

2.1.1 Dijkstra's Shortest Path Algorithm

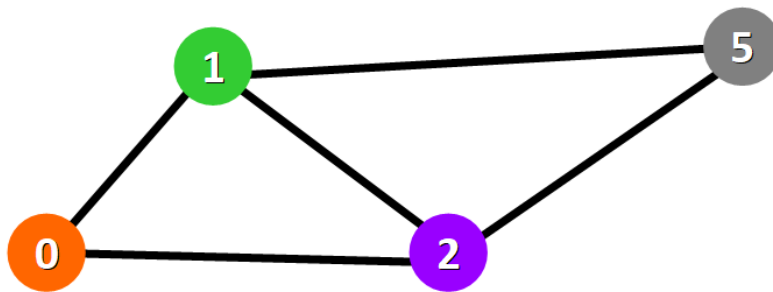
Introduction to Graphs

Basic Concepts

Graphs are data structures used to represent "connections" between pairs of elements.

- These elements are called **nodes**. They represent real-life objects, persons, or entities.
- The connections between nodes are called **edges**.

This is a graphical representation of a graph:

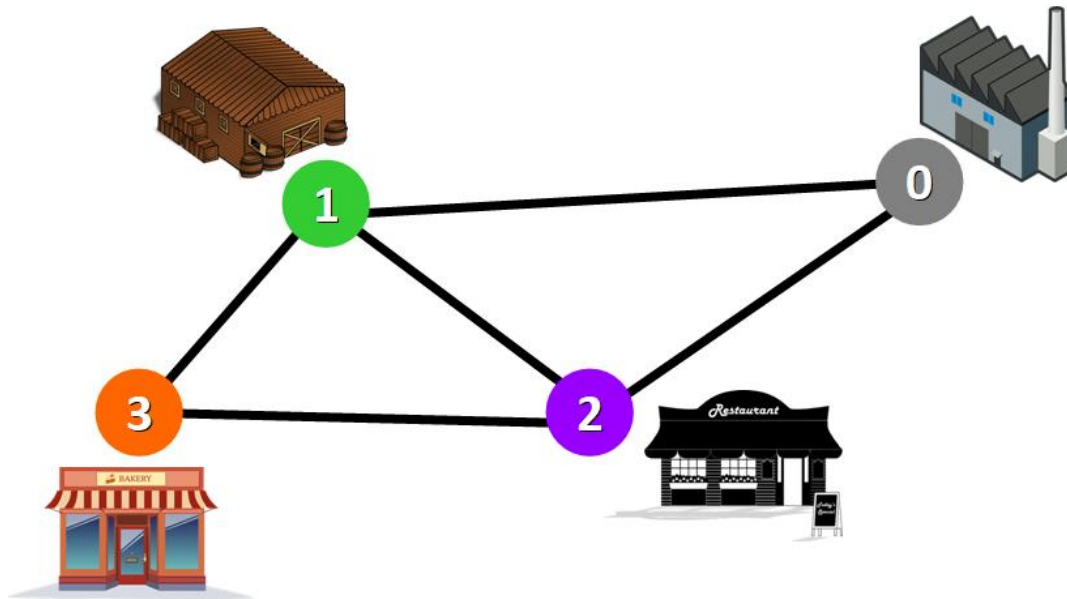


Nodes are represented with colored circles and **edges** are represented with lines that connect these circles.

□ **Tip:** Two nodes are connected if there is an edge between them.

Applications

Graphs are directly applicable to real-world scenarios. For example, we could use graphs to model a transportation network where nodes would represent facilities that send or receive products and edges would represent roads or paths that connect them (see below).

