https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/

Welcome! If you've always wanted to learn and understand Dijkstra's algorithm, then this article is for you. You will see how it works behind the scenes with a step-by-step graphical explanation.

You will learn:

- Basic Graph Concepts (a quick review).
- What Dijkstra's Algorithm is used for.
- How it works behind the scenes with a step-by-step example.

Let's begin. ✨

🔹 Introduction to Graphs

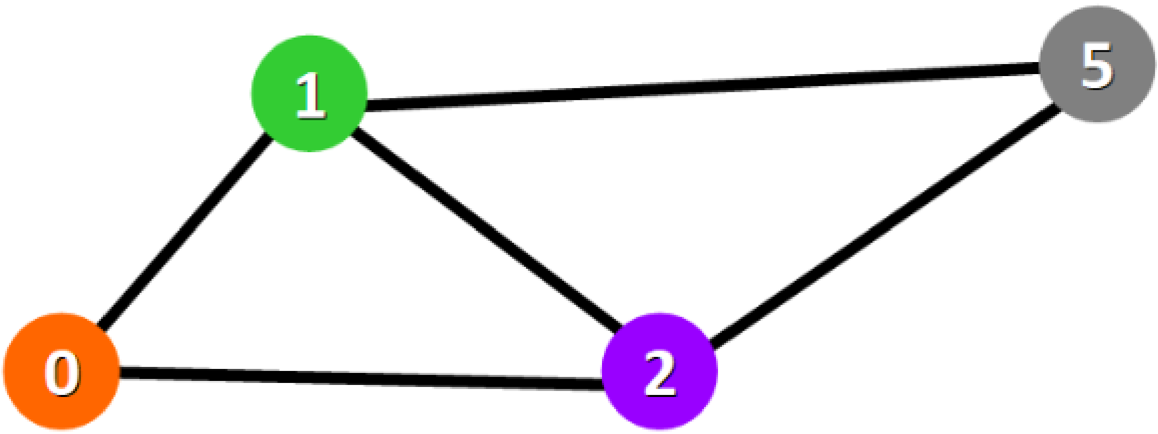
Let's start with a brief introduction to graphs.

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:

