Graph Theory

– Course 5 –

Chapter 5 : Shortest Path – Problem and Algorithms

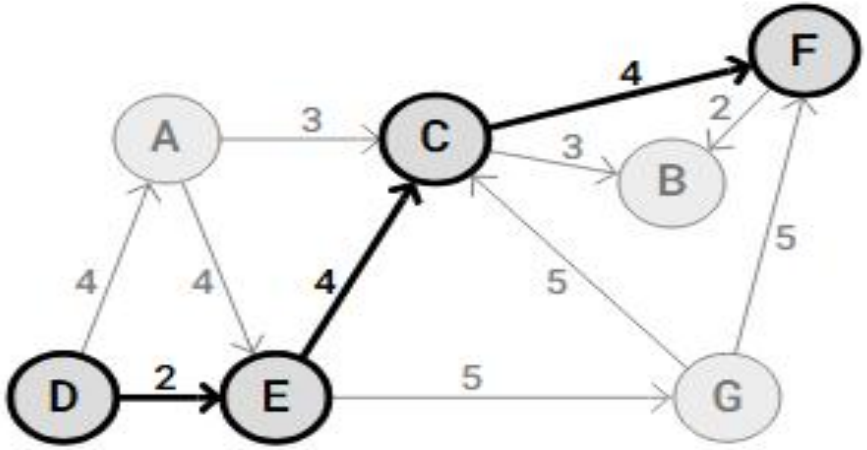
The shortest path problem (1/2)

The shortest path problem is among the most studied problems in graph theory; it is found in many fields:

- Economics: investment problems
- Management: project management
- Network optimization (road networks, telecommunications networks, distribution networks)
- Computer networks and routing protocols (OSPF protocol: **O**pen **S**hortest **P**ath **F**irst)

The shortest path problem (2/2)

- In the shortest path problem, a Graph can represent anything from a road network to a communication network, where the vertices can be intersections, cities, or routers, and the edges can be roads, flight paths, or data links.



The shortest path from vertex D to vertex F in the Graph above is D->E->C->F, with a total path weight of 2+4+4=10. Other paths from D to F are also possible, but they have a higher total weight, so they can not be considered to be the shortest path.

Definition

- Given a graph $G(V, E)$, each edge $e \in E$ is associated with a number $\ell(e)$ called the "length of the edge".
- The shortest path between two vertices u and v consists of finding an elementary path $r(u, v)$ from u to v whose total length $\ell(r)$ is minimal, where $\ell(r) = \sum_{e \in r} \ell(e)$.
- The length ℓ can also represent a **transportation cost**, a **construction cost**, a **time duration**, etc.

Solutions to The Shortest Path Problem (1/3)

- To solve the shortest path problem means to check the edges inside the Graph until we find a path where we can move from one vertex to another using the lowest possible combined weight along the edges.

- This sum of weights along the edges that make up a path is called a path cost or a path weight.

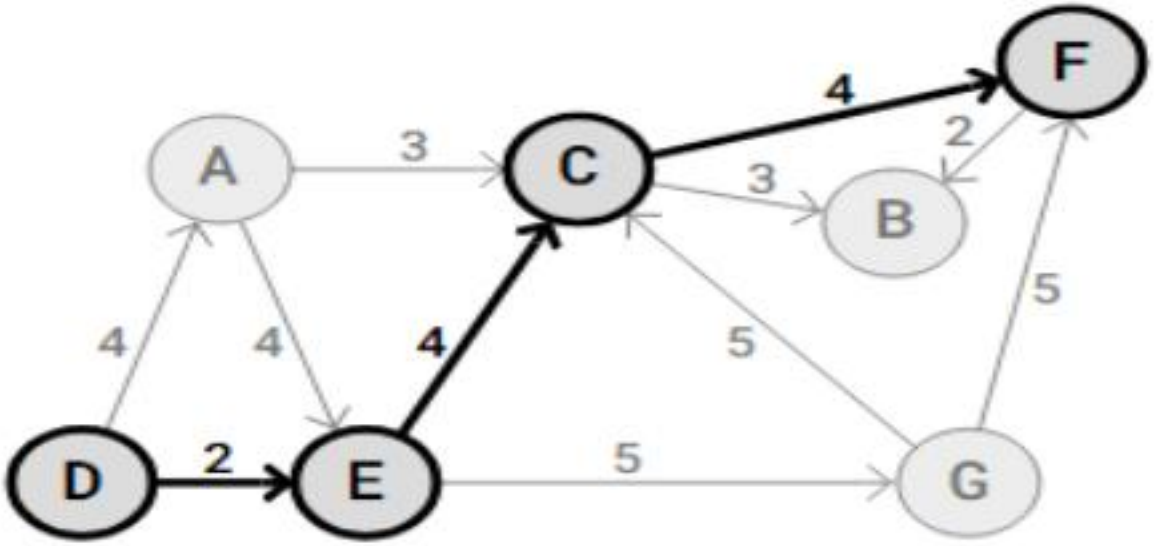
Solutions to The Shortest Path Problem (2/3)

- Algorithms that find the shortest paths, like **Dijkstra's algorithm** or the **Bellman-Ford algorithm**, find the shortest paths from one start vertex to all other vertices.

- To begin with, the algorithms set the distance from the start vertex to all vertices to be infinitely long. And as the algorithms run, edges between the vertices are checked over and over, and shorter paths might be found many times until the shortest paths are found at the end.

Solutions to The Shortest Path Problem (3/3)

- Every time an edge is checked and it leads to a shorter distance to a vertex being found and updated, it is called a **relaxation**, or **relaxing an edge**.

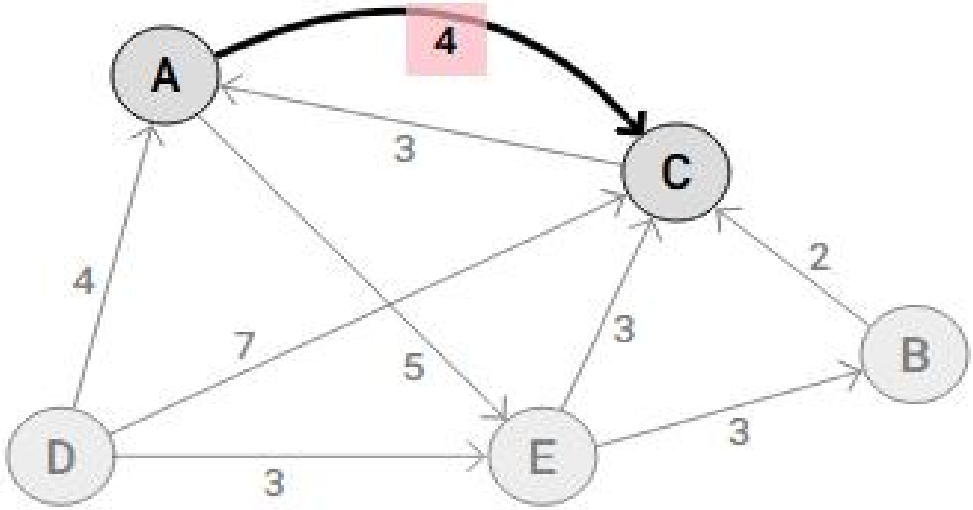


Positive and Negative Edge Weights (1/4)

- Some algorithms that find the shortest paths, like Dijkstra's algorithm, can only find the shortest paths in graphs where all the edges are positive. Such graphs with positive distances are also the easiest to understand because we can think of the edges between vertices as distances between locations.

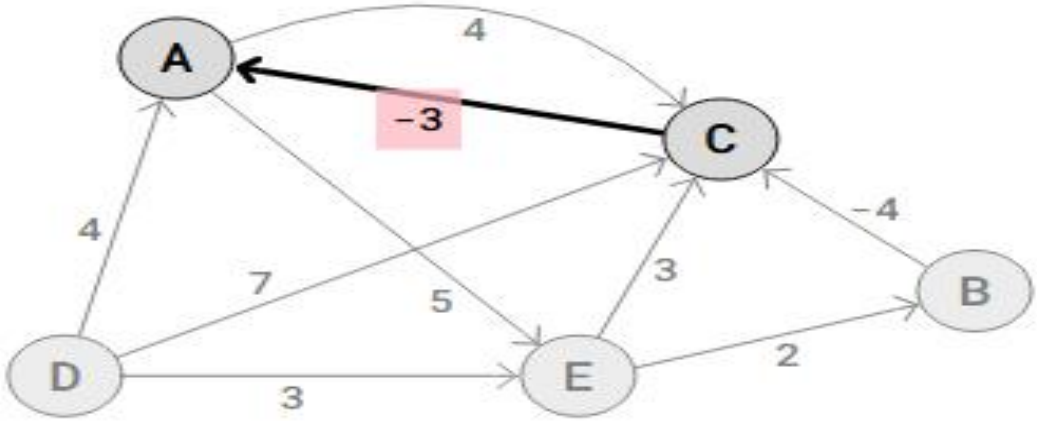
Positive and Negative Edge Weights (2/4)

- If we interpret the edge weights as money lost by going from one vertex to another, a positive edge weight of 4 from vertex A to C in the graph above means that we must spend $4 to go from A to C.



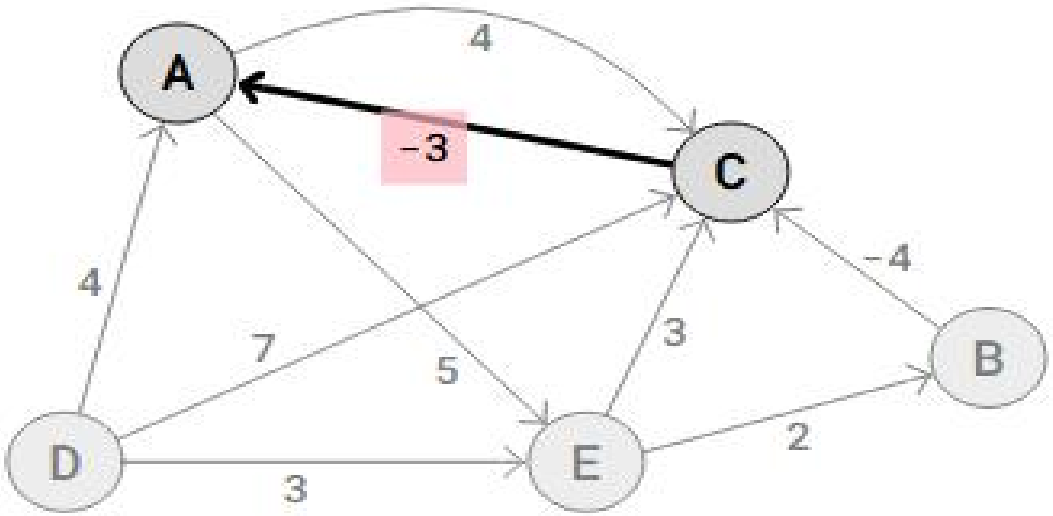
Positive and Negative Edge Weights (3/4)

- The negative edge weight **-3** from vertex **C** to **A** in the graph below can be understood as an edge where there is more money to be made than money lost by going from **C** to **A**. So if for example the cost of fuel is **$5** going from **C** to **A**, and we get paid **$8** for picking up packages in **C** and delivering them in **A**, money lost is **-3**, meaning we are actually earning **$3** in total.



