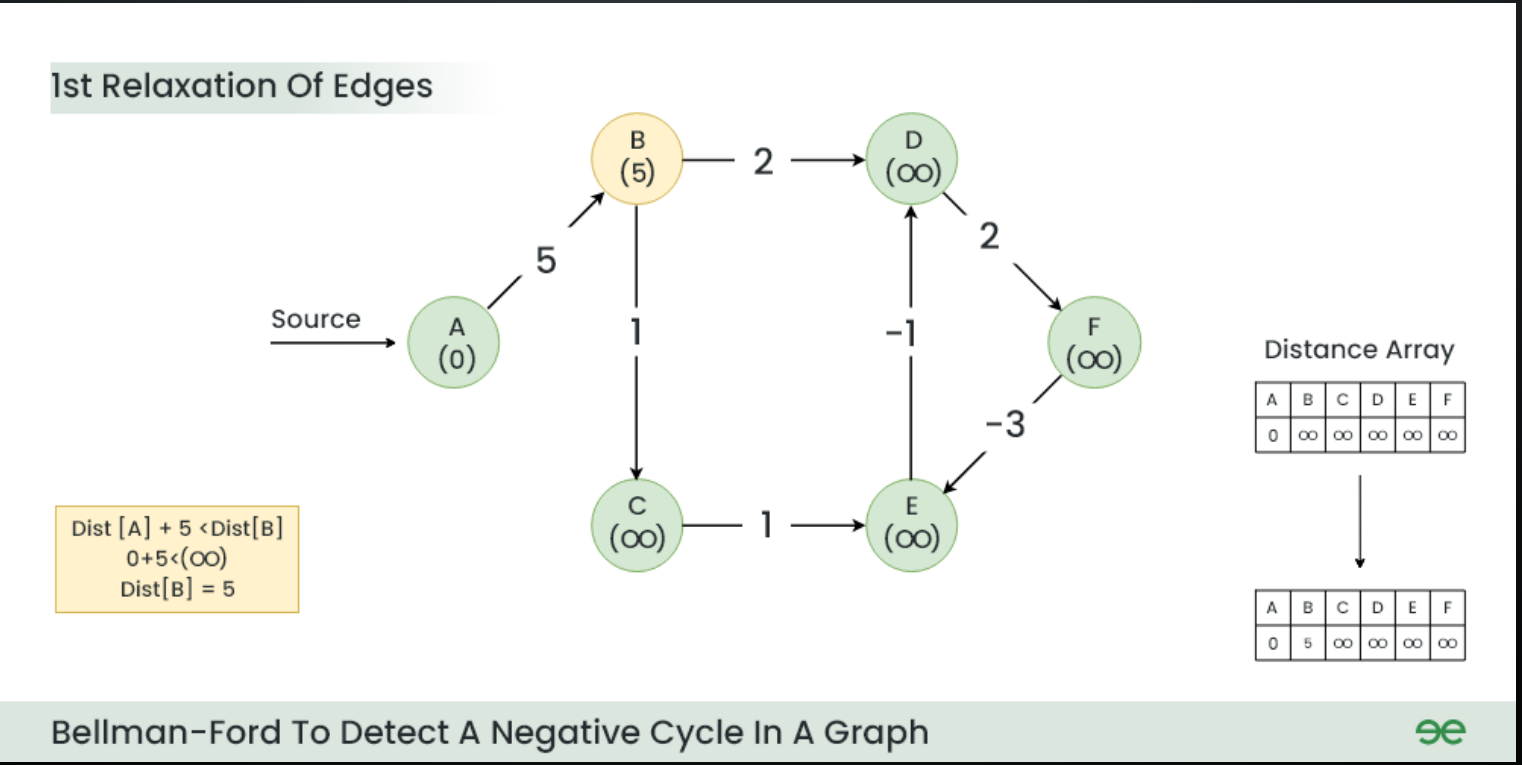
Step 1. Initialize a distance array Dist[] to store the shortest distance for each vertex from the source vertex. Initially distance of the source will be **0** and the Distance of other vertices



EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 2. Start relaxing the edges, during 1st Relaxation:

