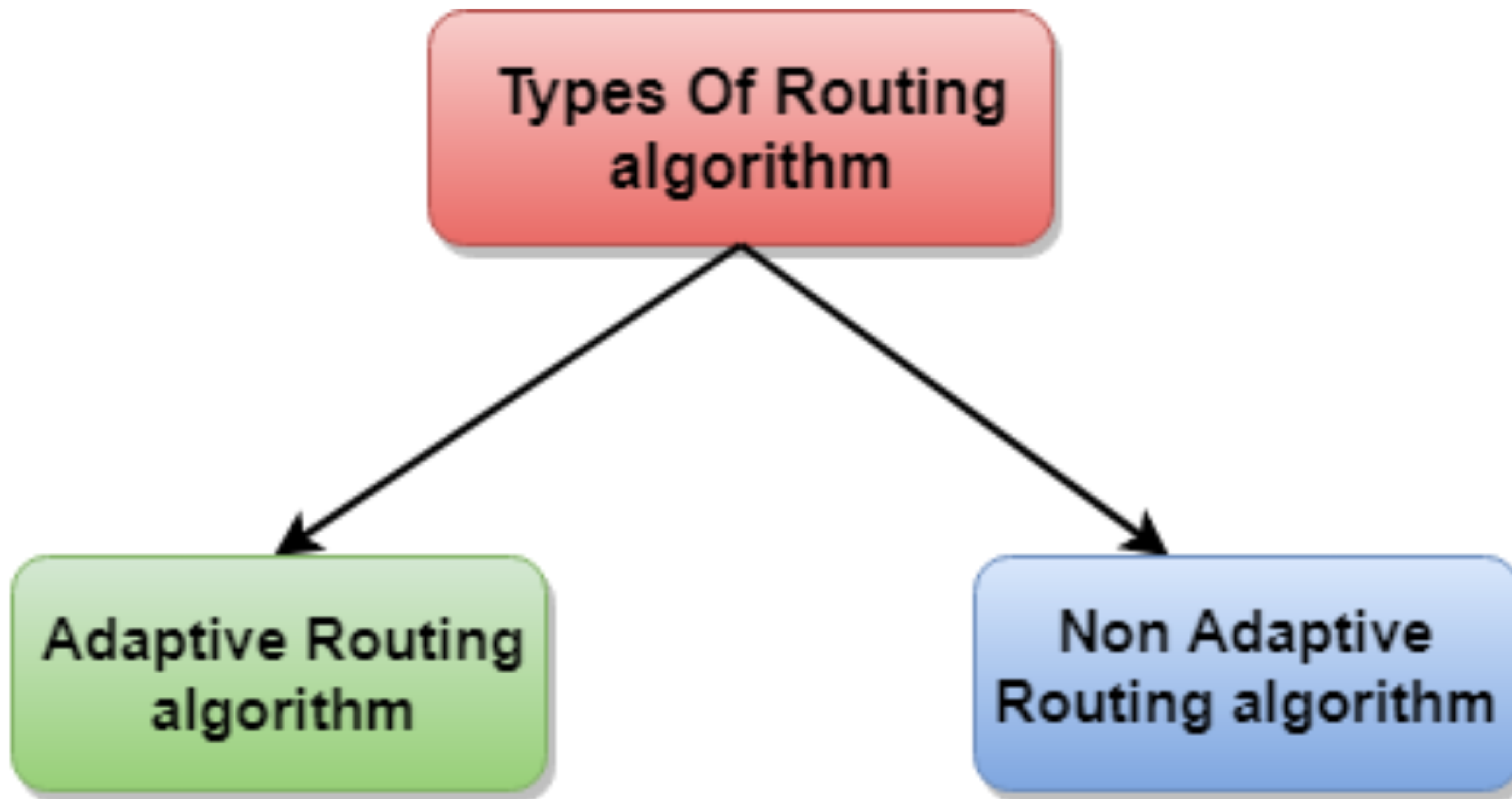
Classification of Routing Algorithms

Routing is process of establishing the routes that data packets must follow to reach the destination. In this process, a routing table table is created which contains information regarding routes which data packets follow. Various routing algorithm are used for the purpose of deciding which route an incoming data packet needs to be transmitted on to reach destination efficiently.

The Routing algorithm is divided into two categories:

- Adaptive Routing algorithm
- Non-adaptive Routing algorithm



Adaptive Routing algorithm

- An adaptive routing algorithm is also known as dynamic routing algorithm.
- This algorithm makes the routing decisions based on the topology and network traffic.

An adaptive routing algorithm can be classified into three parts:

- **(a) Isolated** – In this method each, node makes its routing decisions using the information it has without seeking information from other nodes. The sending nodes doesn't have information about status of particular link. Disadvantage is that packet may be sent through a congested network which may result in delay. Examples: Hot potato routing, backward learning.

- **(b) Centralized** – In this method, a centralized node has entire information about the network and makes all the routing decisions. Advantage of this is only one node is required to keep the information of entire network and disadvantage is that if central node goes down the entire network is done.

- **(c) Distributed –** In this method, the node receives information fro its neighbors and then takes the decision about routing the packets. Disadvantage is that the packet may be delayed if there is change in between interval in which it receives information and sends packet.

Non-Adaptive Routing algorithm

- Non Adaptive routing algorithm is also known as a static routing algorithm.
- When booting up the network, the routing information stores to the routers.

- Non Adaptive routing algorithms do not take the routing decision based on the network topology or network traffic.

The Non-Adaptive Routing algorithm is of two types:

Flooding: In case of flooding, every incoming packet is sent to all the outgoing links except the one from it has been reached. The disadvantage of flooding is that node may contain several copies of a particular packet.

Random walks: In case of random walks, a packet sent by the node to one of its neighbors randomly. An advantage of using random walks is that it uses the alternative routes very efficiently.

The examples of static algorithms are :

(*i*) Shortest path routing

(*ii*) Flooding, and

(*iii*) Flow based routing.