Positive and Negative Edge Weights (4/4)

- So, graphs can also have negative edges, and for such graphs the **Bellman-Ford algorithm** can be used to find the shortest paths.



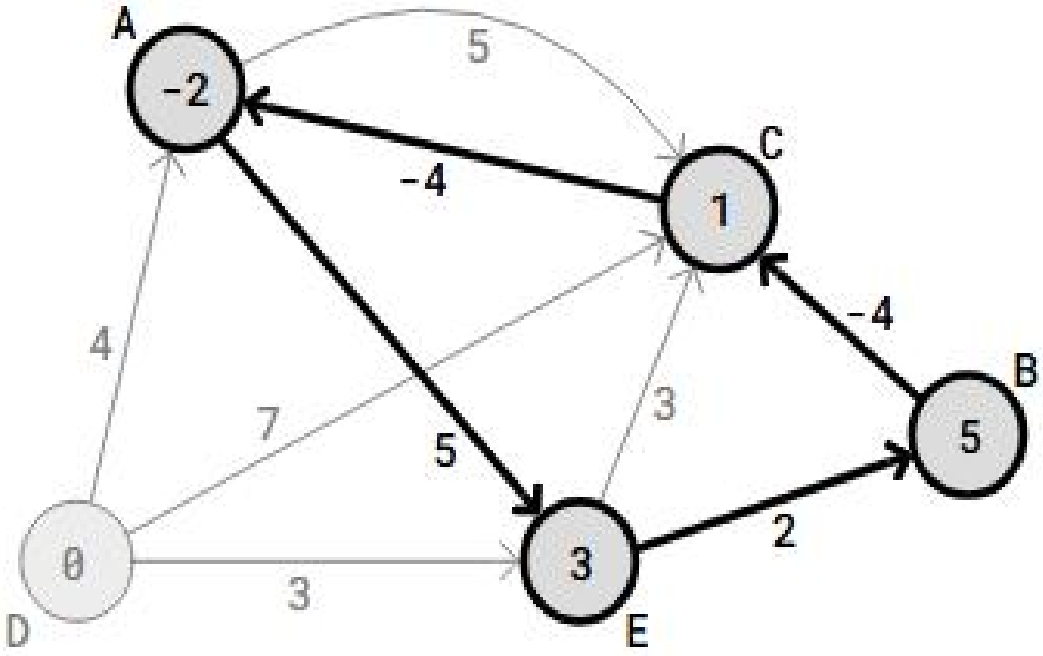
Negative Cycles in Shortest Path Problems (1/3)

- Finding the shortest paths becomes **impossible** if a graph has negative cycles.

- Having a negative cycle means that there is a path where you can go in circles, and the edges that make up this circle have a total path weight that is negative.

Negative Cycles in Shortest Path Problems (2/3)

- In the graph below, the path A->E->B->C->A is a negative cycle because the total path weight is 5+2-4-4=-1.



Negative Cycles in Shortest Path Problems (3/3)

The reason why it is impossible to find the shortest paths in a graph with negative cycles is that it will always be possible to continue running an algorithm to find even shorter paths.

Let's say for example that we are looking for the shortest distance from vertex D in graph above, to all other vertices. At first we find the distance from D to E to be 3, by just walking the edge D->E. But after this, if we walk one round in the negative cycle E->B->C->A->E, then the distance to E becomes 2. After walking one more round the distance becomes 1, which is even shorter, and so on. We can always walk one more round in the negative cycle to find a shorter distance to E, which means the shortest distance can never be found.

Dijkstra's Algorithm (1/7)

- Dijkstra's shortest path algorithm was invented in 1956 by the Dutch computer scientist Edsger W. Dijkstra during a twenty minutes coffee break.

- The reason for inventing the algorithm was to test a new computer called ARMAC.

Dijkstra's Algorithm (2/7)

- Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.

- It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.

Dijkstra's Algorithm (3/7)

- Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.
- It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.
- Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.

Dijkstra's Algorithm (4/7)

- Dijkstra's algorithm is used for solving single-source shortest path problems for directed or undirected paths. Single-source means that one vertex is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices.
- Dijkstra's algorithm does not work for graphs with negative edges. For graphs with **negative edges**, the **Bellman-Ford algorithm** can be used instead.

Dijkstra's Algorithm (5/7)

- To find the **shortest path**, **Dijkstra's algorithm** needs to know which vertex is the source, it needs a way to mark vertices as visited, and it needs an overview of the current shortest distance to each vertex as it works its way through the graph, updating these distances when a shorter distance is found.

Dijkstra's Algorithm (6/7)

1. Set initial distances for all vertices: **0** for the **source vertex**, and **infinity** for **all the other**.

2. Choose the unvisited vertex with the **shortest distance from the start to be the current vertex**. So the algorithm will always start with the source as the current vertex.

3. For each of the **current vertex's unvisited neighbor vertices**, calculate the distance from the **source** and update the distance if the new, calculated, distance is lower.

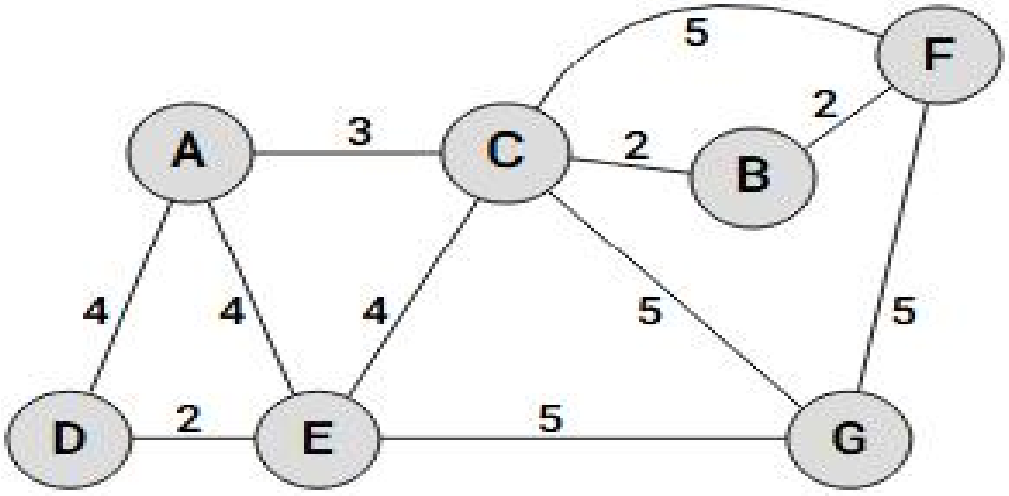
4. We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.

5. Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.

6. In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

Dijkstra's Algorithm (7/7)

- The basic version of Dijkstra's algorithm gives us the **value of the shortest path cost** to every vertex, but not what the actual path is.
- Illustrative Example: Consider the Graph below.

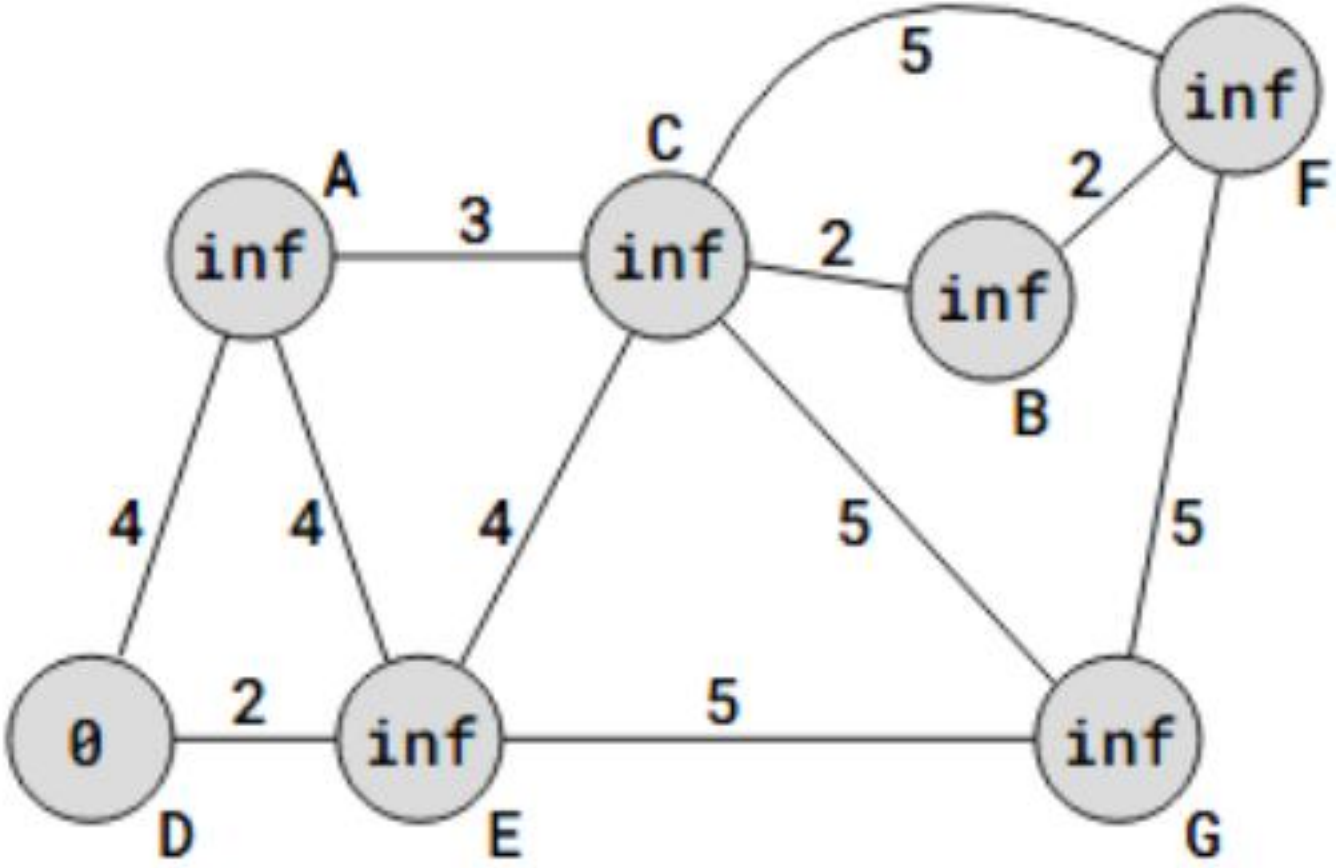


Illustrative Example (1/11)