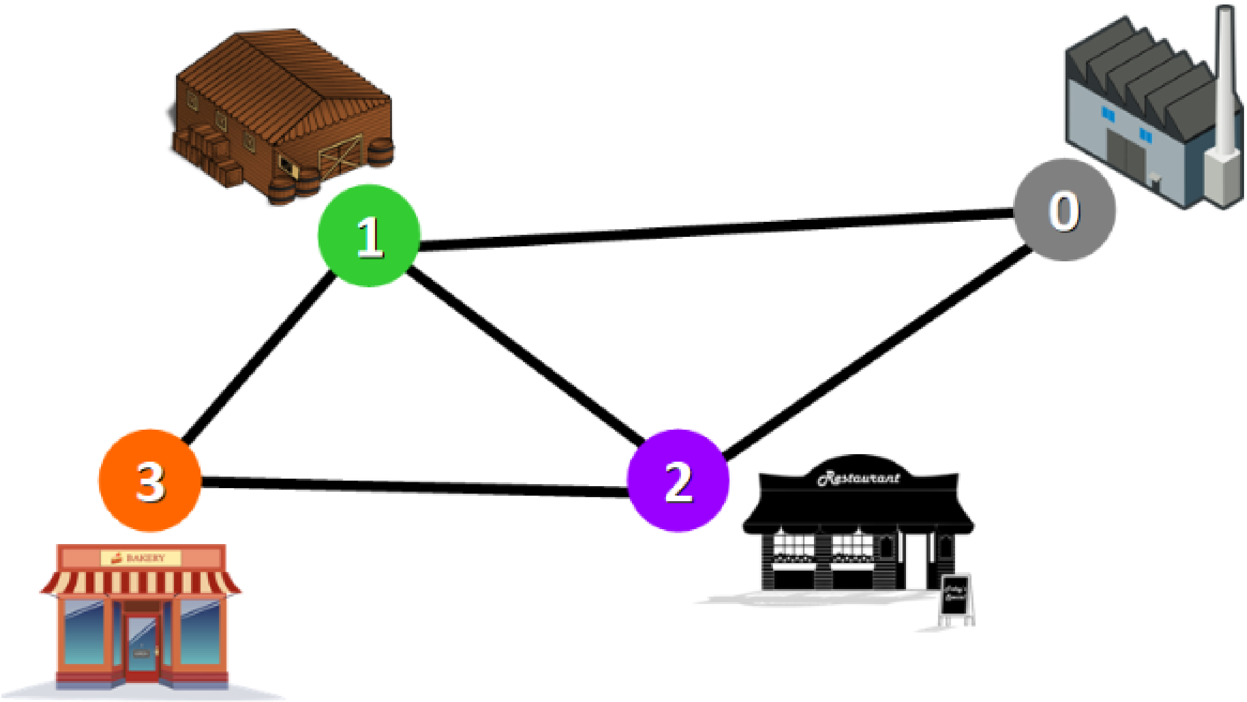


Nodes are represented with colored circles and **edges** are represented with lines that connect these circles.

💡 **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).

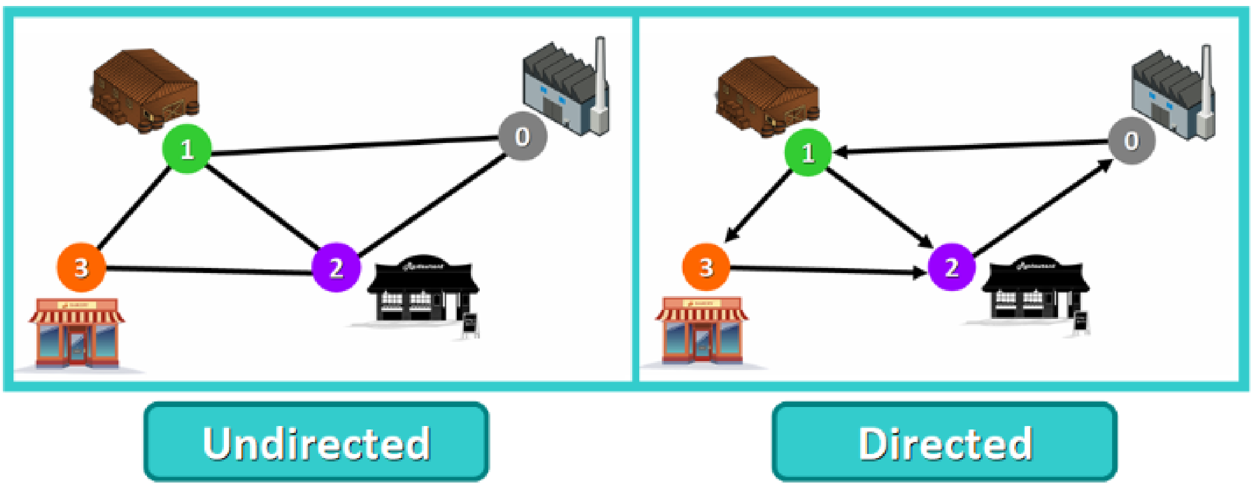


Network represented with a graph

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.