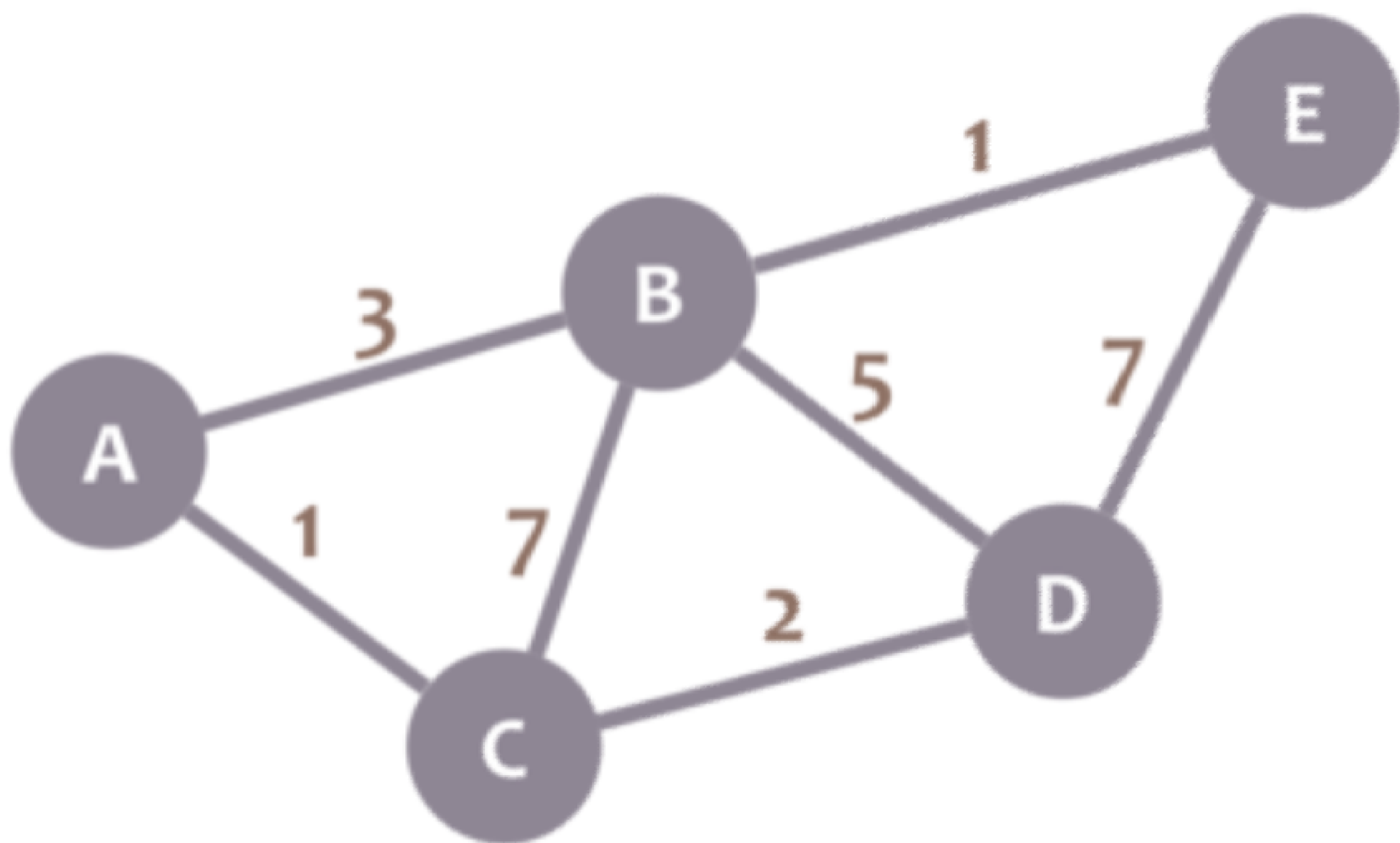
Shortest Path Routing

In shortest path routing, the topology communications network is represented using a directed weighted graph. The nodes in the graph represent switching elements and the directed arcs in the graph represent communication links between switching elements. Each arc has a weight that represents the cost of sending a packet between two nodes in a particular direction. This cost is generally a positive value that can inculcates such factors as delay, throughput, error rate, monetary cost etc. A path between two nodes may go through several intermediary nodes and arc. The objective in shortest path routing is to find a path between two nodes that has the smallest total cost, where the total cost of a path is the sum of the arc costs in that path.

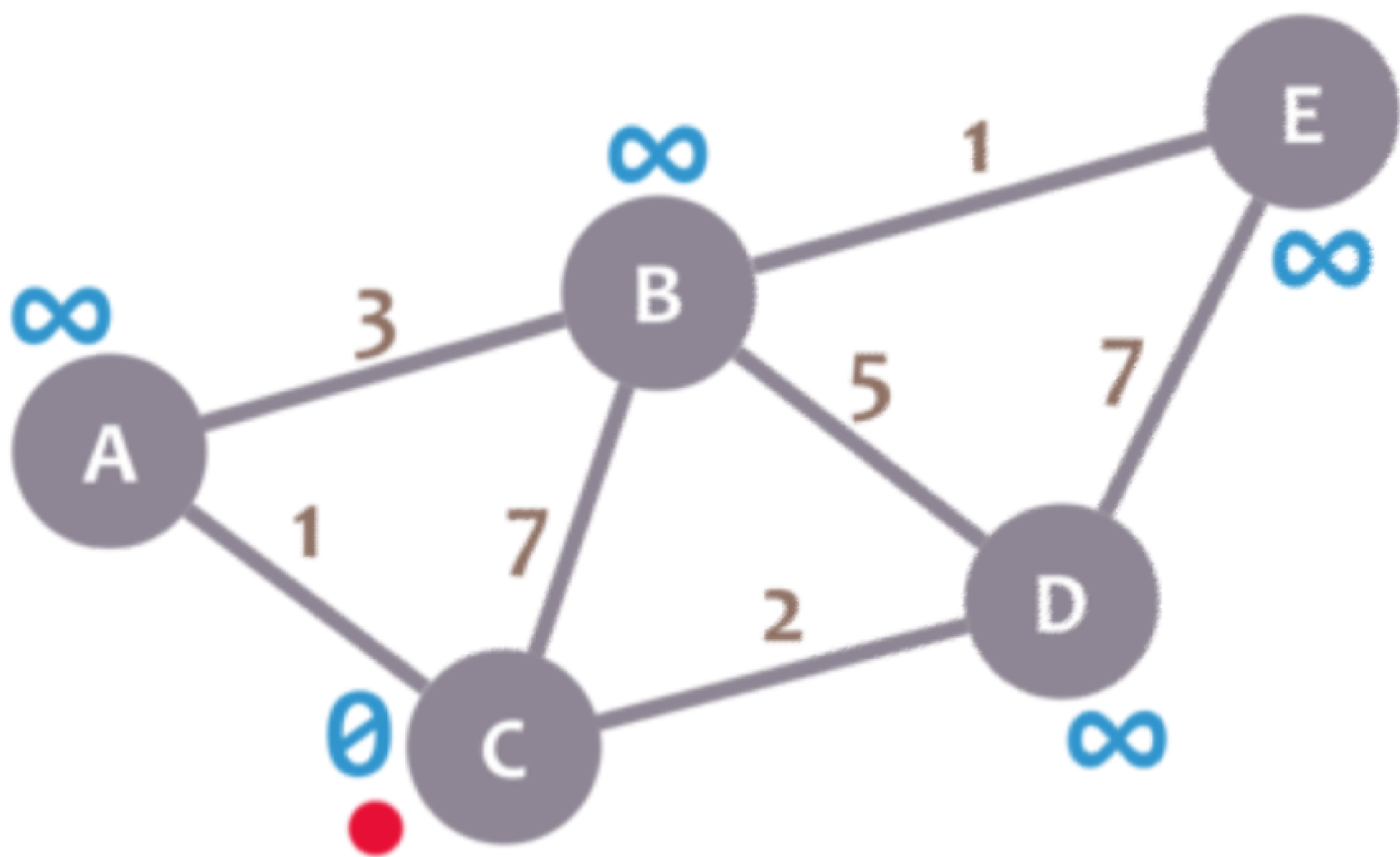
Dijkstra's Shortest Path Algorithm

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra’s algorithm. The algo-rithm creates a tree of shortest paths from the starting vertex, the source, to all other points in the graph.