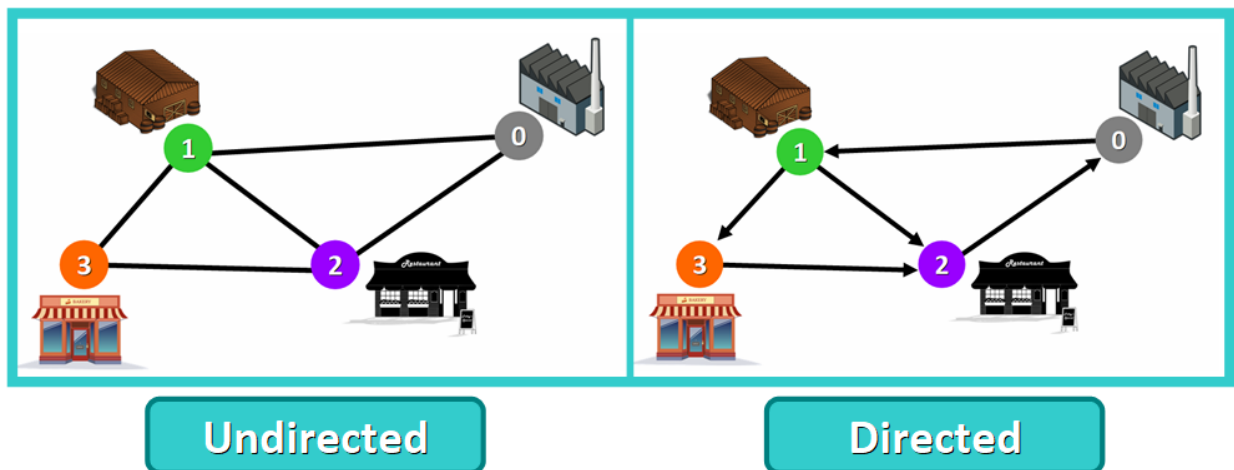


Network represented with a graph

Types of Graphs

Graphs can be:

- **Undirected:** if for every pair of connected nodes, you can go from one node to the other in both directions.
- **Directed:** if for every pair of connected nodes, you can only go from one node to another in a specific direction. We use arrows instead of simple lines to represent directed edges.

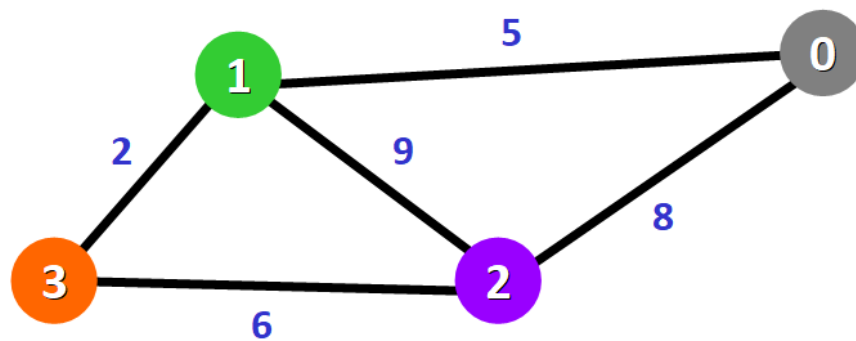


□ **Tip:** in this article, we will work with **undirected** graphs.

Weighted Graphs

A **weight graph** is a graph whose edges have a "weight" or "cost". The weight of an edge can represent distance, time, or anything that models the "connection" between the pair of nodes it connects.

For example, in the weighted graph below you can see a blue number next to each edge. This number is used to represent the weight of the corresponding edge.



□ **Tip:** These weights are essential for Dijkstra's Algorithm. You will see why in just a moment.

Introduction to Dijkstra's Algorithm

Now that you know the basic concepts of graphs, let's start diving into this amazing algorithm.

- Purpose and Use Cases
- History
- Basics of the Algorithm
- Requirements

Purpose and Use Cases

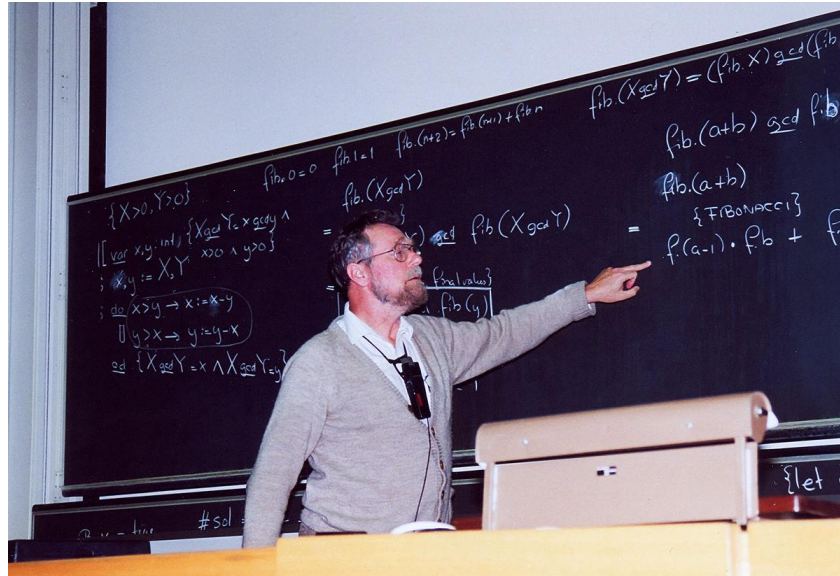
With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree.