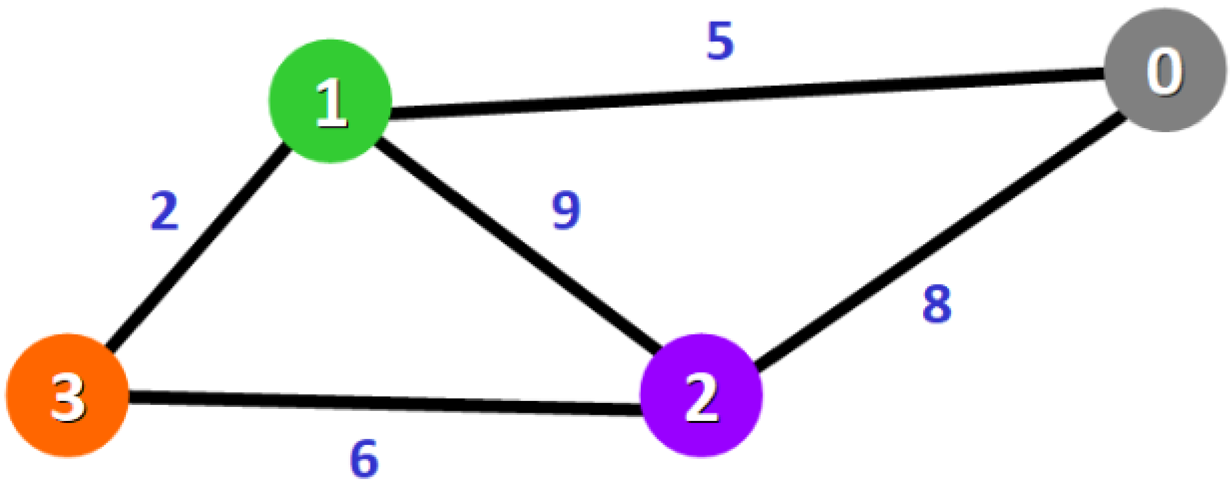


💡 **Tip:** in this article, we will work with **undirected** graphs.

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.



◆ 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

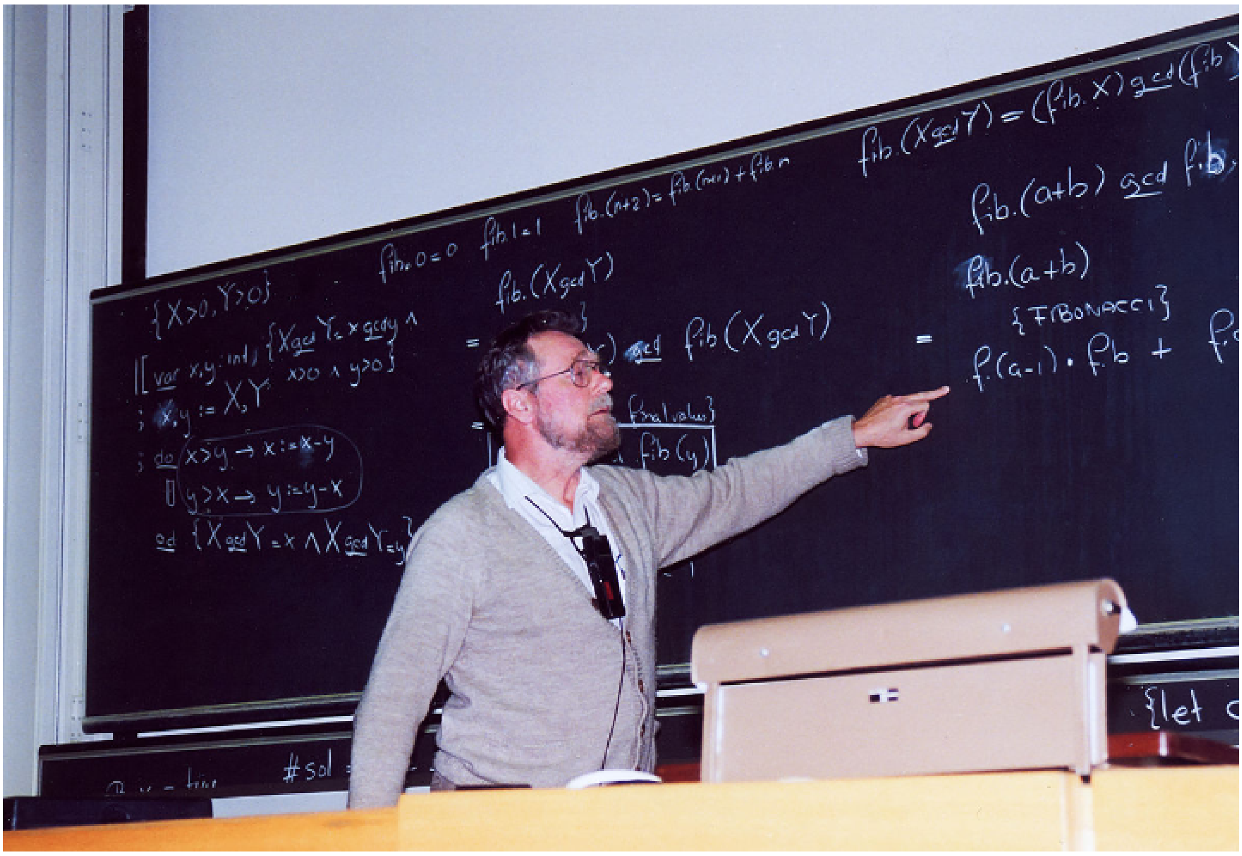
With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree.