Current Distance of **B** > (Distance of **A**) + (Weight of **A** to **B**) i.e. **Infinity** > 0 + 5, therefore, Dist[**B**] = **5**



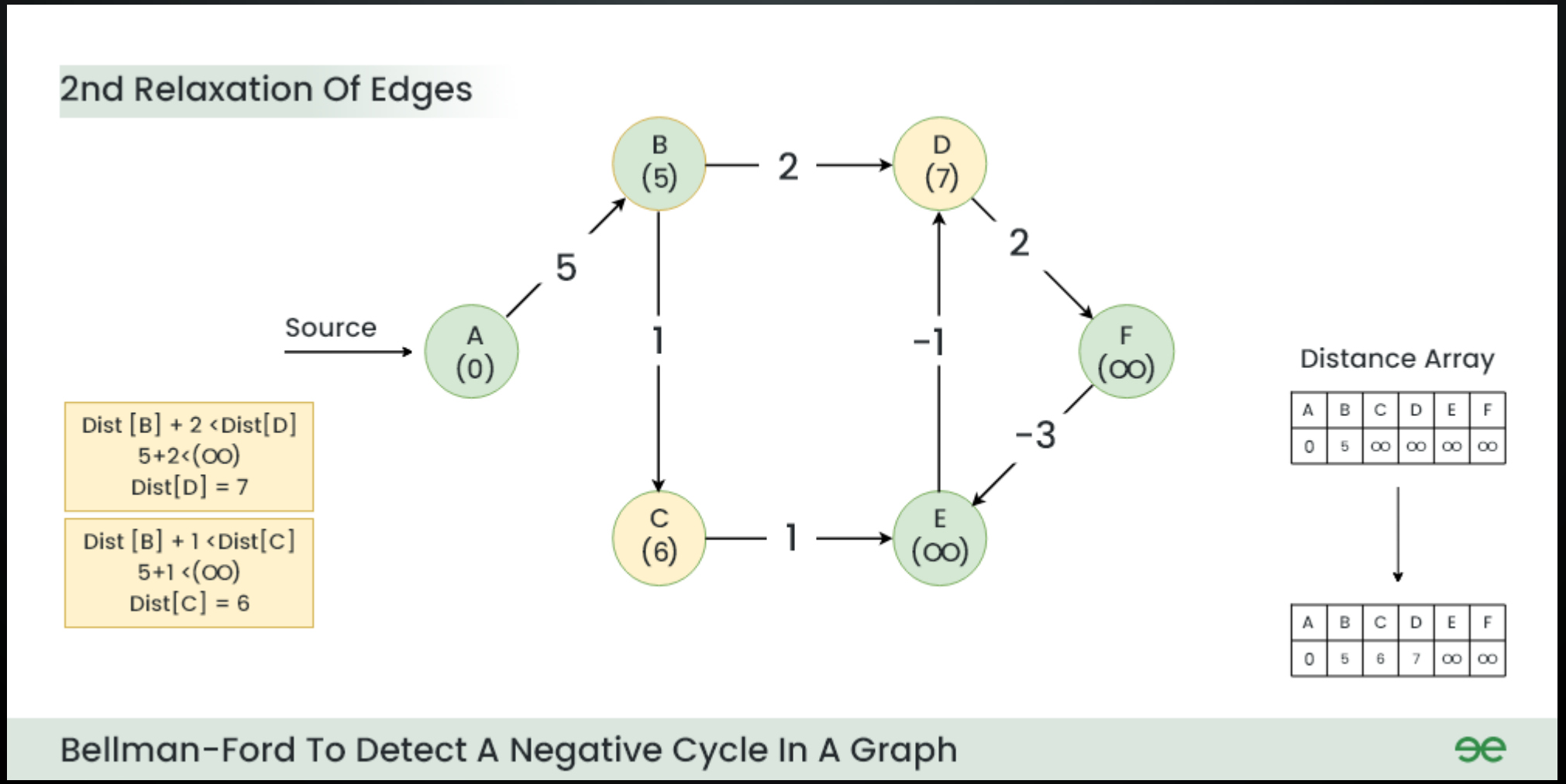
EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 3. During 2nd Relaxation:

- Current Distance of **D** > (Distance of **B**) + (Weight of **B** to **D**) i.e. **Infinity** > 5 + 2, Dist[**D**] = **7**
- Current Distance of **C** > (Distance of **B**) + (Weight of **B** to **C**) i.e. **Infinity** > 5 + 1, Dist[**C**] = **6**

EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 3. Visualization



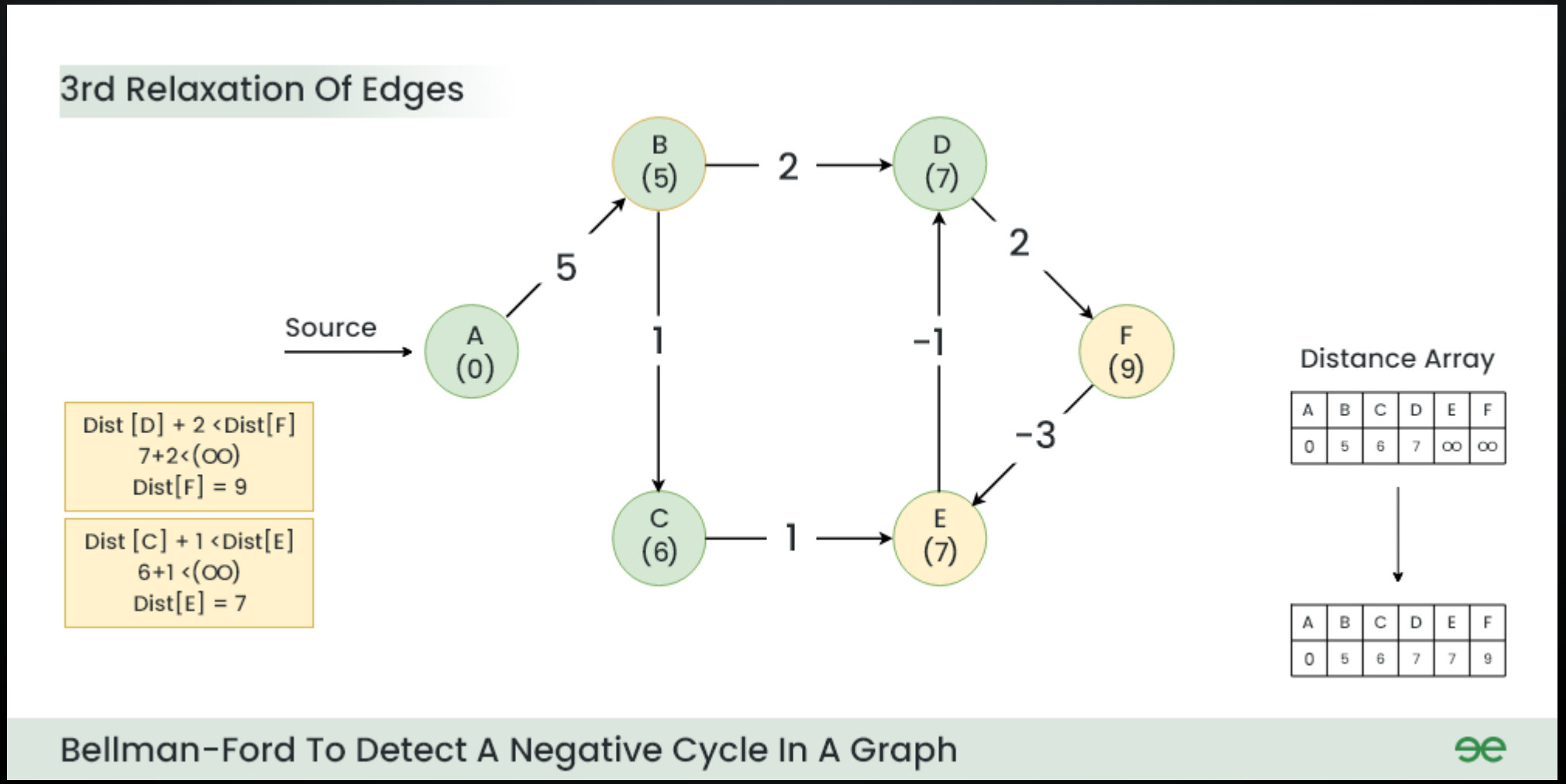
EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 4. During 3nd Relaxation:

- Current Distance of **F** > (Distance of **D**) + (Weight of **D** to **F**) i.e. **Infinity** > 7 + 2, Dist[**F**] = **9**
- Current Distance of **E** > (Distance of **C**) + (Weight of **C** to **E**) i.e. **Infinity** > 6 + 1, Dist[**E**] = **7**

EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 4. Visualization



EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 5. During 4th Relaxation:

- Current Distance of **D** > (Distance of **E**) + (Weight of **E** to **D**) i.e.

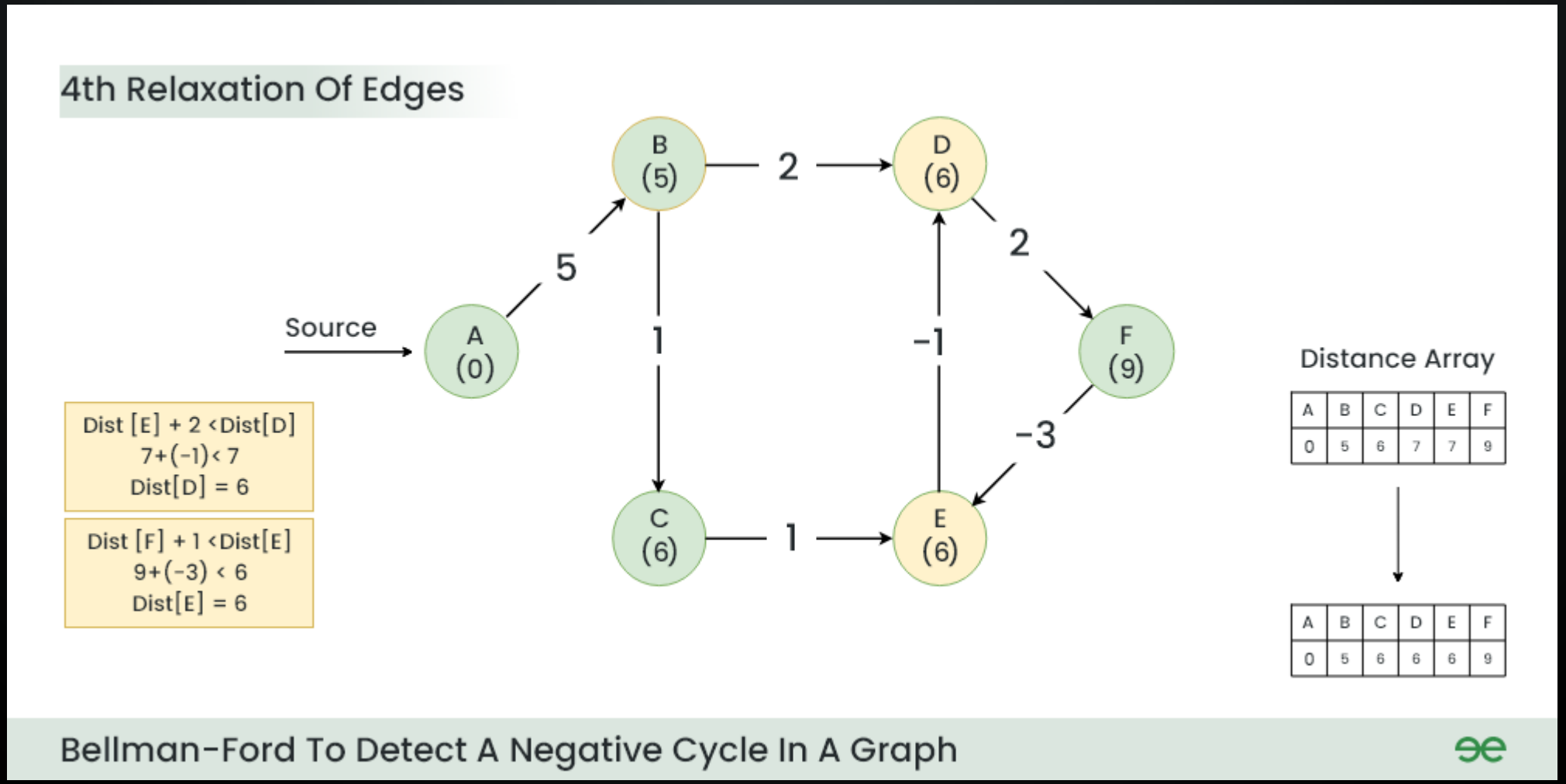