This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

History

This algorithm was created and published by Dr. Edsger W. Dijkstra, a brilliant Dutch computer scientist and software engineer.

In 1959, he published a 3-page article titled "A note on two problems in connexion with graphs" where he explained his new algorithm.



Dr. Edsger Dijkstra at ETH Zurich in 1994 (image by Andreas F. Borchert)

During an interview in 2001, Dr. Dijkstra revealed how and why he designed the algorithm:

What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later. The publication is still quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame. — As quoted in the article Edsger W. Dijkstra from An interview with Edsger W. Dijkstra.

□ **Unbelievable, right?** In just 20 minutes, Dr. Dijkstra designed one of the most famous algorithms in the history of Computer Science.

Basics of Dijkstra's Algorithm

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Requirements

Dijkstra's Algorithm can only work with graphs that have **positive** weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.

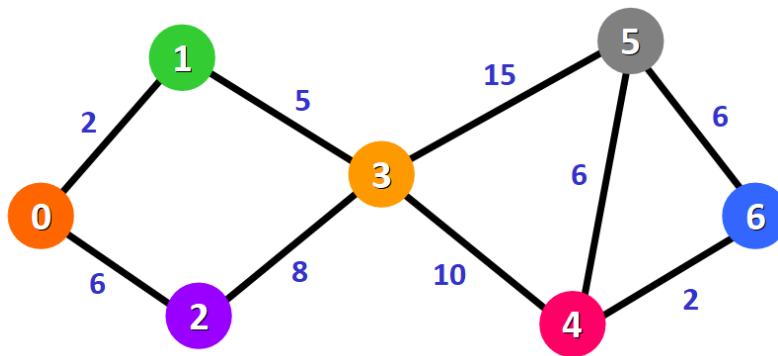
If there is a negative weight in the graph, then the algorithm will not work properly. Once a node has been marked as "visited", the current path to that node is marked as the shortest path to reach that node. And negative weights can alter this if the total weight can be decremented after this step has occurred.

Example of Dijkstra's Algorithm

Now that you know more about this algorithm, let's see how it works behind the scenes with a step-by-step example.

We have this graph:

5



The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

□ **Tip:** For this graph, we will assume that the weight of the edges represents the distance between two nodes.

We will have 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.

Initially, we have this list of distances (please see the list below):

- The distance from the source node to itself is 0. For this example, the source node will be node 0 but it can be any node that you choose.
- The distance from the source node to all other nodes has not been determined yet, so we use the infinity symbol to represent this initially.

Distance:

0: 0
1: ∞
2: ∞
3: ∞
4: ∞
5: ∞
6: ∞

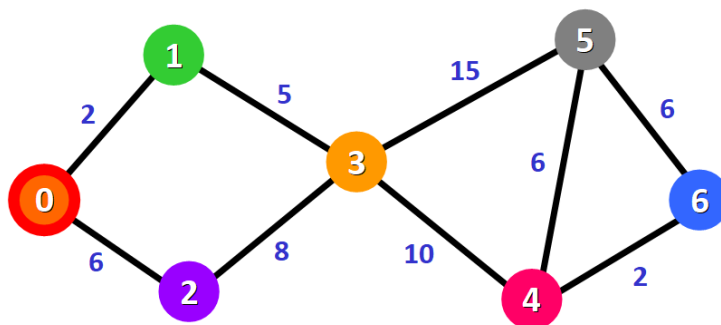
We also have this list (see below) to keep track of the nodes that have not been visited yet (nodes that have not been included in the path):