Dijkstra's Algorithm allows you to calculate the shortest path between one node (you pick which one) and *every other node in the graph*. You'll find a description of the algorithm at the end of this page, but, let's study the algorithm with an explained example! Let's calculate the shortest path between node C and the other nodes in our graph:

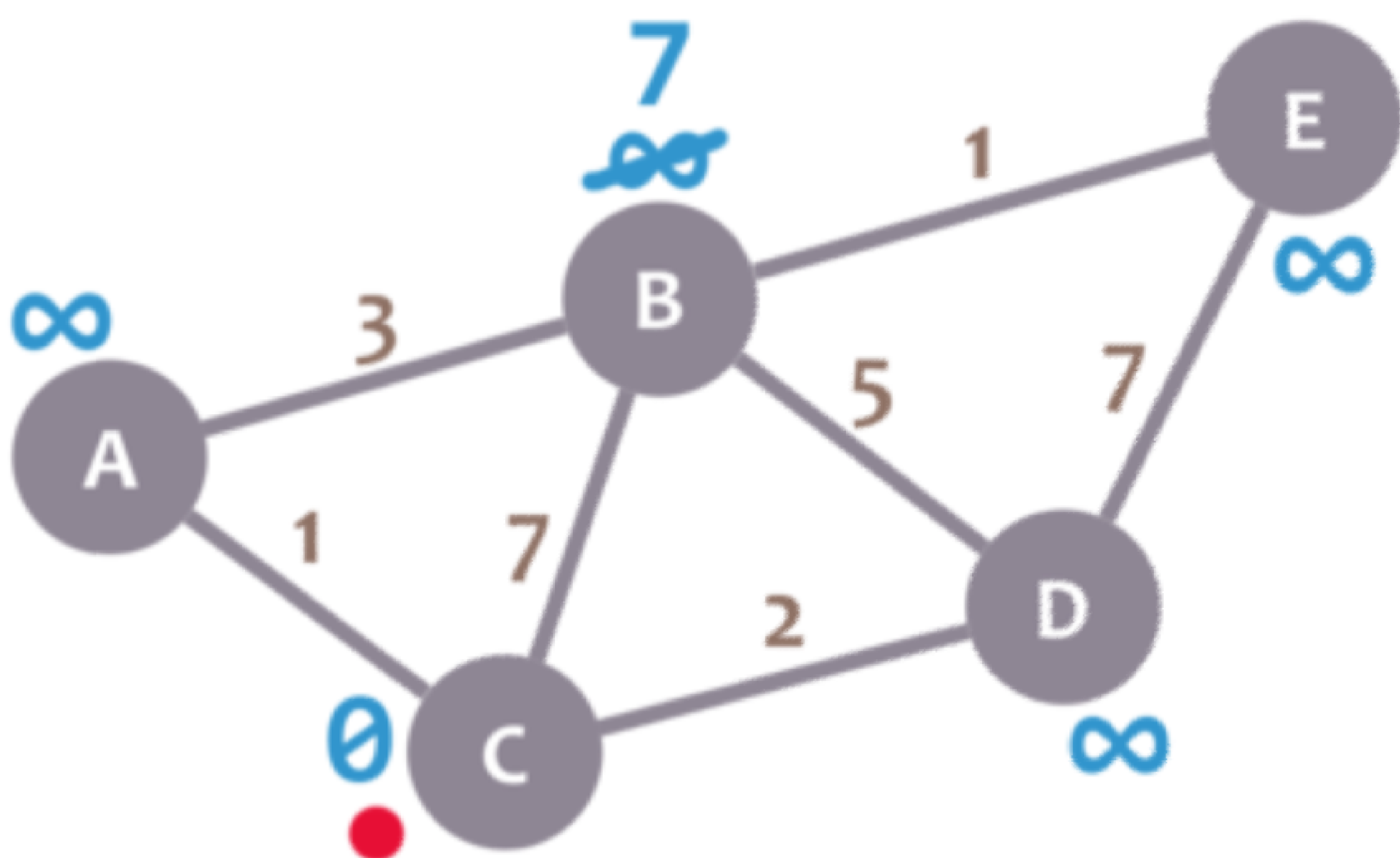


During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):