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

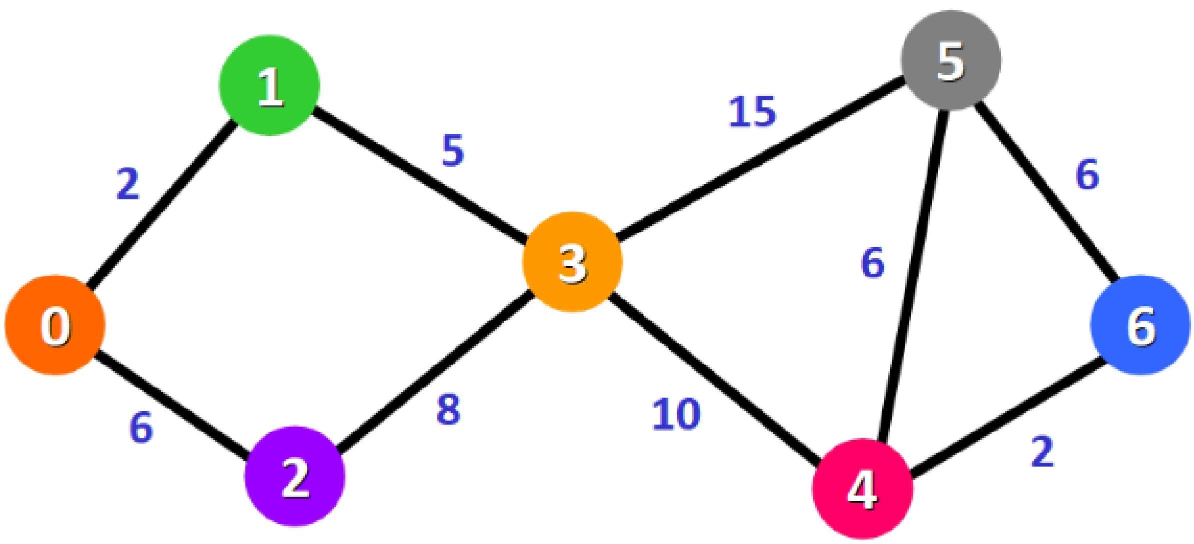
Dijkstra's Algorithm can only work with graphs that have positive weights. This is because, during the process, the weights of the edges have to be added to find the shortest path.

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 a step-by-step example.

We have this graph:



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: ∞

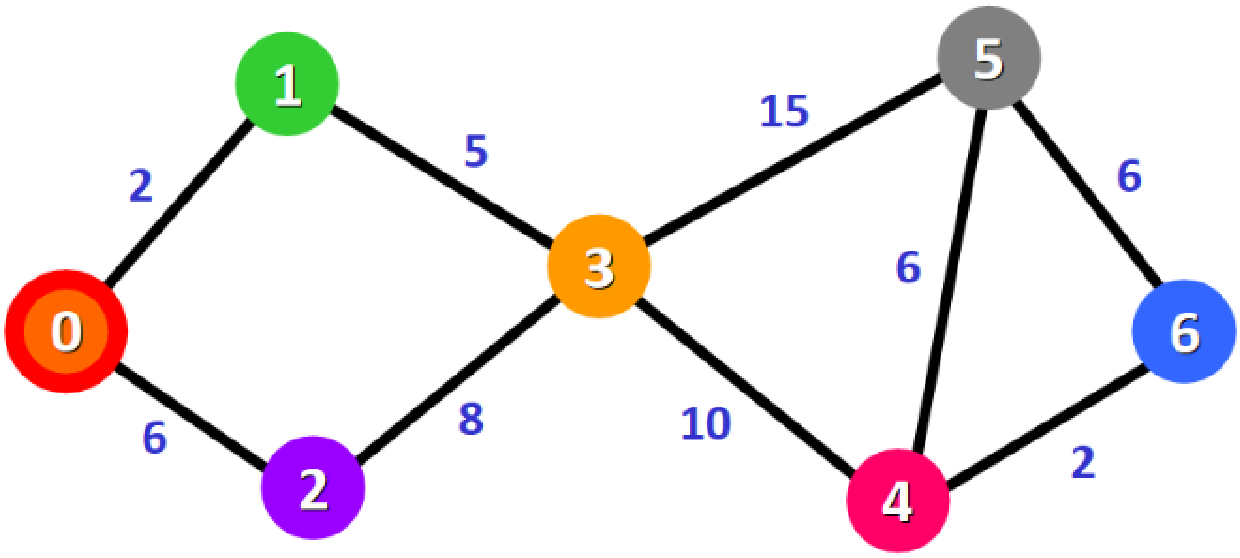
We also have this list (see below) to keep track of the nodes that have not been visited yet (nodes that have not been included in the path):