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

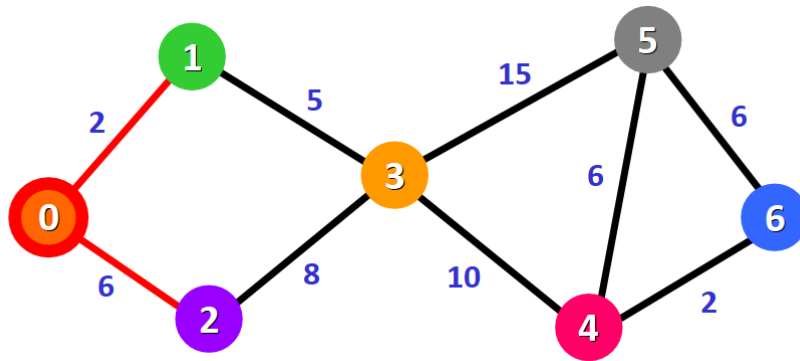
□ **Tip:** Remember that the algorithm is completed once all nodes have been added to the path.

Since we are choosing to start at node 0, we can mark this node as visited. Equivalently, we cross it off from the list of unvisited nodes and add a red border to the corresponding node in diagram:

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



Now we need to start checking the distance from node 0 to its adjacent nodes. As you can see, these are nodes 1 and 2 (see the red edges):



□ **Tip:** This doesn't mean that we are immediately adding the two adjacent nodes to the shortest path. Before adding a node to this path, we need to check if we have found the shortest path to reach it. We are simply making an initial examination process to see the options available.

We need to update the distances from node 0 to node 1 and node 2 with the weights of the edges that connect them to node 0 (the source node). These weights are 2 and 6, respectively:

Distance:

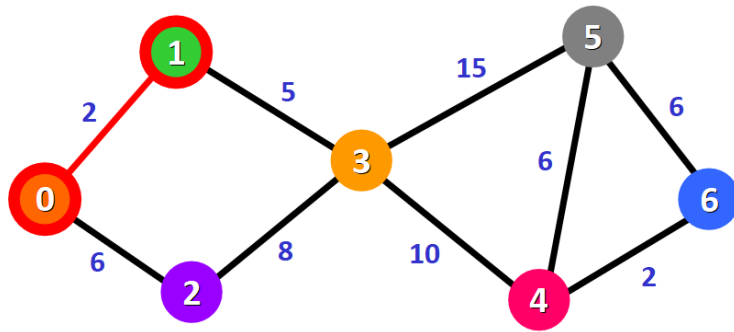
0: 0
 1: ~~∞~~ 2
 2: ~~∞~~ 6
 3: ∞
 4: ∞
 5: ∞
 6: ∞

After updating the distances of the adjacent nodes, we need to:

- 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.

If we check the list of distances, we can see that node 1 has the shortest distance to the source node (a distance of 2), so we add it to the path.

In the diagram, we can represent this with a red edge:



We mark it with a red square in the list to represent that it has been "visited" and that we have found the shortest path to this node:

Distance:

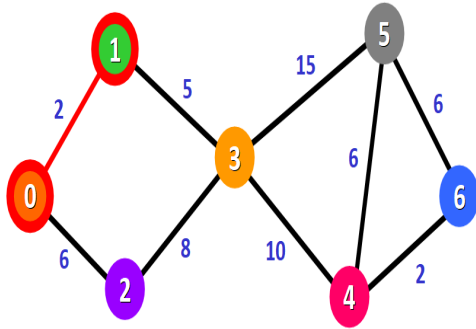
0:	0	
1:	∞	2 ■
2:	∞	6
3:	∞	
4:	∞	
5:	∞	
6:	∞	

We cross it off from the list of unvisited nodes:

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

Now we need to analyze the new adjacent nodes to find the shortest path to reach them. We will only analyze the nodes that are adjacent to the nodes that are already part of the shortest path (the path marked with red edges).

Node 3 and node 2 are both adjacent to nodes that are already in the path because they are directly connected to node 1 and node 0, respectively, as you can see below. These are the nodes that we will analyze in the next step.