7 > 7 + (-1), Dist[**D**] = **6**

- Current Distance of **E** > (Distance of **F**) + (Weight of **F** to **E**) i.e.

7 > 9 + (-3), Dist[**E**] = **6**

EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 5. Visualization



EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 6. During 5th Relaxation:

- Current Distance of **F** > (Distance of **D**) + (Weight of **D** to **F**) i.e.

9 > 6 + 2, Dist[**F**] = **8**

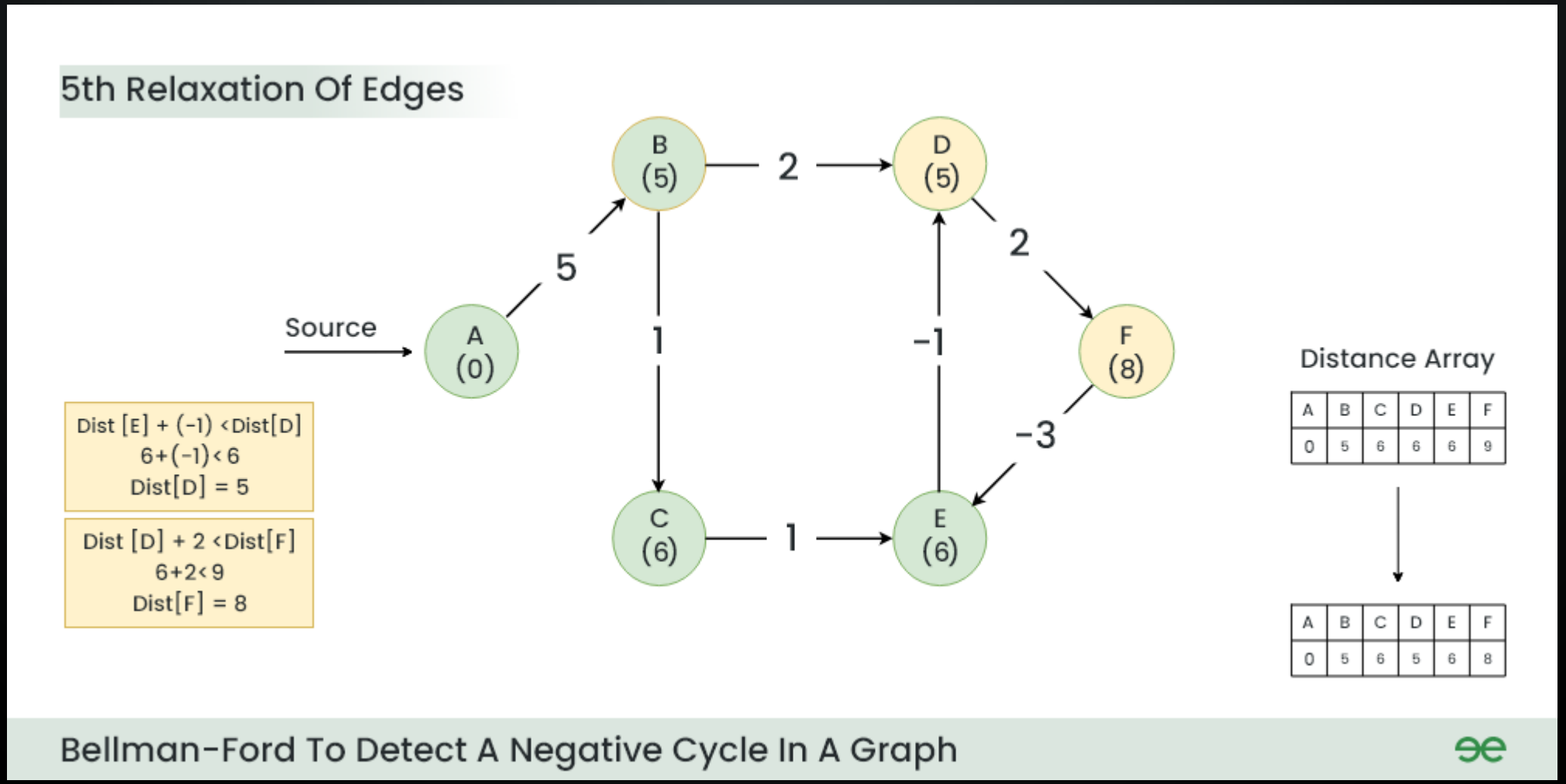
- Current Distance of **D** > (Distance of **E**) + (Weight of **E** to **D**) i.e.