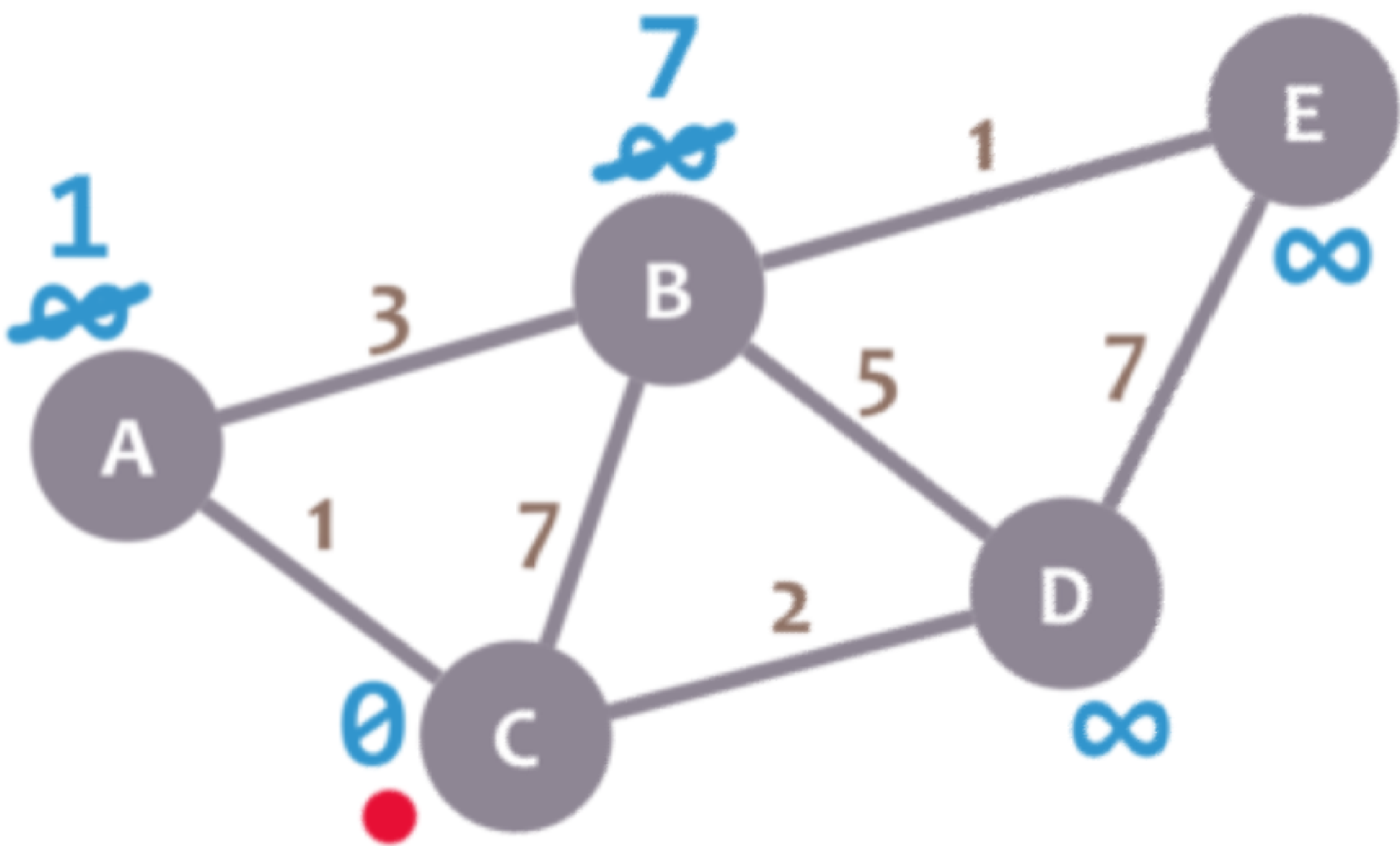


We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

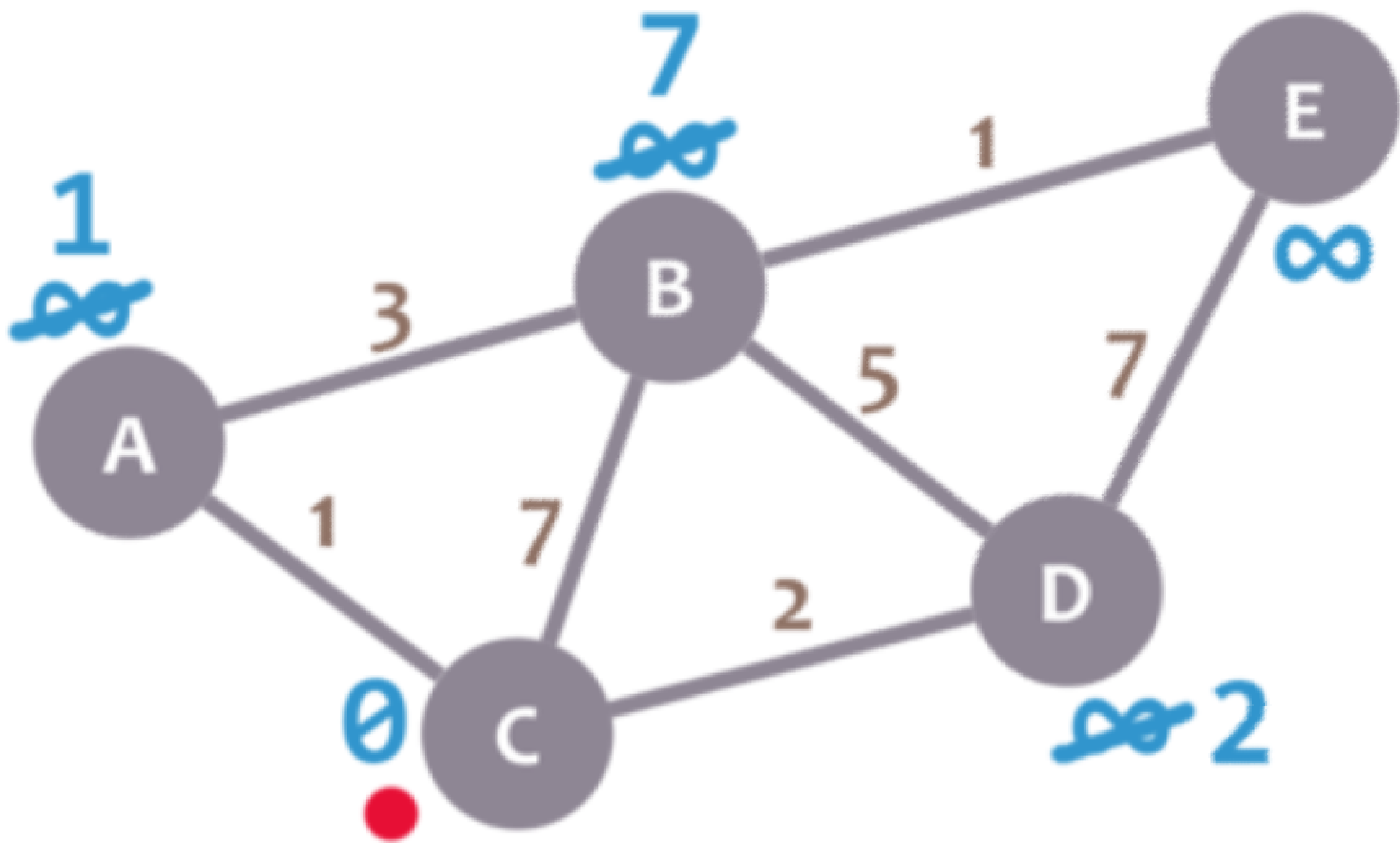
Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