Since we already have the distance from the source node to node 2 written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node 3):

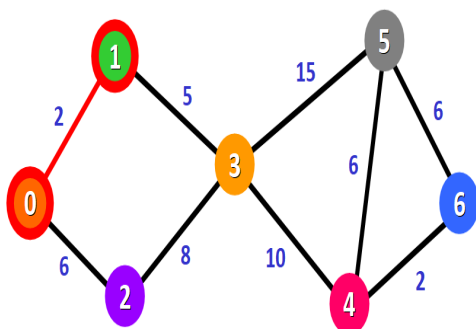
Distance:

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

This distance is **7**. Let's see why.

To find the distance from the source node to another node (in this case, node 3), we add the weights of all the edges that form the shortest path to reach that node:

- **For node 3:** the total distance is **7** because we add the weights of the edges that form the path 0 → 1 → 3 (2 for the edge 0 → 1 and 5 for the edge 1 → 3).



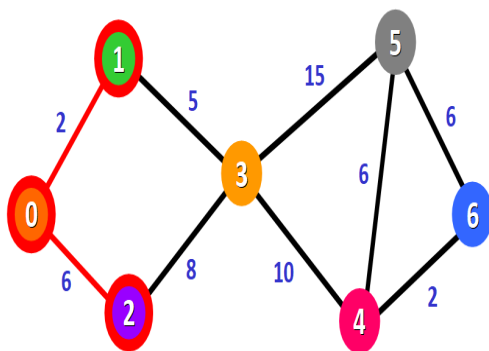
Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path. We must select the **unvisited** node with the shortest (currently known) distance to the source node.

From the list of distances, we can immediately detect that this is node 2 with distance **6**:

Distance:

0:	0	
1:	∞	2 -
2:	∞	6 -
3:	∞	7
4:	∞	
5:	∞	
6:	∞	

We add it to the path graphically with a red border around the node and a red edge:



We also mark it as visited by adding a small red square in the list of distances and crossing it off from the list of unvisited nodes:

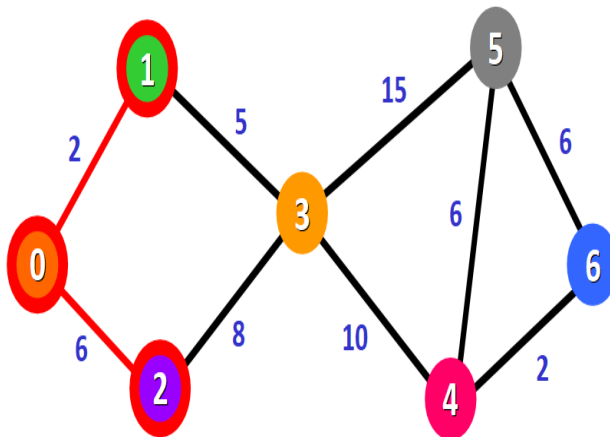
Distance:

0:	0	
1:	∞	2 -
2:	∞	6 -
3:	∞	7
4:	∞	
5:	∞	
6:	∞	

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

Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node 3.

You can see that we have two possible paths $0 \rightarrow 1 \rightarrow 3$ or $0 \rightarrow 2 \rightarrow 3$. Let's see how we can decide which one is the shortest path.



Node 3 already has a distance in the list that was recorded previously (**7**, see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path $0 \rightarrow 1 \rightarrow 3$.

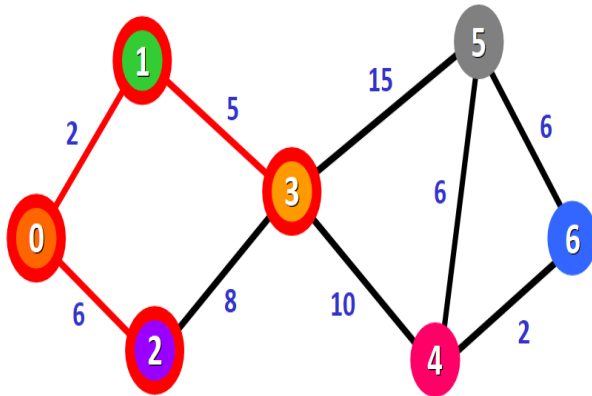
But now we have another alternative. If we choose to follow the path $0 \rightarrow 2 \rightarrow 3$, we would need to follow two edges $0 \rightarrow 2$ and $2 \rightarrow 3$ with weights **6** and **8**, respectively, which represents a total distance of **14**.