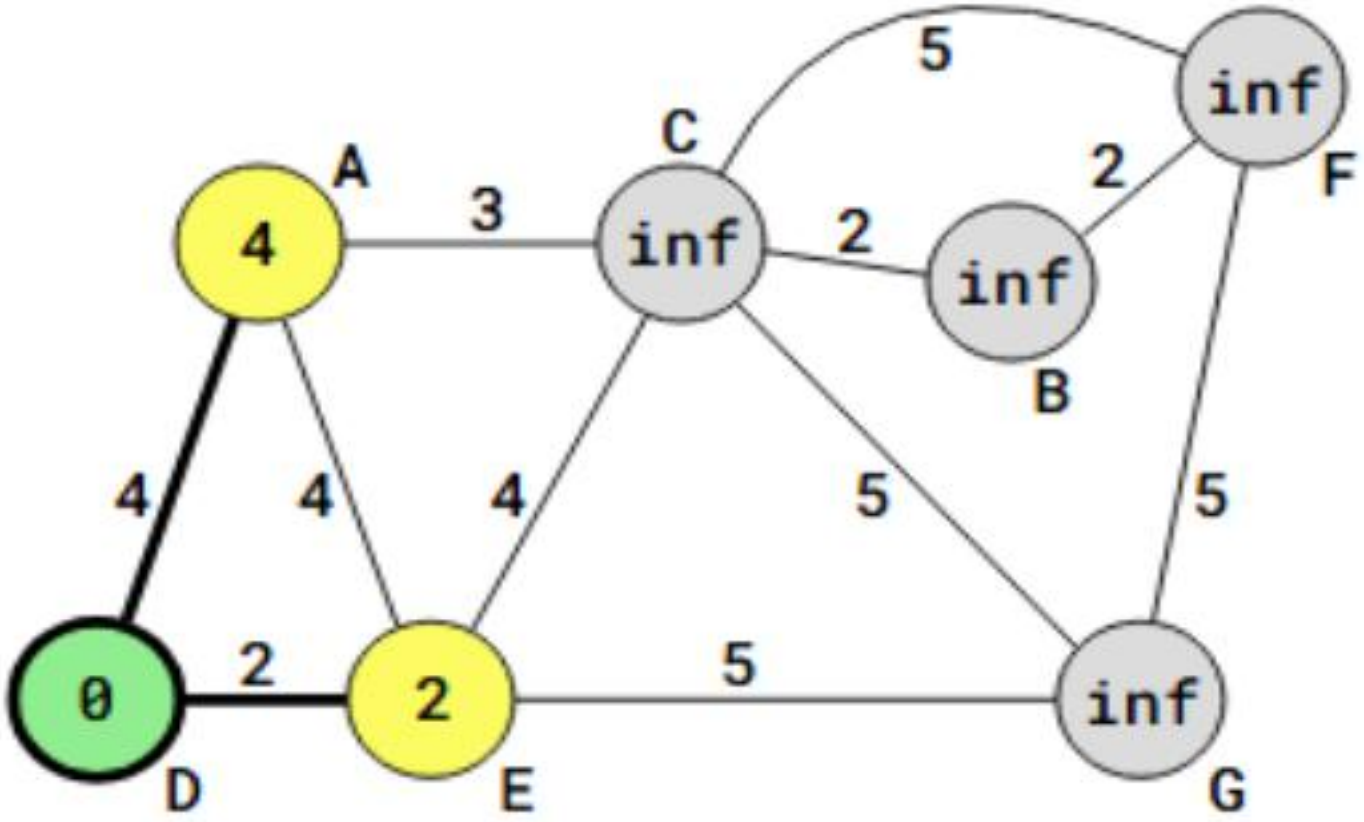
- We want to find the **shortest path** from the source vertex **D** to all other vertices, so that for example the shortest path to **C** is **D->E->C**, with **path weight 2+4=6.**
- To find the shortest path, Dijkstra's algorithm uses an array with the distances to all other vertices, and initially sets these distances to **infinite**, or a **very big number**. And the distance to the vertex we start from (the source) is set to **0**.

```
distances = [inf, inf, inf, 0, inf, inf, inf]
#vertices   [ A ,  B ,  C , D,  E ,  F ,  G ]
```

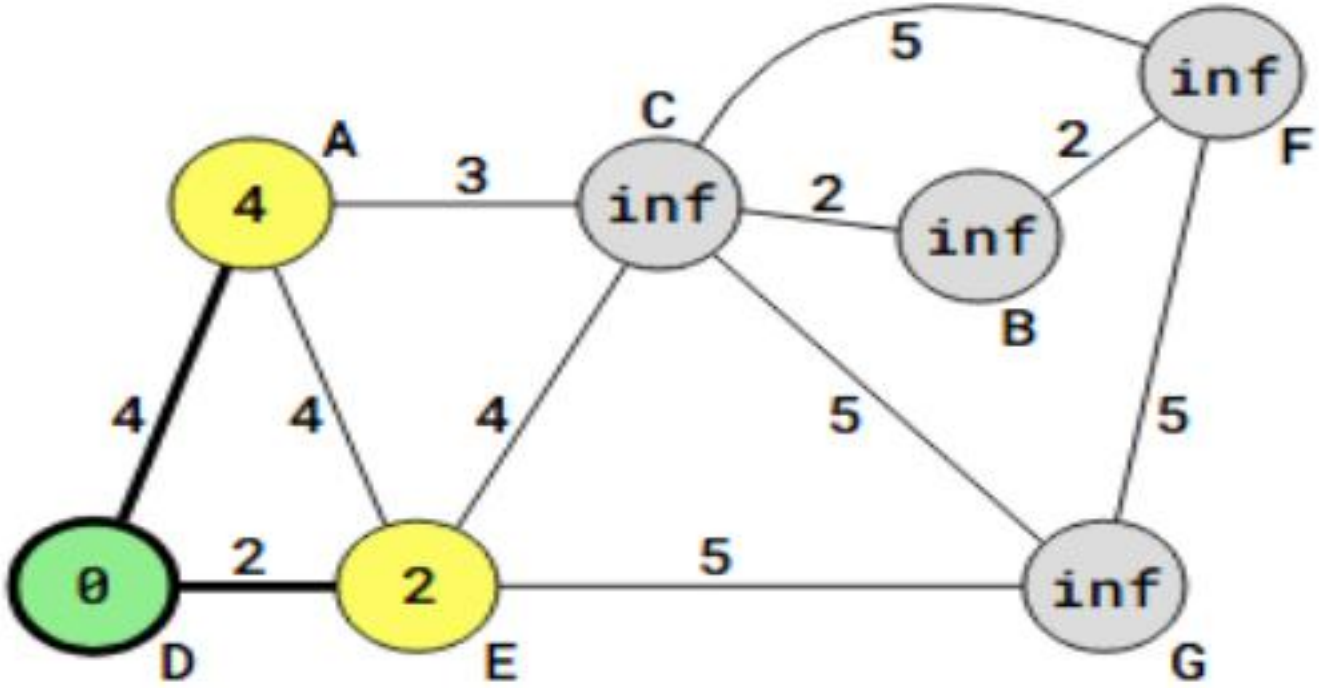
Illustrative Example (2/11)



Illustrative Example (3/11)



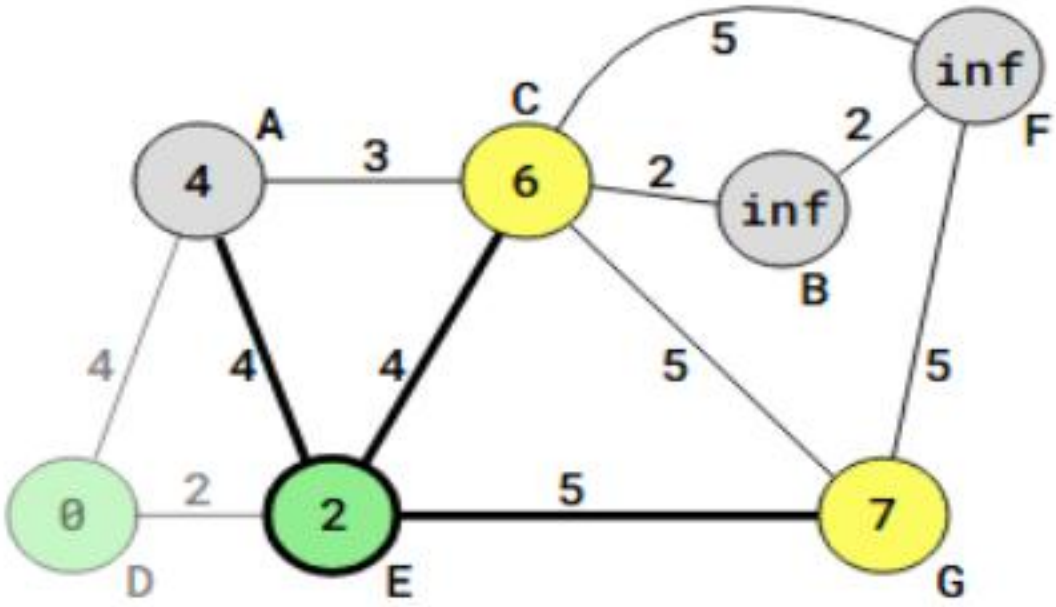
Illustrative Example (4/11)



- Updating the distance values in this way is called '**relaxing**'.
- After **relaxing vertices A and E**, vertex D is considered visited, and will not be visited again.

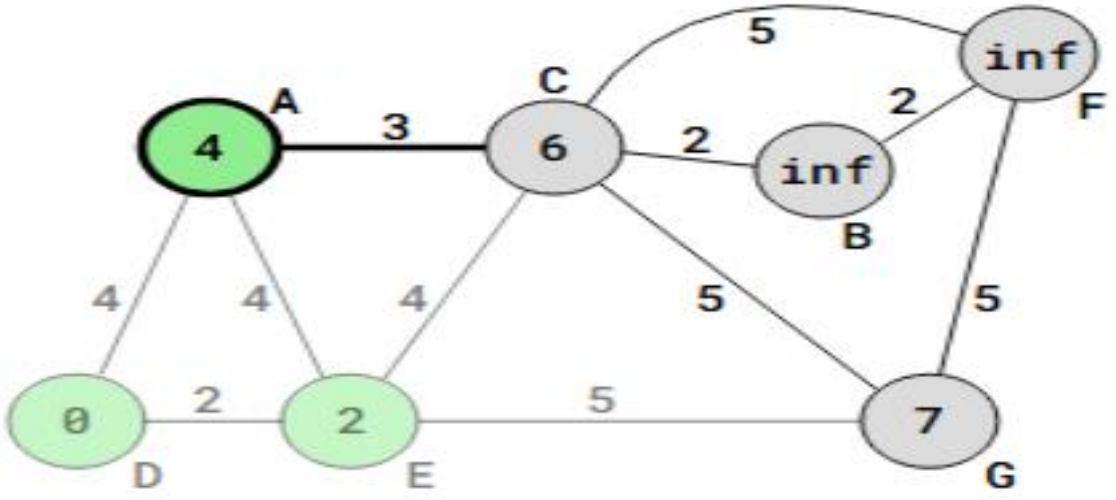
Illustrative Example (5/11)

- The **next vertex to be chosen** as the current vertex must the vertex with the **shortest distance to the source vertex (vertex D)**, among the previously unvisited vertices. **Vertex E is therefore chosen as the current vertex after vertex D**.