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

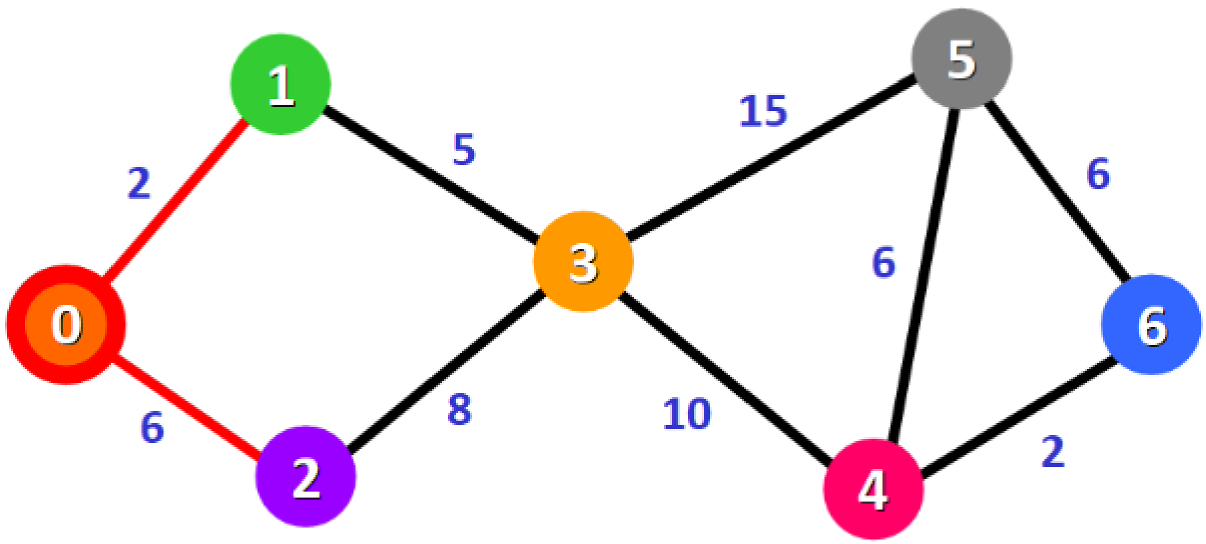
💡 **Tip:** Remember that the algorithm is completed once all nodes have been added to the path.

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:

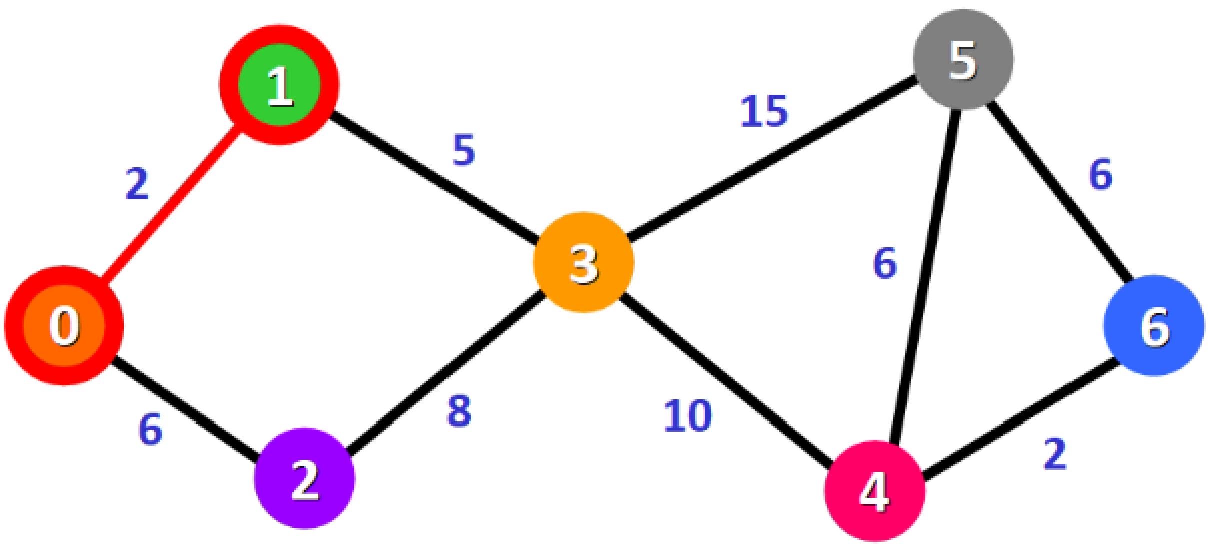
0: 0
1: $\cancel{\infty}$ 2
2: $\cancel{\infty}$ 6
3: ∞
4: ∞
5: ∞
6: ∞

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:

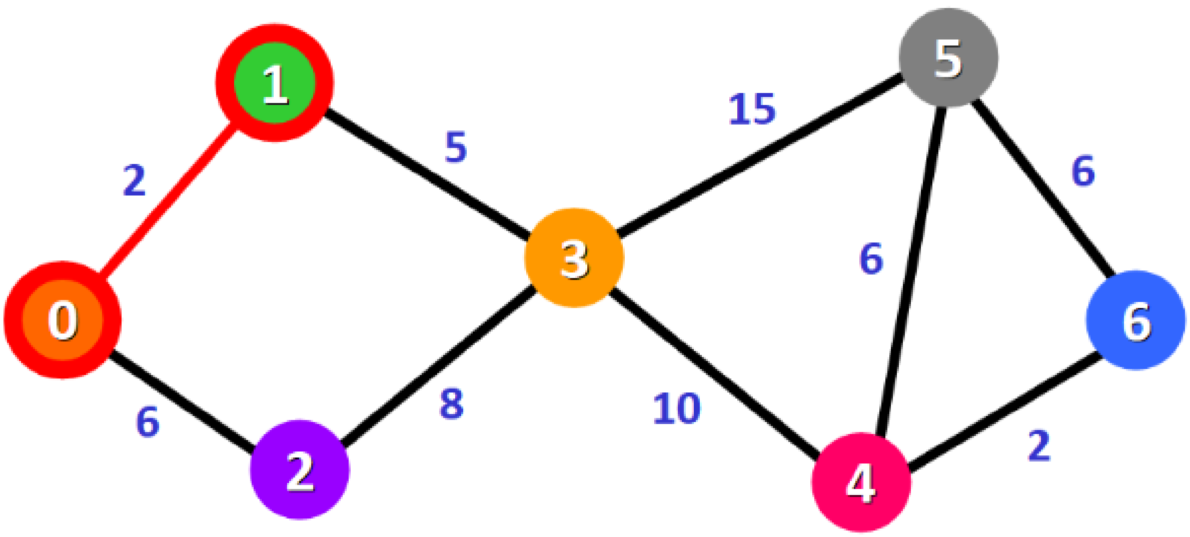
0: 0
1: ~~∞~~ 2
2: ~~∞~~ 6
3: ∞
4: ∞
5: ∞
6: ∞

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):