Distance:

0: 0
 1: ~~∞~~ 2 ■
 2: ~~∞~~ 6 ■
 3: ~~∞~~ 7 from (5 + 2) vs. 14 from (6 + 8)
 4: ∞
 5: ∞
 6: ∞

Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path $0 \rightarrow 1 \rightarrow 3$. **We only update the distance if the new path is shorter.**

Therefore, we add this node to the path using the first alternative: $0 \rightarrow 1 \rightarrow 3$.



We mark this node as visited and cross it off from the list of unvisited nodes:

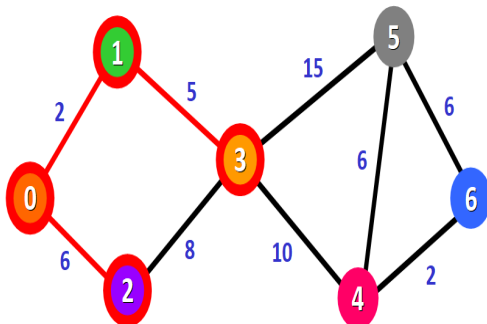
Distance:

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

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

Now we repeat the process again.

We need to check the new adjacent nodes that we have not visited so far. This time, these nodes are node 4 and node 5 since they are adjacent to node 3.



We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

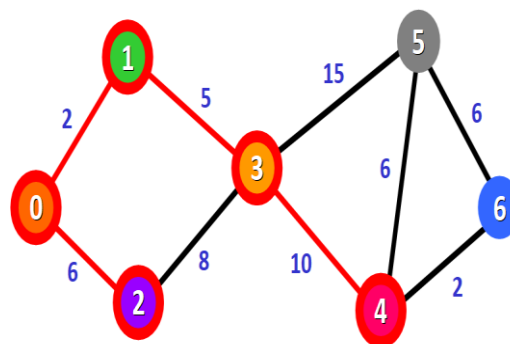
- **For node 4:** the distance is **17** from the path 0 -> 1 -> 3 -> 4.
- **For node 5:** the distance is **22** from the path 0 -> 1 -> 3 -> 5.

□ **Tip:** Notice that we can only consider extending the shortest path (marked in red). We cannot consider paths that will take us through edges that have not been added to the shortest path (for example, we cannot form a path that goes through the edge 2 -> 3).

Distance:

0: 0
 1: ~~∞~~ 2 ■
 2: ~~∞~~ 6 ■
 3: ~~∞~~ 7 ■
 4: ~~∞~~ 17 from (2 + 5 + 10)
 5: ~~∞~~ 22 from (2 + 5 + 15)
 6: ∞

We need to choose which unvisited node will be marked as visited now. In this case, it's node 4 because it has the shortest distance in the list of distances. We add it graphically in the diagram:



We also mark it as "visited" by adding a small red square in the list:

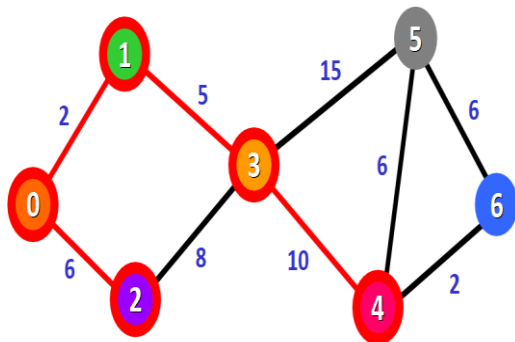
Distance:

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

And we cross it off from the list of unvisited nodes:

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

And we repeat the process again. We check the adjacent nodes: node 5 and node 6. We need to analyze each possible path that we can follow to reach them from nodes that have already been marked as visited and added to the path.



For node 5:

- The first option is to follow the path $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$, which has a distance of **22** from the source node ($2 + 5 + 15$). This distance was already recorded in the list of distances in a previous step.
- The second option would be to follow the path $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, which has a distance of **23** from the source node ($2 + 5 + 10 + 6$).

Clearly, the first path is shorter, so we choose it for node 5.