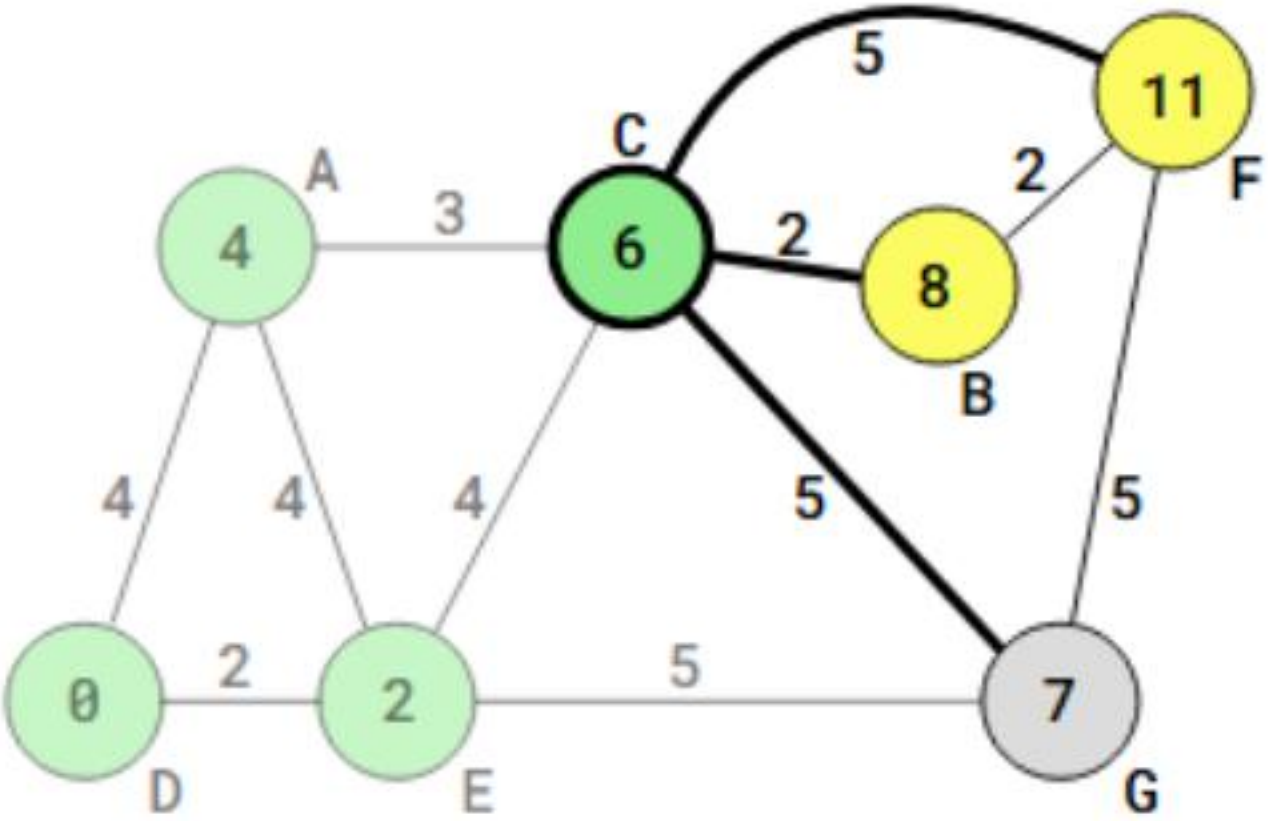


Illustrative Example (6/11)

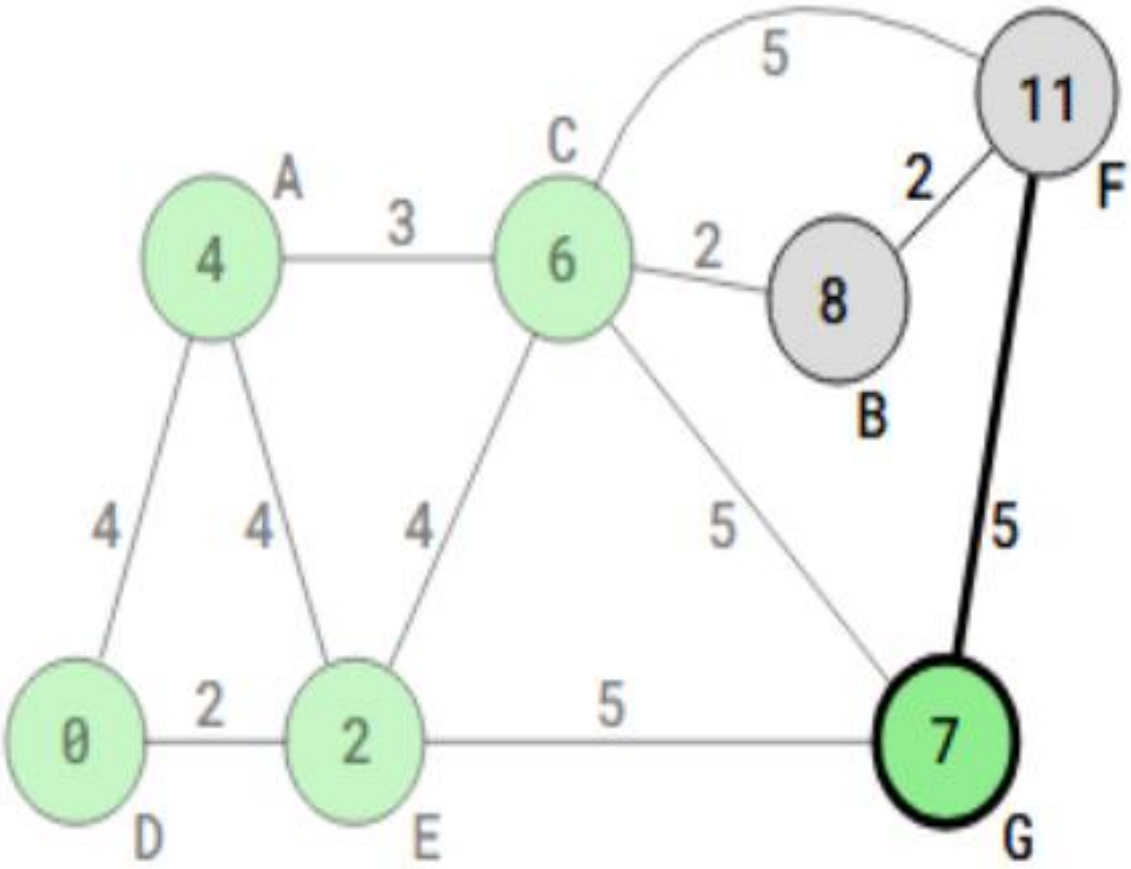
- The distance to all adjacent and not previously visited vertices from vertex **E** must now be calculated, and updated if needed.
- For example, the calculated distance from **D** to vertex **A**, via **E**, is **2+4=6**. But the current distance to vertex **A** is already **4**, which is lower, so the distance to vertex **A** is not updated.



Illustrative Example (7/11)

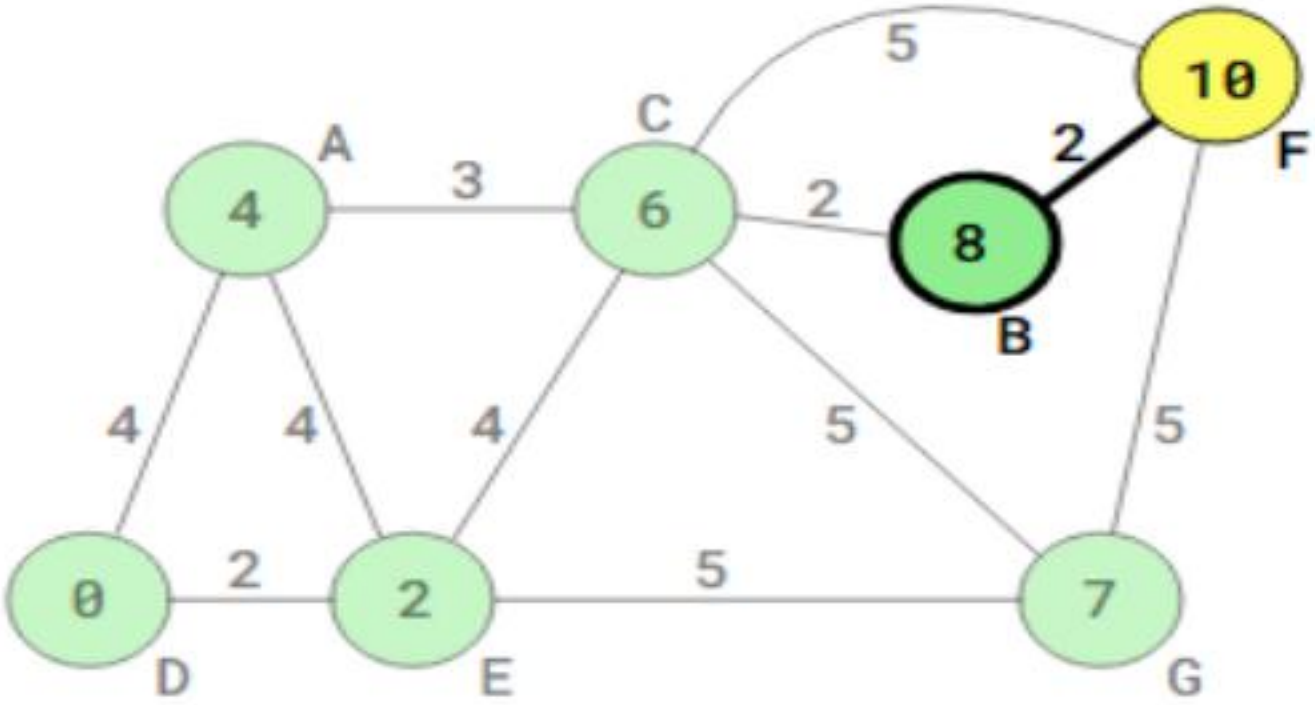


Illustrative Example (8/11)



Illustrative Example (9/11)