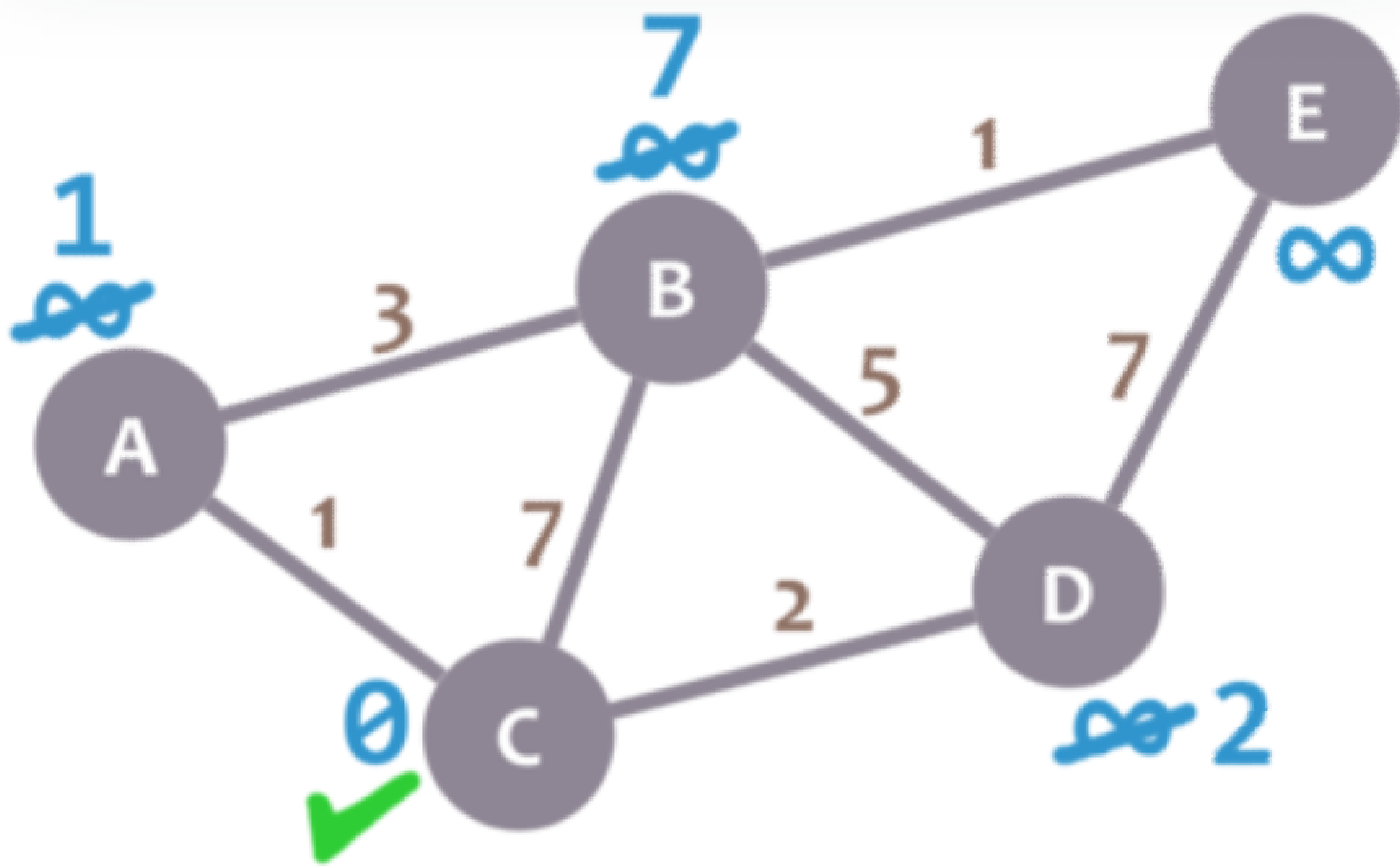
So far, so good. Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



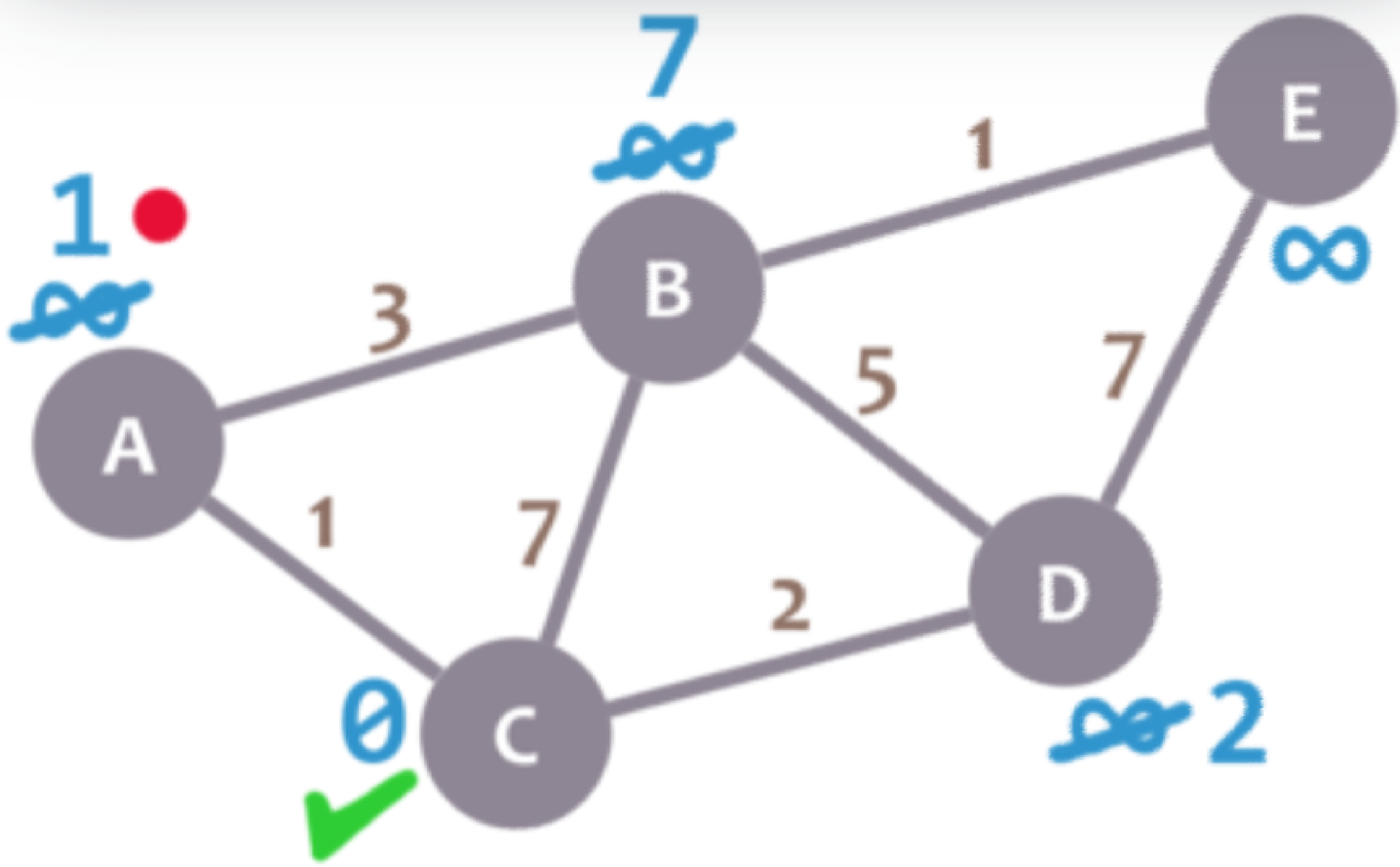
OK. Repeat the same procedure for D:



Great. We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:

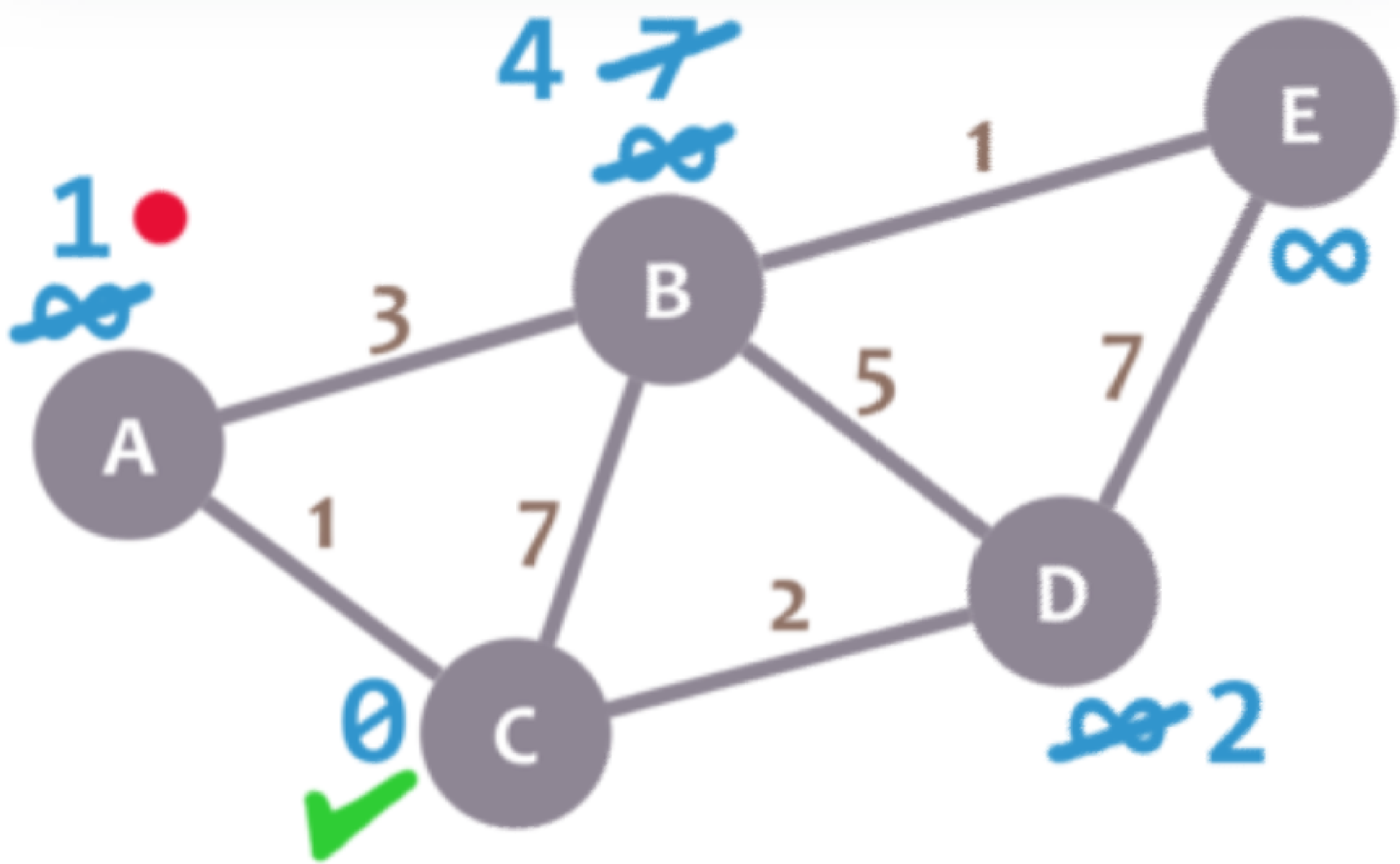


We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

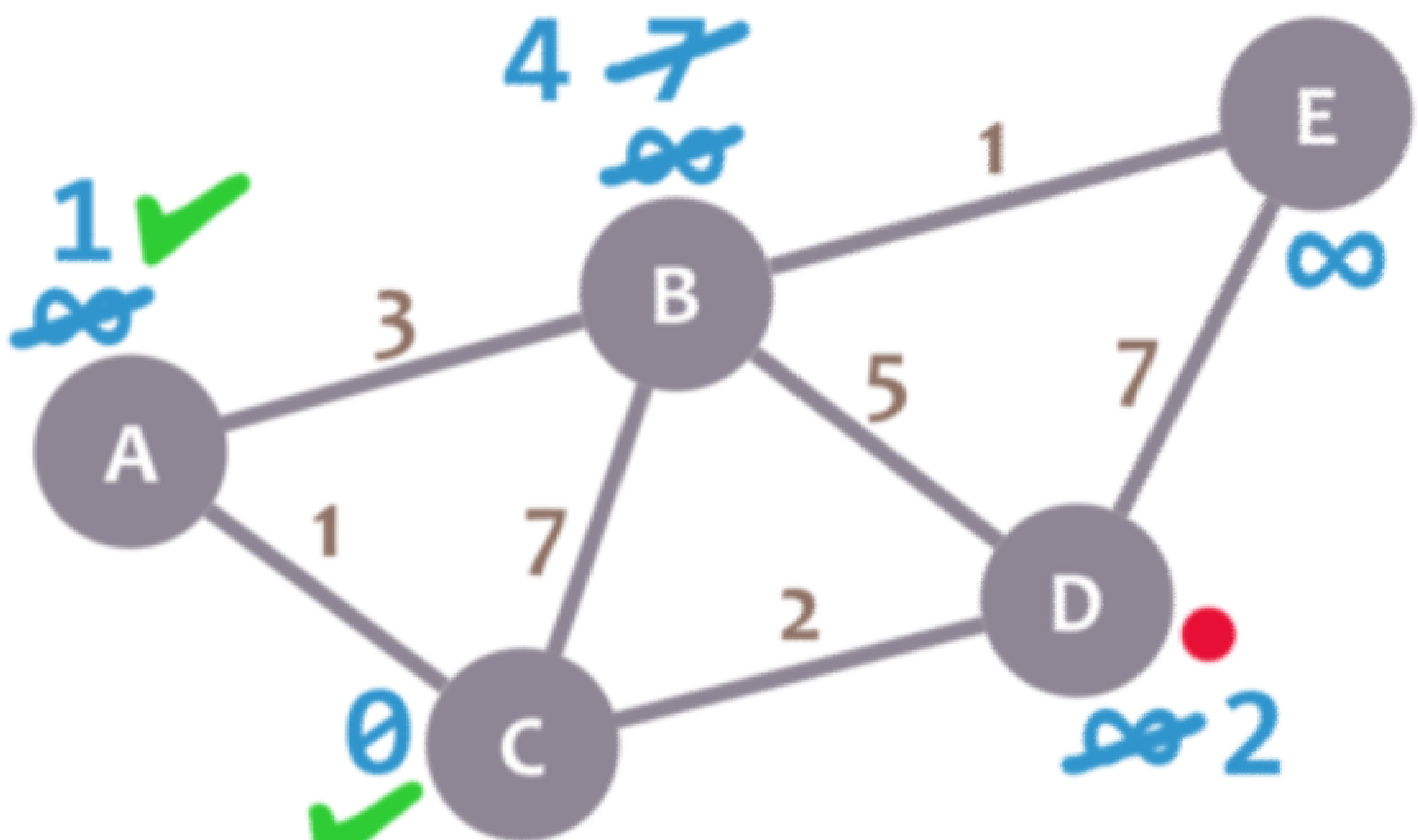


And now we repeat the algorithm. We check