For node 6:

- The path available is $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6$, which has a distance of **19** from the source node ($2 + 5 + 10 + 2$).

Distance:

0: 0
 1: ~~2~~ ■
 2: ~~6~~ ■
 3: ~~7~~ ■
 4: ~~17~~ ■
 5: ~~22~~ vs. **23** ($2 + 5 + 10 + 6$)
 6: ~~19~~ from ($2 + 5 + 10 + 2$)

We mark the node with the shortest (currently known) distance as visited. In this case, node 6.

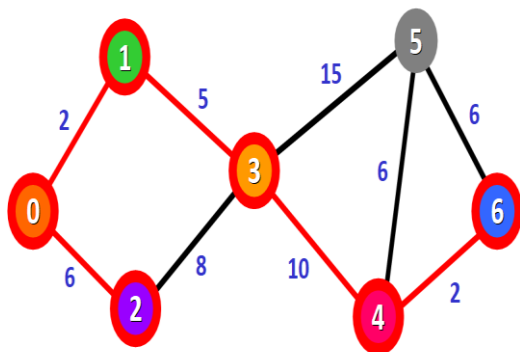
Distance:

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

And we cross it off from the list of unvisited nodes:

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

Now we have this path (marked in red):



Only one node has not been visited yet, node 5. Let's see how we can include it in the path.

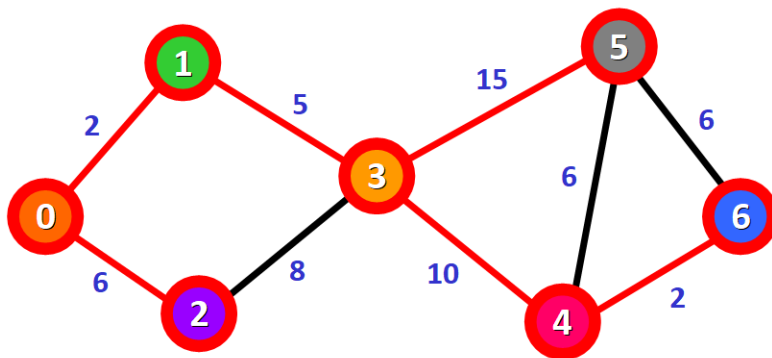
There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

- **Option 1:** 0 → 1 → 3 → 5 with a distance of **22** (2 + 5 + 15).
- **Option 2:** 0 → 1 → 3 → 4 → 5 with a distance of **23** (2 + 5 + 10 + 6).
- **Option 3:** 0 → 1 → 3 → 4 → 6 → 5 with a distance of **25** (2 + 5 + 10 + 2 + 6).

Distance:

0: 0
1: ~~2~~ ■
2: ~~6~~ ■
3: ~~7~~ ■
4: ~~17~~ ■
5: ~~22~~ from (2 + 5 + 15) vs. 23 from (2 + 5 + 10 + 6) vs. 25 from (2 + 5 + 10 + 2 + 6)
6: ~~19~~ ■

We select the shortest path: 0 -> 1 -> 3 -> 5 with a distance of **22**.



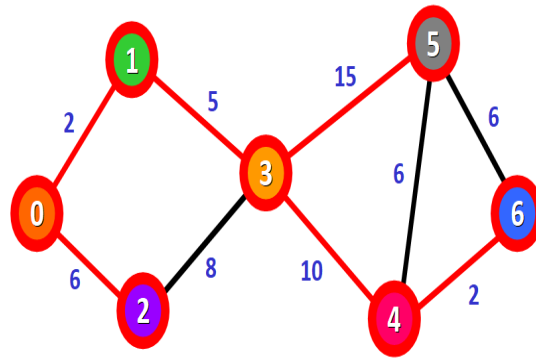
We mark the node as visited and cross it off from the list of unvisited nodes:

Distance:

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

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

And voilà! We have the final result with the shortest path from node 0 to each node in the graph.



In the diagram, the red lines mark the edges that belong to the shortest path. You need to follow these edges to follow the shortest path to reach a given node in the graph starting from node 0.

For example, if you want to reach node 6 starting from node 0, you just need to follow the red edges and you will be following the shortest path 0 -> 1 -> 3 -> 4 -> 6 automatically.