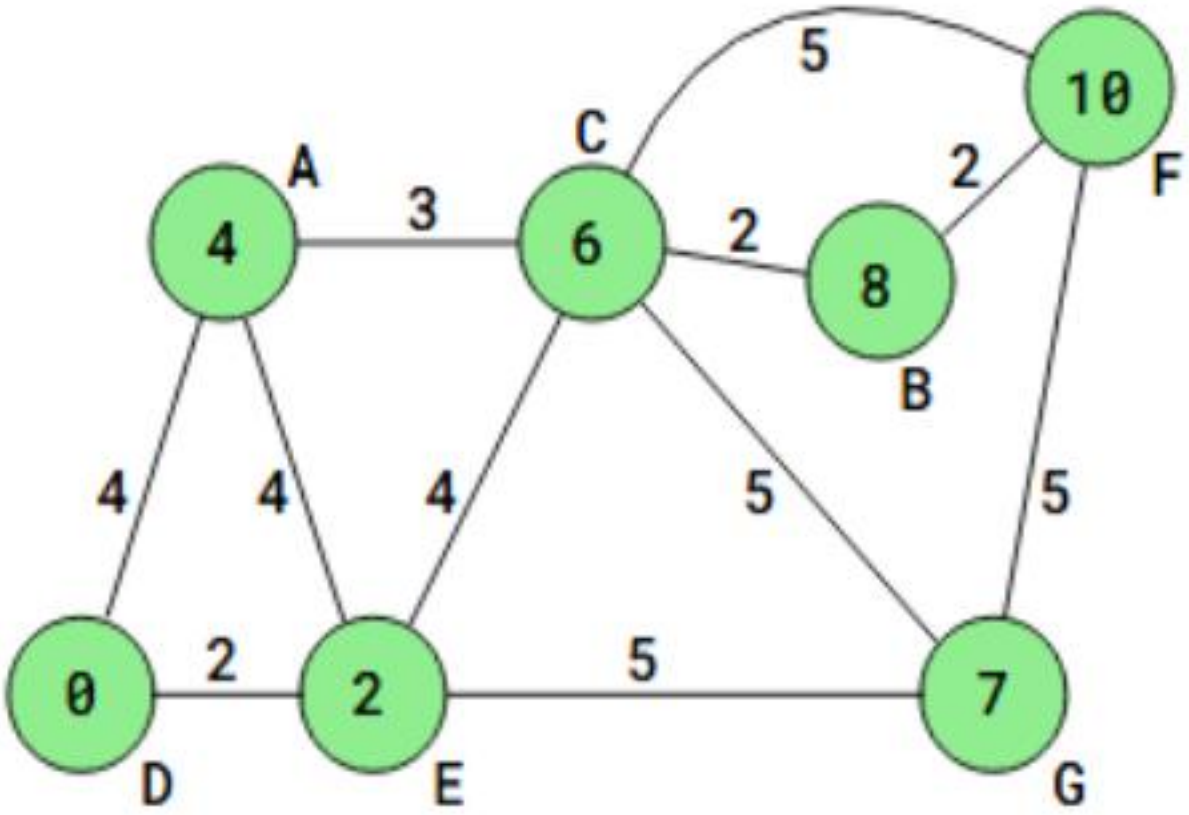
- Vertex G is marked as visited, and **B** becomes the current vertex because it has the lowest distance of the remaining unvisited vertices.



Illustrative Example (10/11)

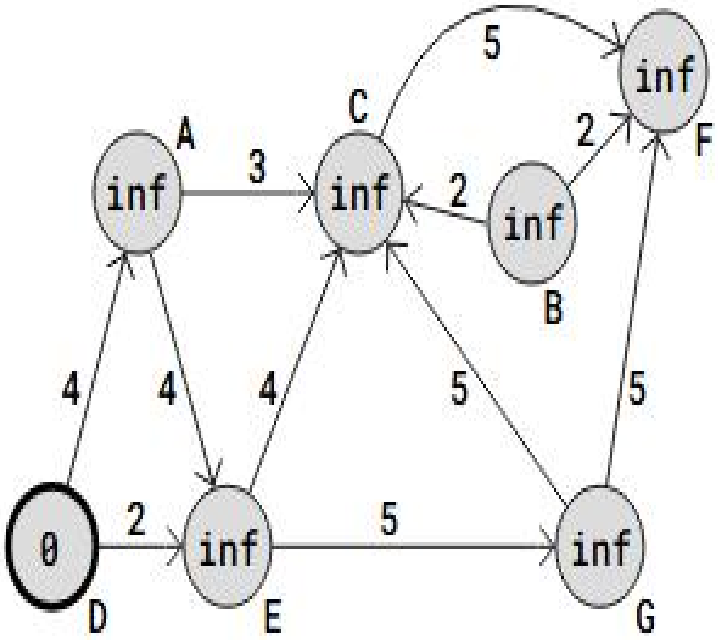
- The new distance to **F** via **B** is 8+2=10, because it is lower than F's existing distance of 11.
- Vertex **B** is marked as visited, and there is nothing to check for the last unvisited vertex **F**, so Dijkstra's algorithm is finished.
- Every vertex has been visited only once, and the result is the lowest distance from the source vertex **D** to every other vertex in the graph.

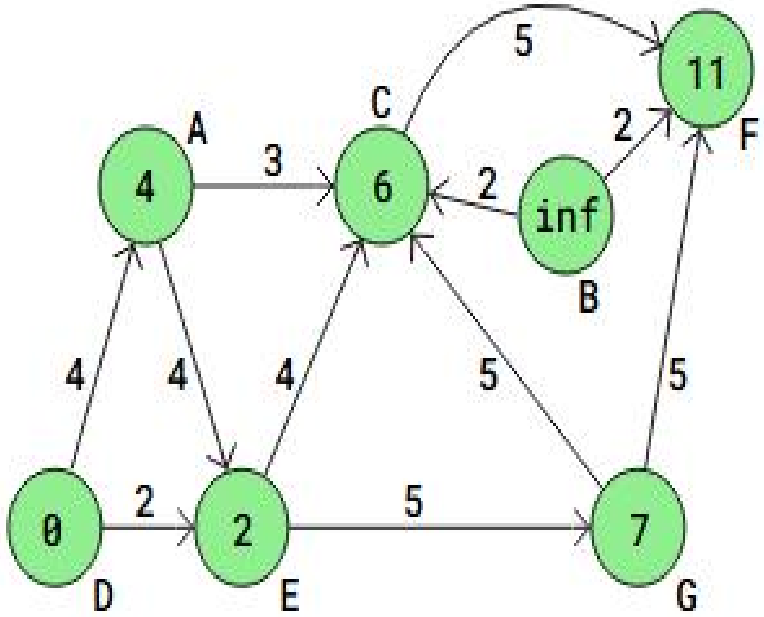
Illustrative Example (11/11)



Dijkstra's Algorithm on Directed Graphs (1/2)

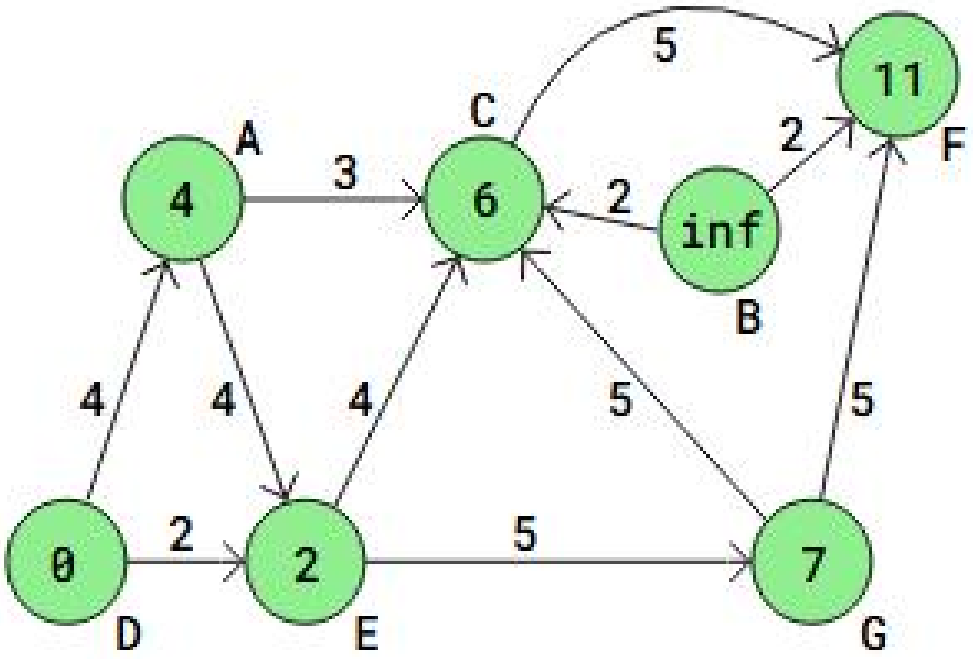
- To run Dijkstra's algorithm on directed graphs, very few changes are needed.





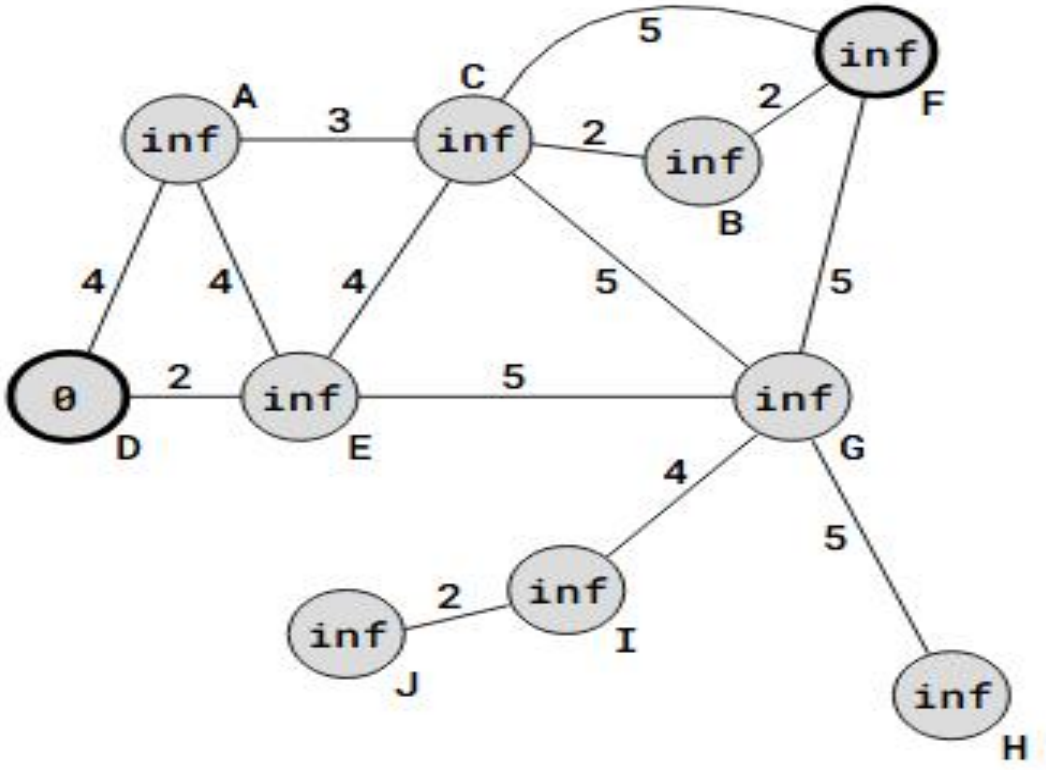
Dijkstra's Algorithm on Directed Graphs (2/2)

- There is a key difference: in this case, vertex **B** cannot be visited from **D**, and this means that the shortest distance from **D** to **F** is now **11**, not **10**, because the path can no longer go through vertex B.