6 > 6 + (-1), Dist[**D**] = **5**

- Since the graph has **6** vertices, So during the 5th relaxation the shortest distance for all the vertices should have been calculated.

EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 6. Visualization



EXAMPLE OF BELLMAN-FORD ALGORITHM

Step 7. Now the final relaxation i.e. the 6th relaxation should indicate the presence of a negative cycle if there are any changes in the distance array of the 5th relaxation.

During the 6th relaxation, the following changes can be seen:

- Current Distance of **E** > (Distance of **F**) + (Weight of **F** to **E**) i.e.

6 > 8 + (-3), Dist[**E**] = **5**

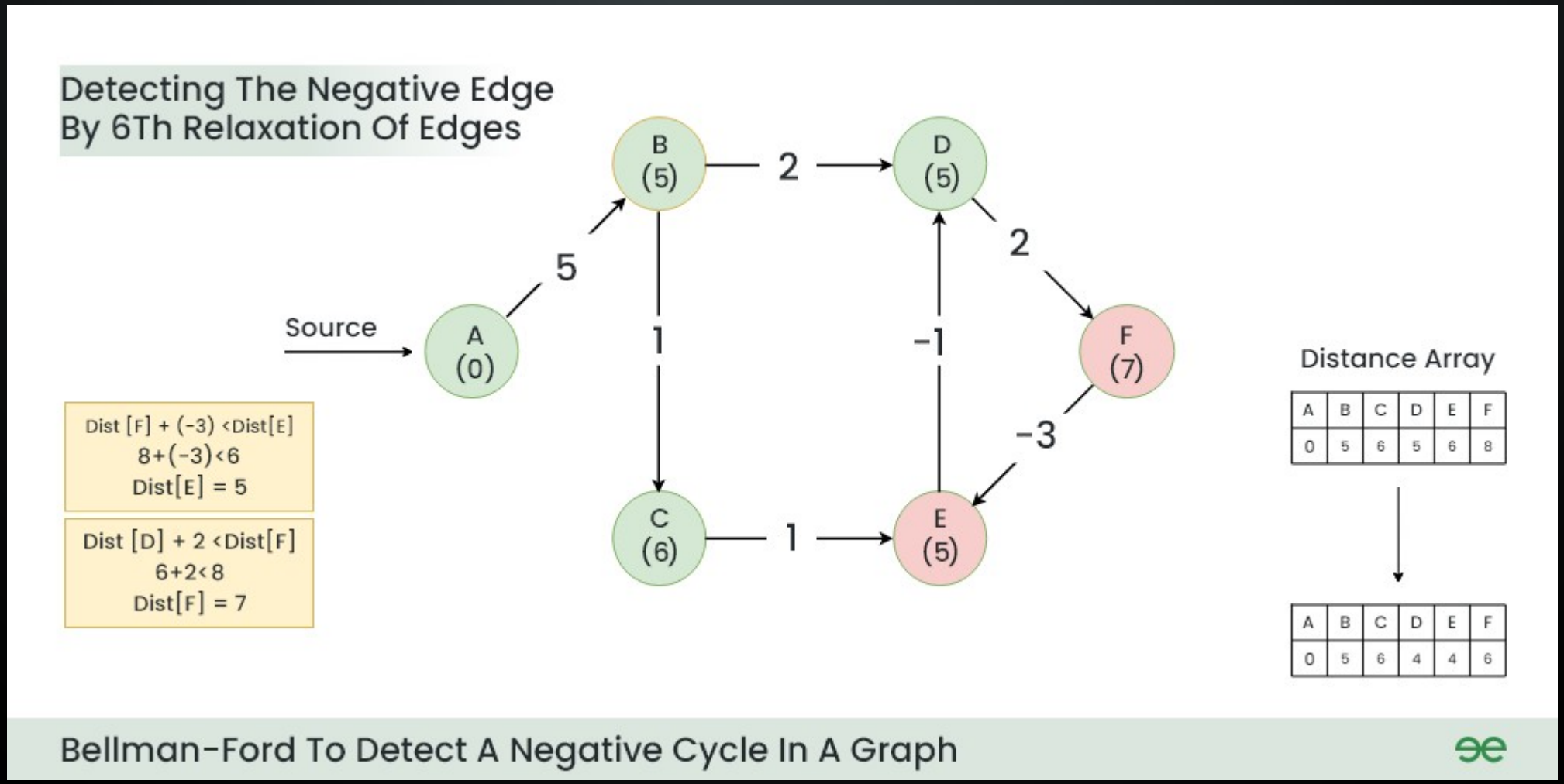
- Current Distance of **F** > (Distance of **D**) + (Weight of **D** to **F**) i.e.