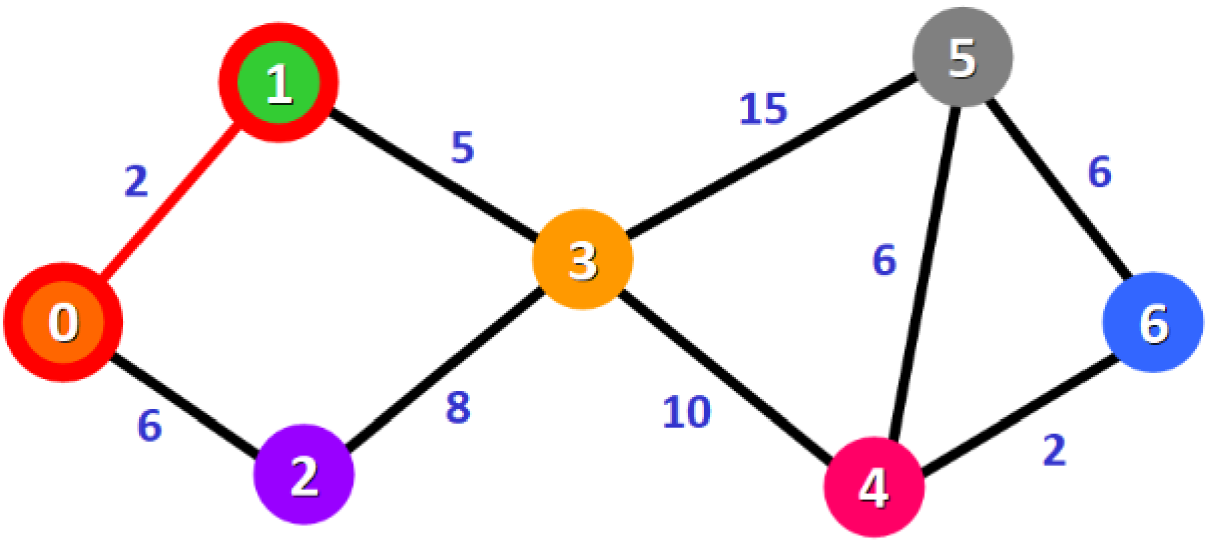
Distance:

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).



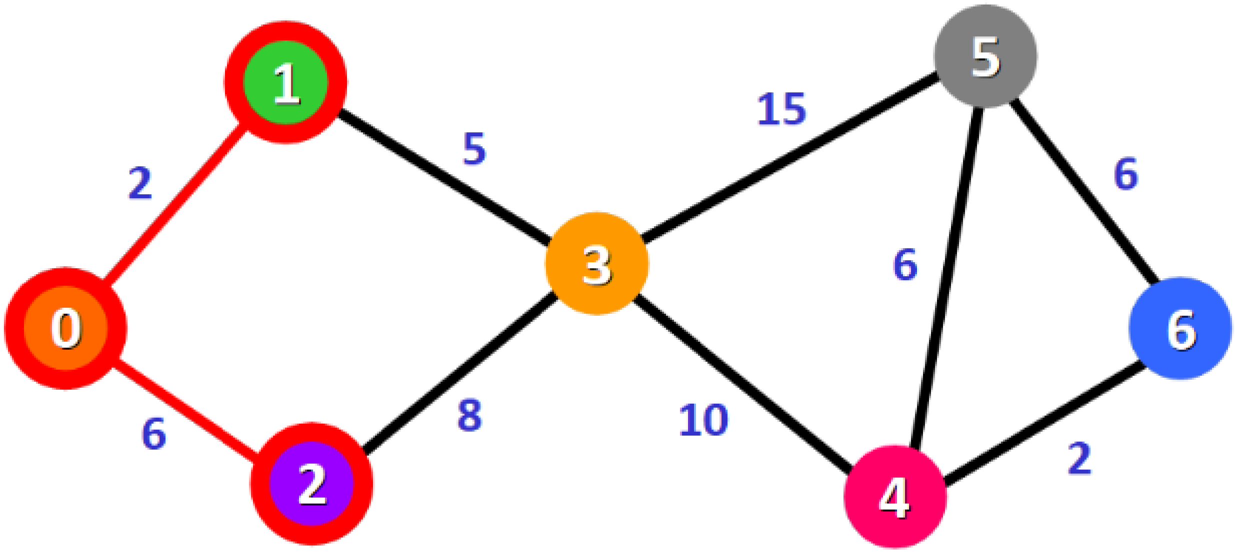
Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path. We must select the **unvisited** node with the shortest (currently known) distance to the source node.