Dijkstra's Algorithm with a Single Destination Vertex (1/3)

- Let's say we are only interested in finding the shortest path between two vertices, like finding the shortest distance between vertex **D** and vertex **F** in the graph below.

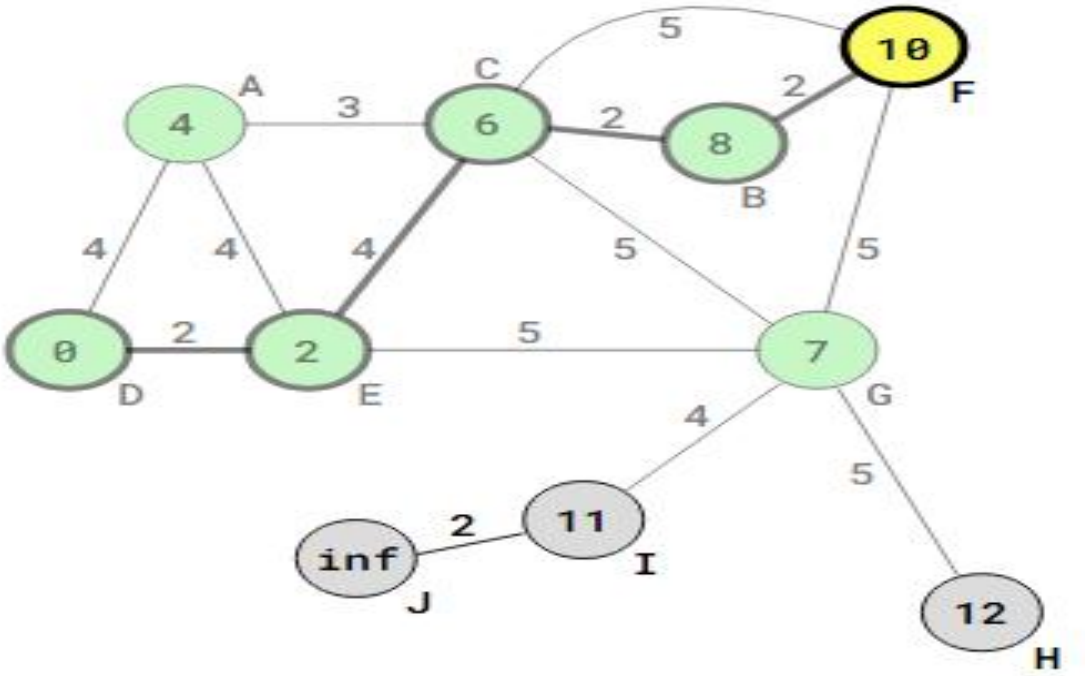


Dijkstra's Algorithm with a Single Destination Vertex (2/3)

- **Dijkstra's algorithm** is normally used for finding the shortest path from one source vertex to all other vertices in the graph, but it can also be modified to only find the shortest path from the source to a single destination vertex, by just stopping the algorithm when the destination is reached (visited).
- This means that for the specific graph, Dijkstra's algorithm will stop after visiting **F** (**the destination vertex**), before visiting vertices **H**, **I** and **J** because they are farther away from **D** than **F** is.

Dijkstra's Algorithm with a Single Destination Vertex (3/3)

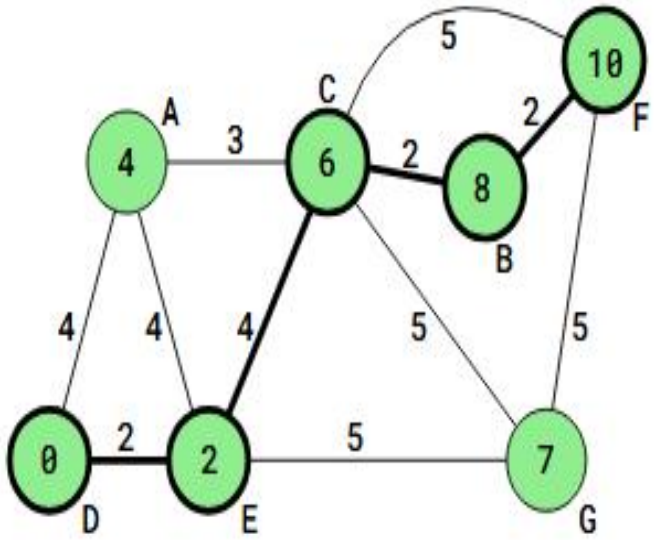
- The vertex **F** has just got updated with distance **10** from vertex **B**. Since **F** is the unvisited vertex with the lowest distance from **D**, it would normally be the next current vertex, but since it is the destination, the algorithm stops. If the algorithm did not stop, **J** would be the next vertex to get an updated distance 11+2=13, from vertex **I**.



Returning The Paths from Dijkstra's Algorithm

- With a few adjustments, the actual shortest paths can also be returned by Dijkstra's algorithm, in addition to the shortest path values. So for example, instead of just returning that the shortest path value is 10 from vertex D to F, the algorithm can also return that the shortest path is "D->E->C->B->F".

To return the path, we create a **predecessors array** to keep the previous vertex in the shortest path for each vertex. The predecessors array can be used to backtrack to find the shortest path for every vertex.



Bellman-Ford Algorithm (1/2)

- The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices.

- The Bellman-Ford algorithm can also be used for graphs with positive edges (both directed and undirected), like we can with Dijkstra's algorithm, but Dijkstra's algorithm is **preferred** in such cases because it is **faster**.

Bellman-Ford Algorithm (2/2)

- Using the Bellman-Ford algorithm on a graph with **negative cycles** will not produce a result of shortest paths because in a negative cycle we can always go one more round and get a shorter path.

- A **negative cycle** is a path we can follow in circles, where the sum of the edge weights is negative.

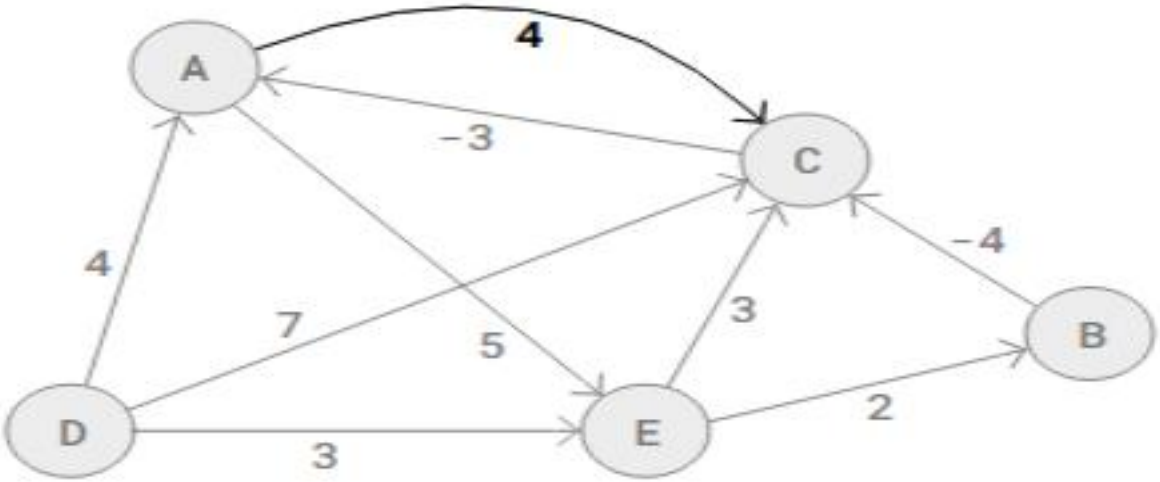
Bellman-Ford Algorithm (How it works) (1/2)

1. Set initial distance to **zero** for the **source vertex**, and set initial distances to **infinity** for **all other vertices**.
2. For each edge, check if a shorter distance can be calculated, and update the distance if the calculated distance is shorter.
3. Check all edges (**step 2**) **V-1 times**. This is as many times as there are vertices **(V)**, minus one.
4. **Optional**: Check for negative cycles.

Bellman-Ford Algorithm (How it works) (2/2)

- The **Bellman-Ford algorithm** is actually quite straight forward, because it **checks all edges**, using the adjacency matrix. Each check is to see if a **shorter distance** can be made by going from the vertex on one side of the edge, via the edge, to the vertex on the other side of the edge. And this check of all edges is done V−1 times, with **V** being the number of vertices in the graph.

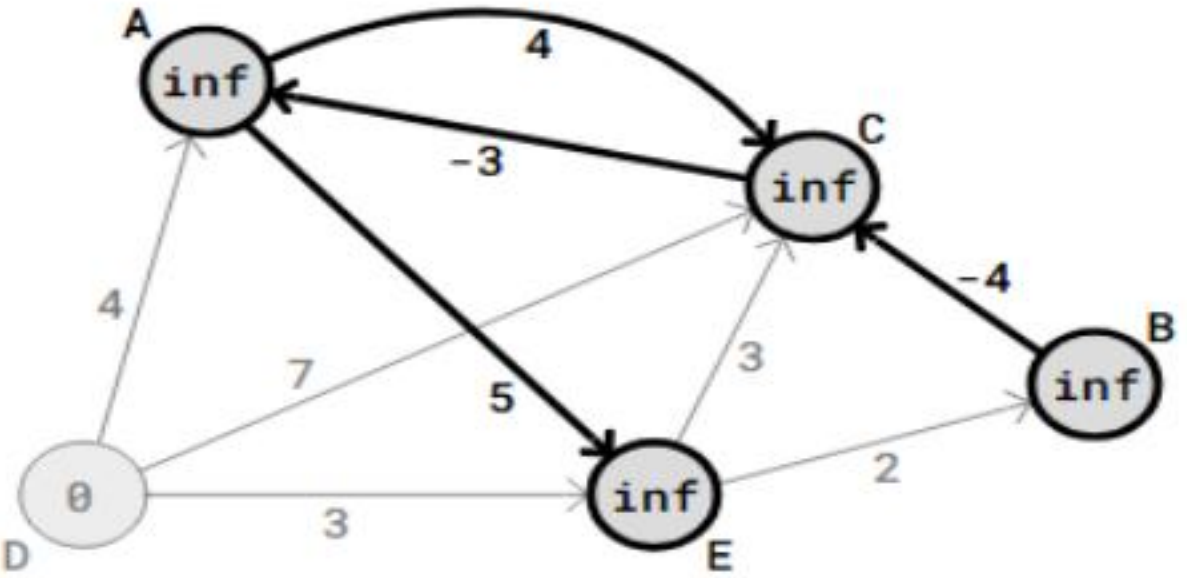
Illustrative Example (1/9)