8 > 5 + 2, Dist[**F**] = **7**

Since we observe changes in the Distance array hence, we can

EXAMPLE OF BELLMAN–FORD ALGORITHM

Step 7. A negative cycle (D->F->E) exists in the graph.



BELLMAN-FORD ALGORITHM PSEUDOCODE

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
  distance[S] <- 0
  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U
  for each edge (U,V) in G
    if distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists
  return distance[], previous[]
```

BELLMAN-FORD TIME AND SPACE COMPLEXITY

Time Complexity

Best Case Complexity **O(E)**

Average Case Complexity **O(V*E)**

Worst Case Complexity **O(V*E)**

Space Complexity is **O(V)**.

Bellman Ford vs Dijkstra

Bellman Ford's algorithm and Dijkstra's algorithm are very similar in structure. While Dijkstra looks only to the immediate neighbors of a vertex, Bellman goes through each edge in every iteration.

PRACTICE: SHORTEST PATH PROBLEM

Problem. You have an undirected graph with **N** vertices ($2 <= N <= 10^5$) and **M** edges ($2 <= M <= 10^5$). You need to find the shortest path from vertex **1** to vertex **N**.

Task. Create a program using C/C++/Python to solve this problem.

Input. The first string contains two numbers **N** and **M**. Next follow **M** strings each with three numbers: **u** ($1 <= u <= N$), **v** ($1 <= v <= N$) — vertices of the edge, and **w** ($1 <= w <= 10^7$) — weight of the edge.

Output. The single number — the sum of all edges weight in the shortest path.

PRACTICE: SHORTEST PATH PROBLEM

Code example:

```
#include <iostream>
#define MAX 100001
#define INF 0x7fffffff
typedef struct _node_t { int id; int cost; long way; bool vis; struct _node_t* ref; } node_t;
int N, M;
node_t Graph[MAX];
void add_node(int prev, int curr, int cost) { /*your code here*/ };
int main()
{
    cin >> N >> K;
    for(int i = 1; i <= N; i++) Graph[i].id = i;
    int f, s;
    for(int i = 0; i < M; i++)
    {
        cin >> u >> v >> w;
        add_node(u, v, w);
    }
    /*your code here*/
    cout << Graph[N].way;
    return 0;
}
```

THANK YOU!