the neighbours of our current node, ignoring the visited nodes. This means we only check B.

For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



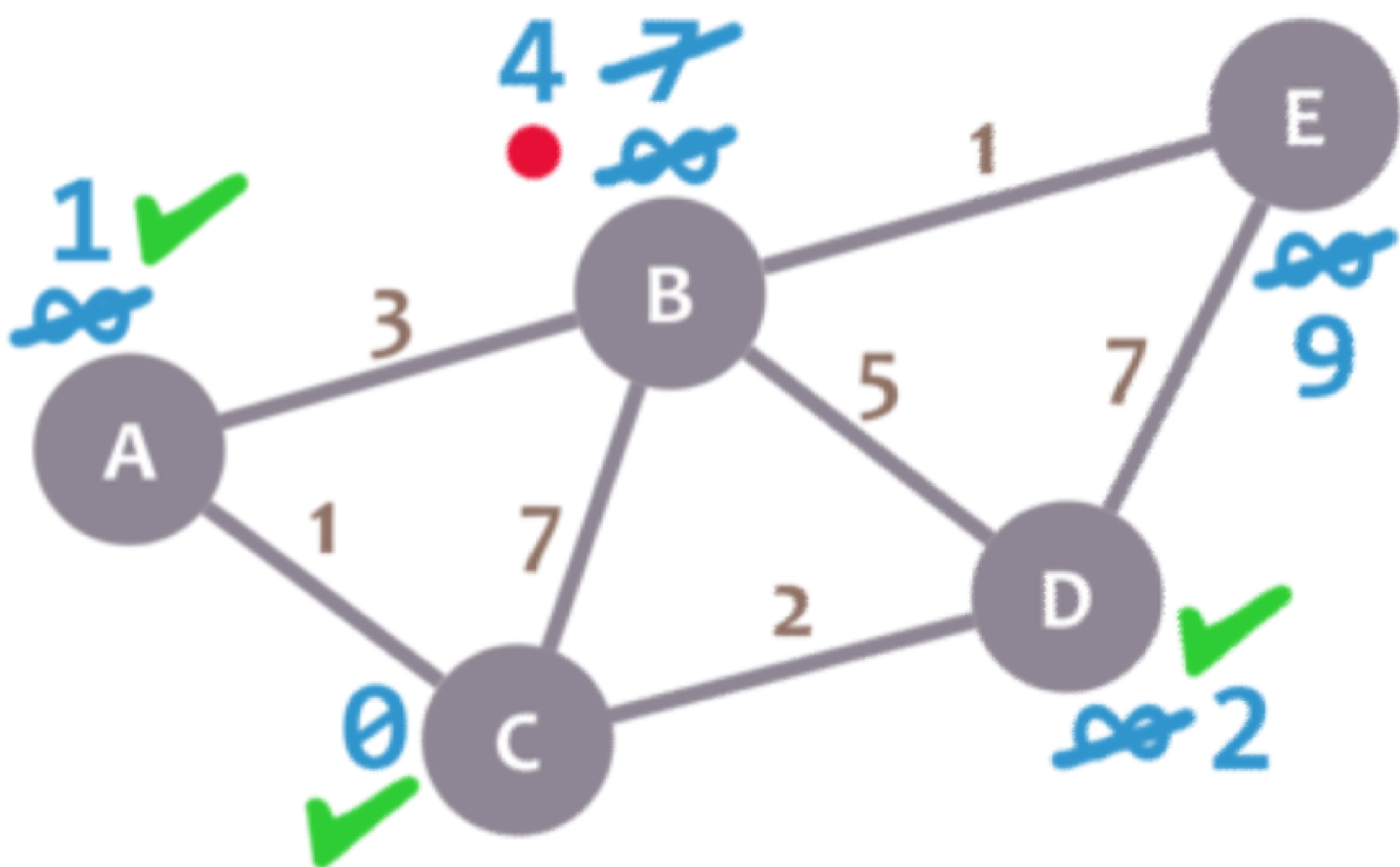
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