	A	B	C	D	E
A			4		5
B			-4		
C	-3				
D	4		7		3
E		2	3		

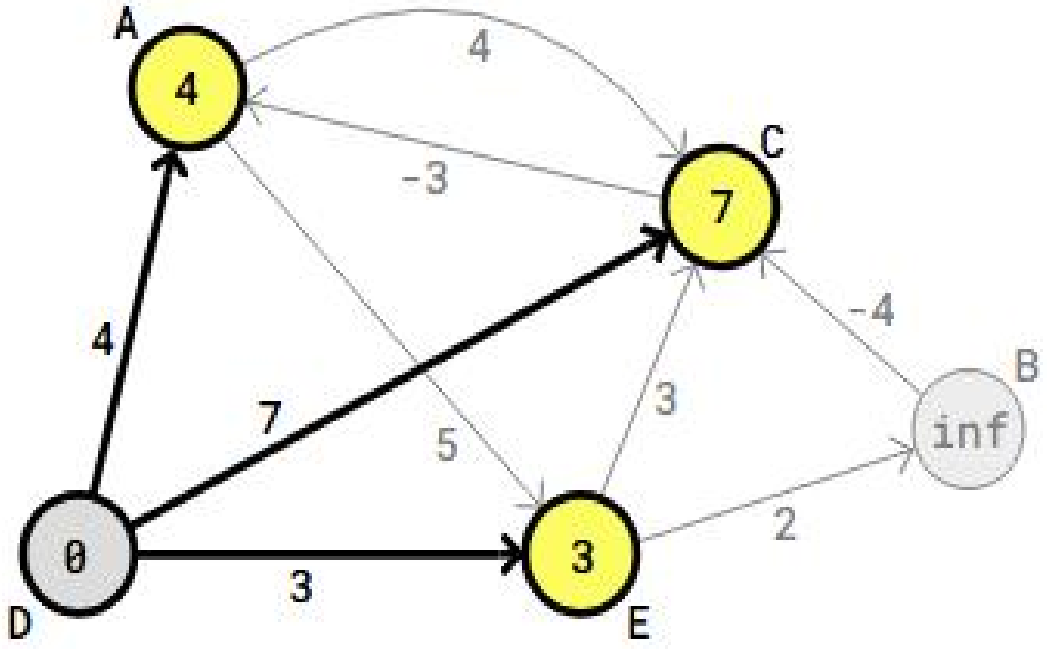
Illustrative Example (2/9)

- The first four edges that are checked in our graph are A->C, A->E, B->C, and C->A. These first four edge checks do not lead to any updates of the shortest distances because the starting vertex of all these edges has an infinite distance.



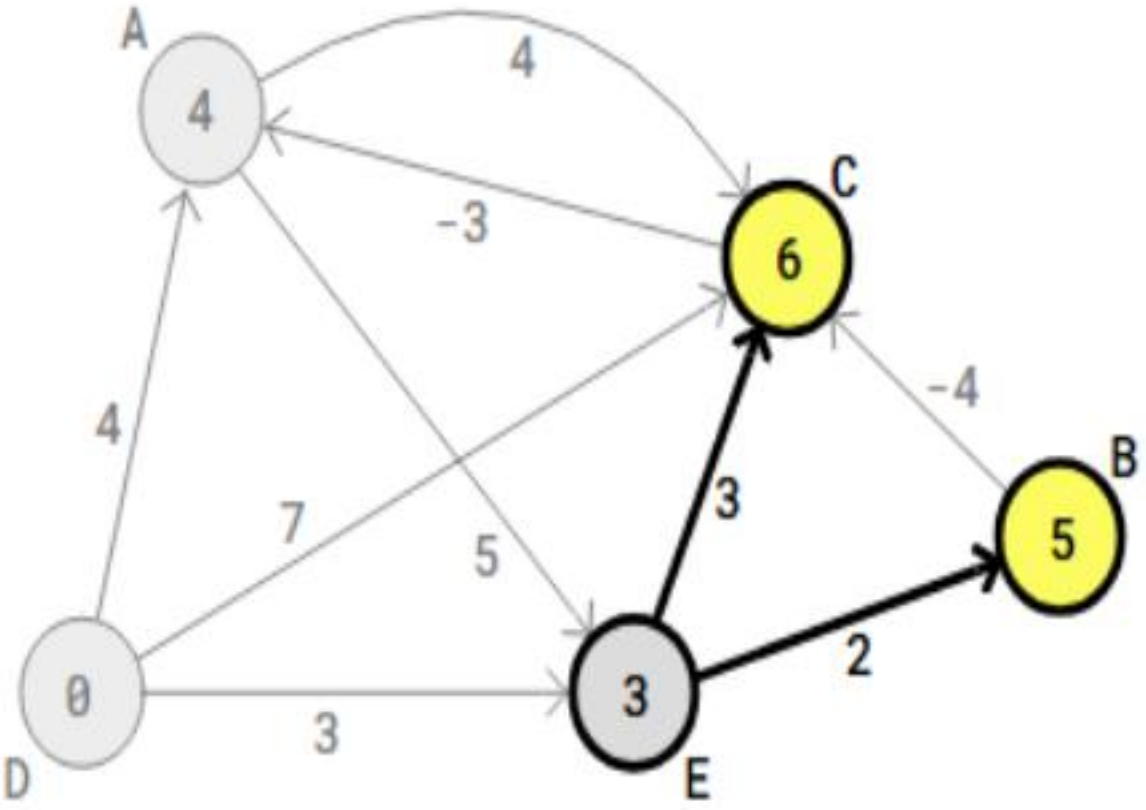
Illustrative Example (3/9)

- After the edges from vertices **A**, **B**, and **C** are checked, the edges from **D** are checked. Since the starting point **(vertex D)** has distance **0**, the updated distances for **A**, **B**, and **C** are the edge weights going out from vertex **D**.



Illustrative Example (4/9)

- The next edges to be checked are the edges going out from vertex **E**, which leads to updated distances for vertices **B** and **C**.

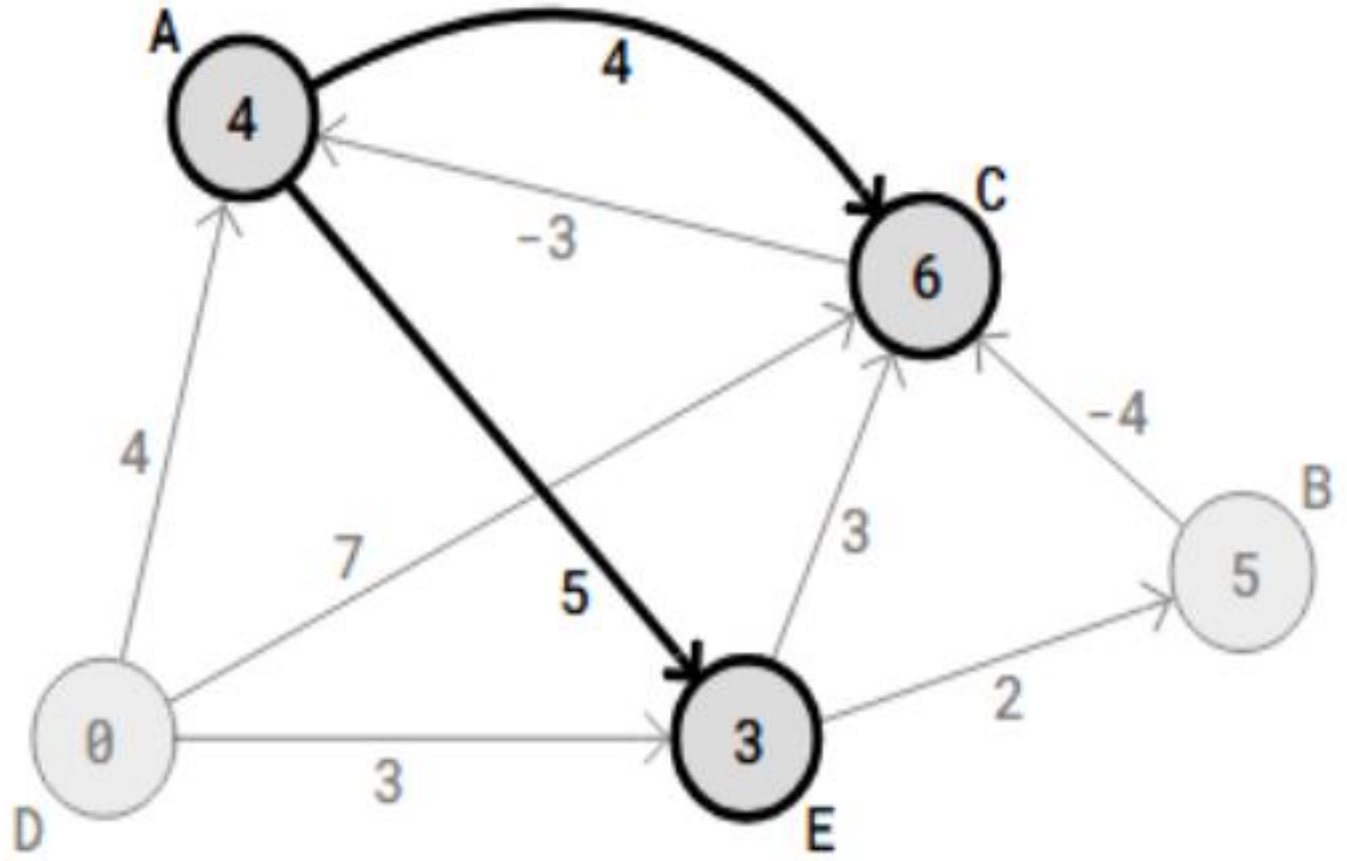


Illustrative Example (5/9)

- The Bellman-Ford algorithm have now checked all edges **1 time**. The algorithm will check all edges **3 more times** before it is finished, because Bellman-Ford will check all edges as many times as there are vertices in the graph, minus 1.

- The algorithm starts checking all edges a second time, starting with checking the edges going out from vertex **A**. Checking the edges **A->C** and **A->E** do not lead to updated distances.

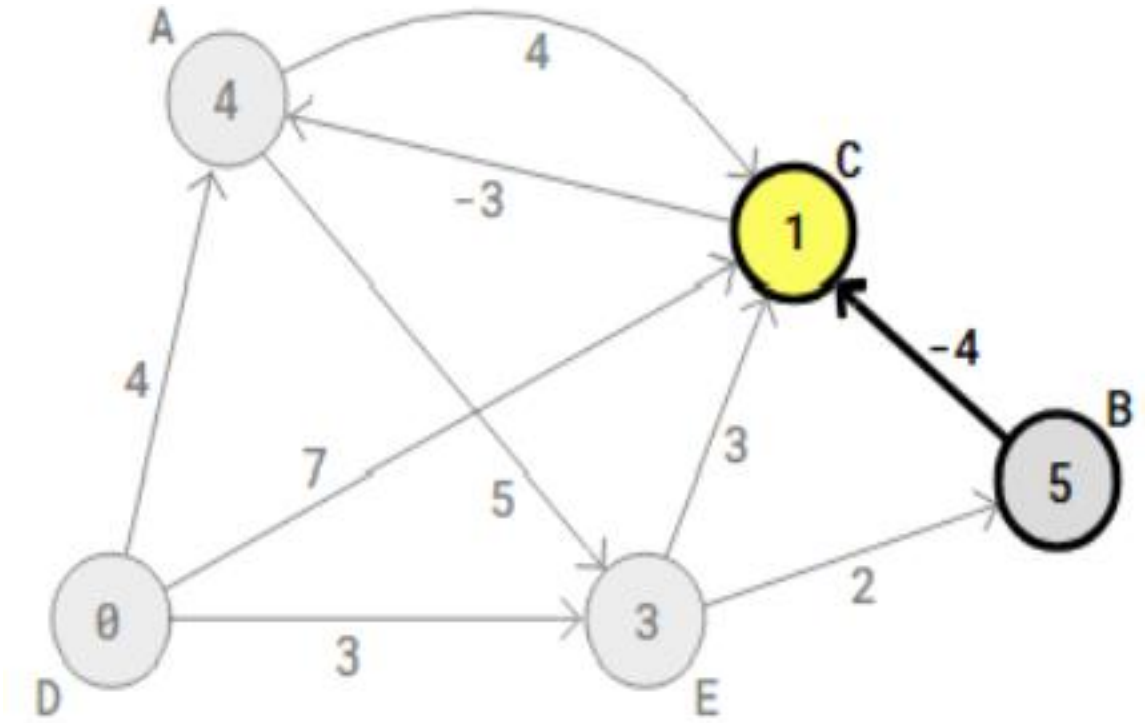
Illustrative Example (6/9)



Illustrative Example (7/9)

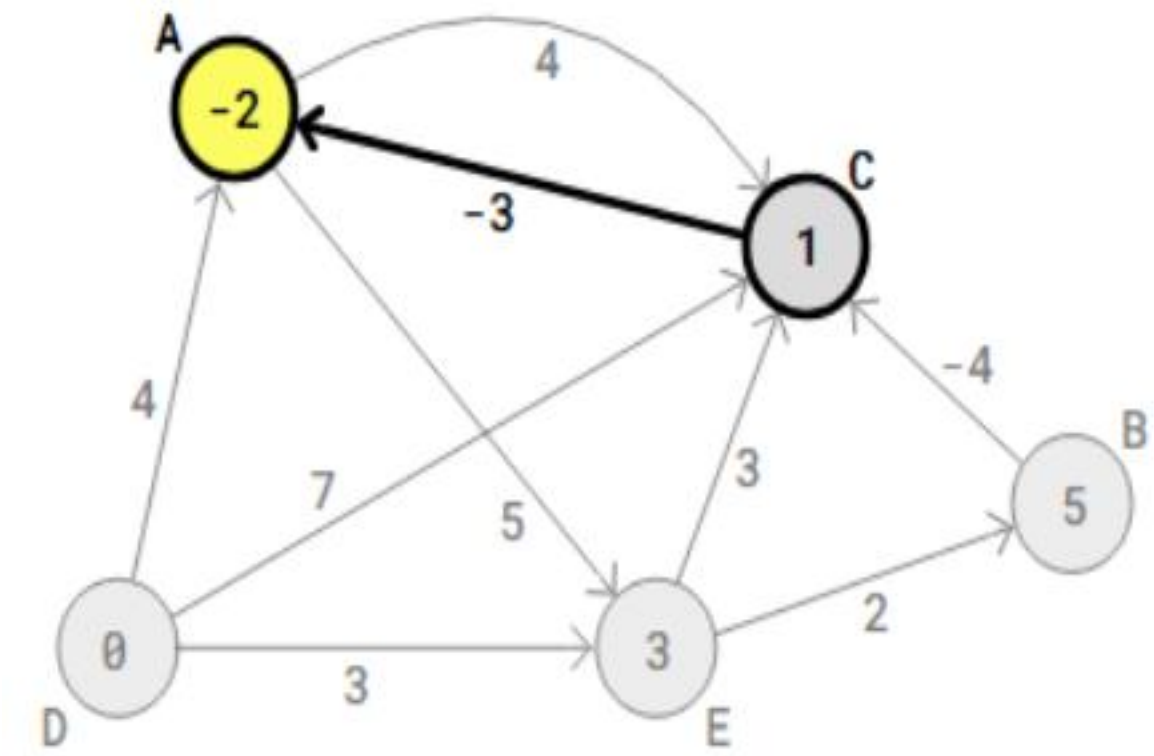
- The next edge to be checked is **B->C**, going out from vertex **B**.

  This leads to an updated distance from vertex **D** to **C** of **5-4=1**.



Illustrative Example (8/9)

- Checking the next edge **C->A**, leads to an updated distance **1-3=-2** for vertex **A**.



Illustrative Example (9/9)

- The check of edge **C->A** in **round 2** of the **Bellman-Ford algorithm** is actually the last check that leads to an updated distance for this specific graph. The algorithm will continue to check all edges **2 more times** **without updating any distances**.

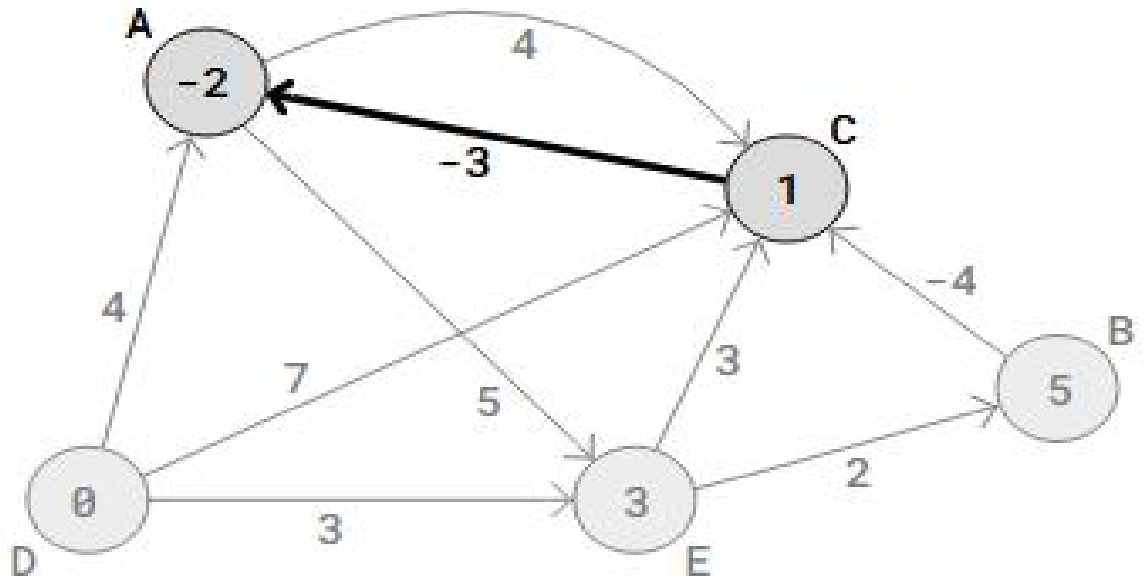
- Checking all edges **V−1 times** in the **Bellman-Ford algorithm** may seem like a lot, but it is done this many times to make sure that the shortest distances will always be found.

Negative Edges in The Bellman-Ford Algorithm (1/2)

- To say that the **Bellman-Ford algorithm** finds the "shortest paths" is not intuitive, because how can we draw or imagine distances that are negative? So, to make it easier to understand we could instead say that it is the "**cheapest paths**" that are found with Bellman-Ford.
- In practice, the **Bellman-Ford algorithm** could for example help us to find **delivering routes** where the edge weights represent the cost of fuel and other things, minus the money to be made by driving that edge between those two vertices.

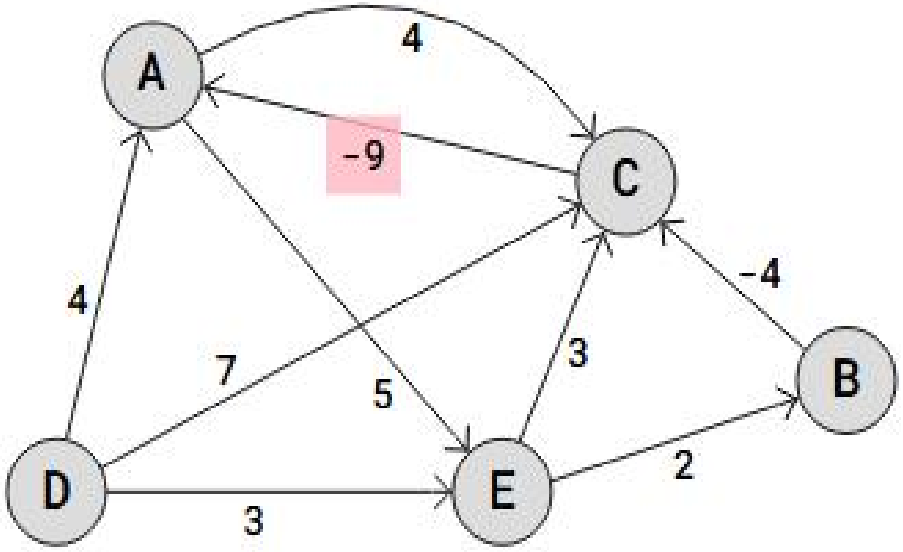
Negative Edges in The Bellman-Ford Algorithm (2/2)

- The **-3** weight on edge C->A could mean that the fuel cost is $5 driving from **C** to **A**, and that we get paid **$8 for picking up packages in C and delivering them in A**. So we end up earning **$3** more than we spend. Therefore, a total of **$2** can be made by driving the delivery route **D->E->B->C->A** in our graph above.



Negative Cycles in The Bellman-Ford Algorithm (1/2)

- If we can go in circles in a graph, and the sum of edges in that circle is **negative**, we have a **negative cycle**.
- By changing the weight on edge C->A from -3 to -9, we get **two negative cycles**: **A->C->A** and **A->E->C->A**. And every time we check these edges with the **Bellman-Ford algorithm**, the distances we calculate and update just become lower and lower.



Negative Cycles in The Bellman-Ford Algorithm (2/2)

- The problem with **negative cycles** is that a shortest path does not exist, because we can always go one more round to get a path that is **shorter**.

- That is why it is useful to implement the Bellman-Ford algorithm with **detection for negative cycles**.

Detection of Negative Cycles in the B-F Algorithm

- After running the **Bellman-Ford algorithm**, checking all edges in a graph **V−1 times**, all the shortest distances are found.
- If the graph **contains negative cycles**, and we go **one more round checking all edges**, we will find at least one shorter distance in this last round.
- So, to detect negative cycles in the **Bellman-Ford algorithm**, after checking all edges V−1 times, **we just need to check all edges one more time**, and if we **find a shorter distance this last time**, we **can conclude that a negative cycle must exist**.

Returning The Paths from The B-F Algorithm

- By recording the **predecessor** of each vertex whenever an edge is **relaxed**, we can use that later in order to have the actual shortest paths. This means we can give more information in our result, with the actual path in addition to the path weight: "**D->E->B->C->A, Distance: -2**".