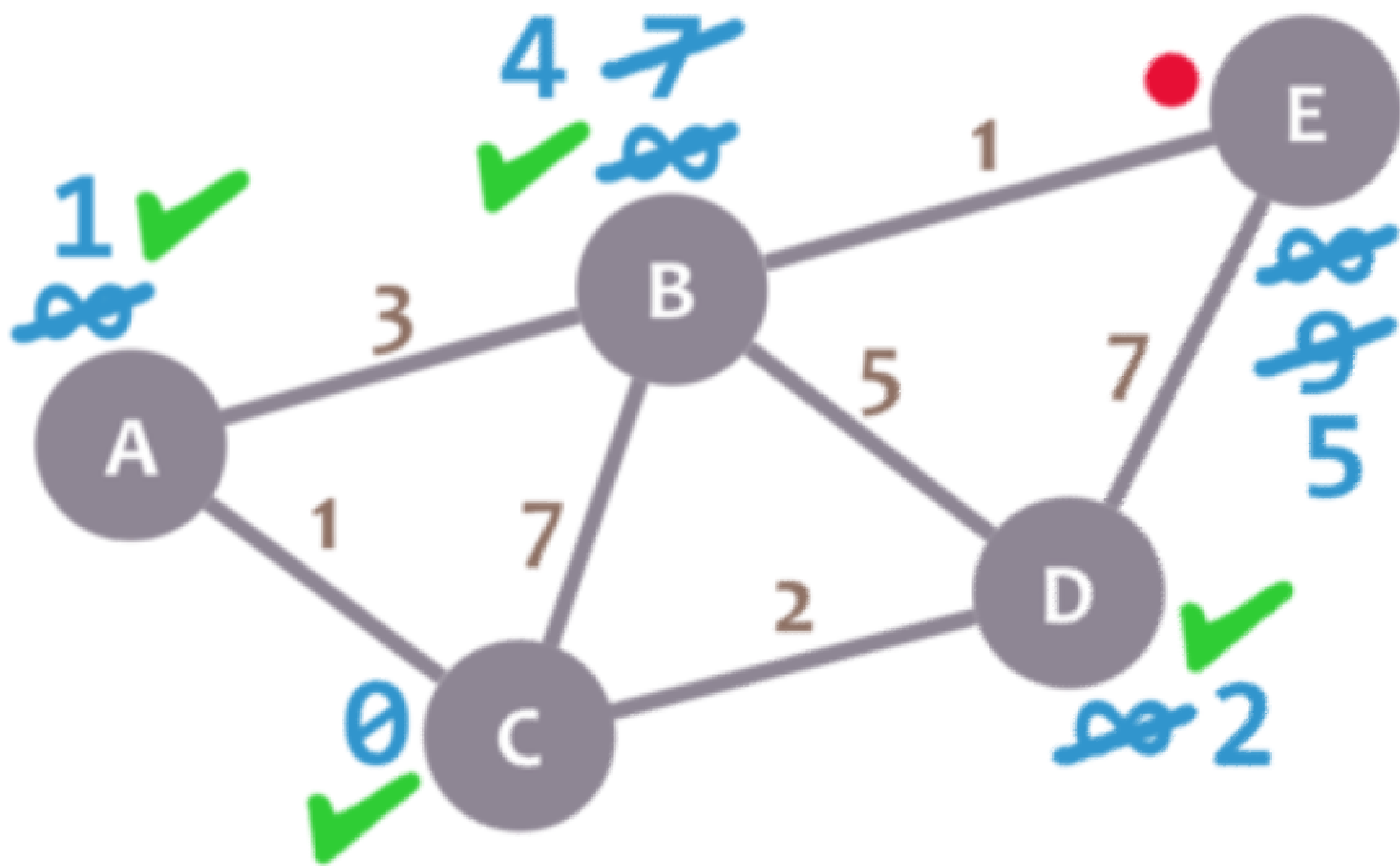
We repeat the algorithm again. This time, we check B and E.

For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

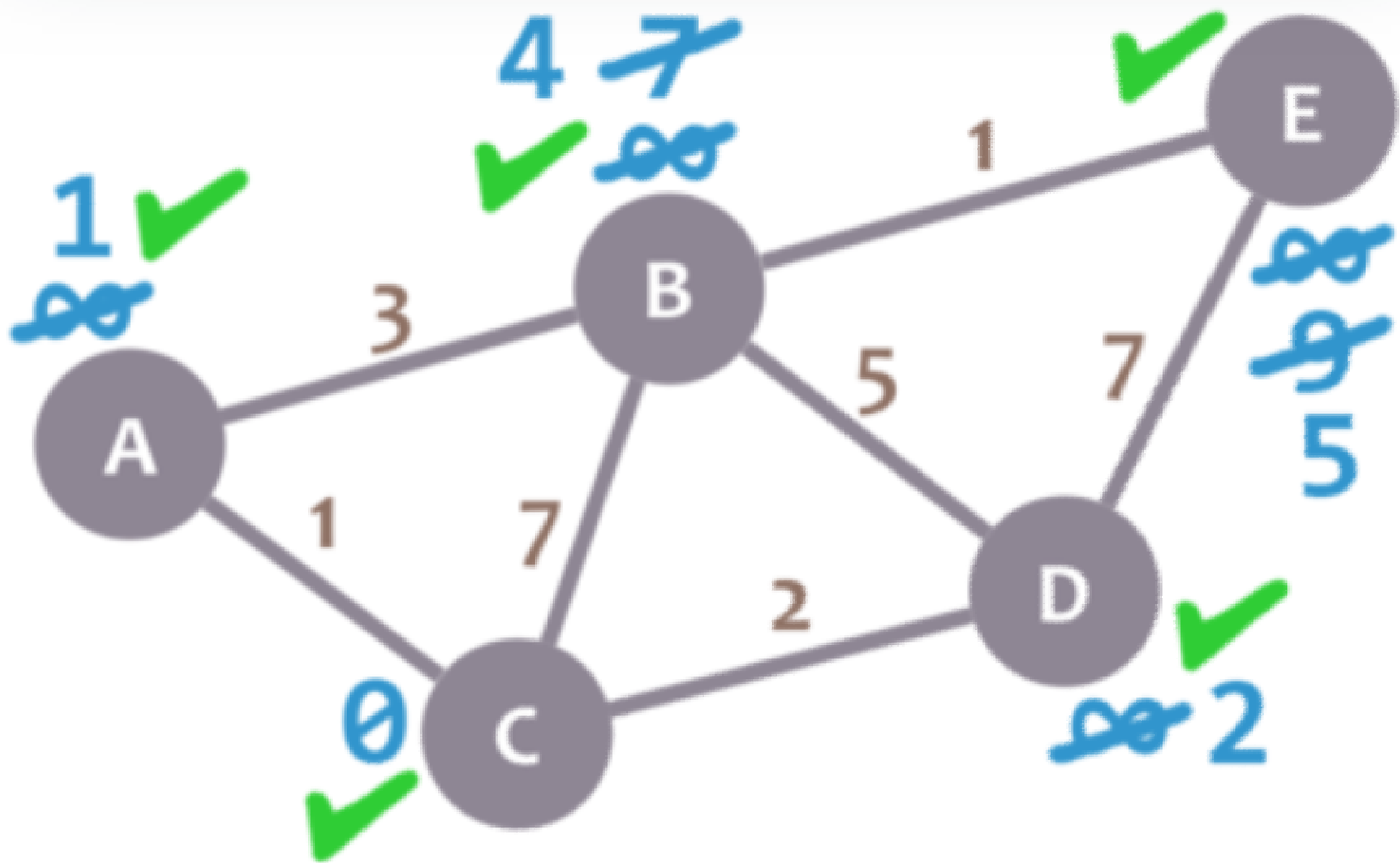
We mark D as visited and set our current node to B.



Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!