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:

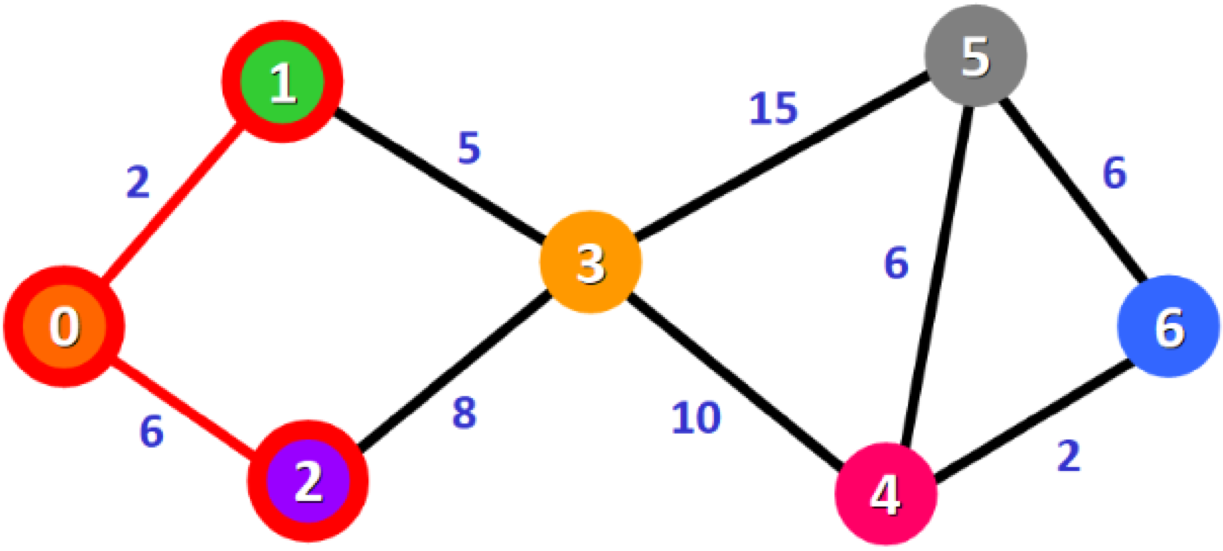
Distance:

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

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

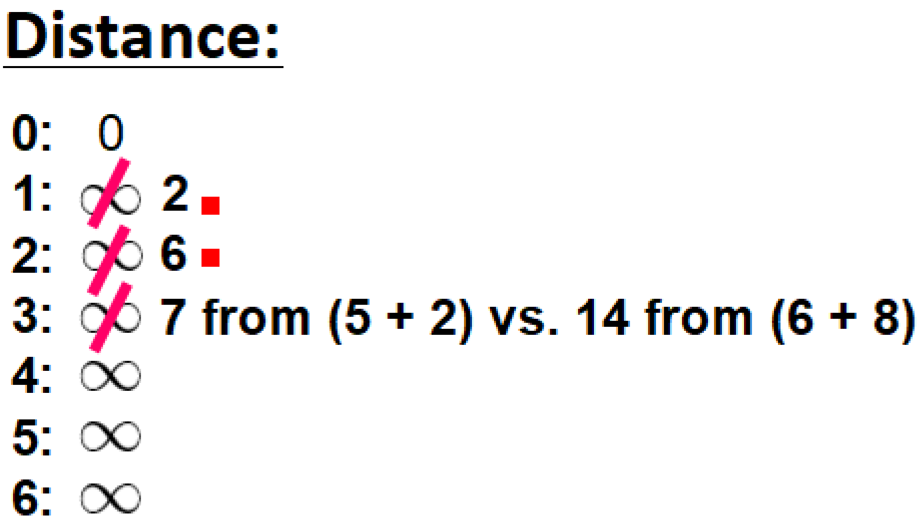
Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node 3.

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



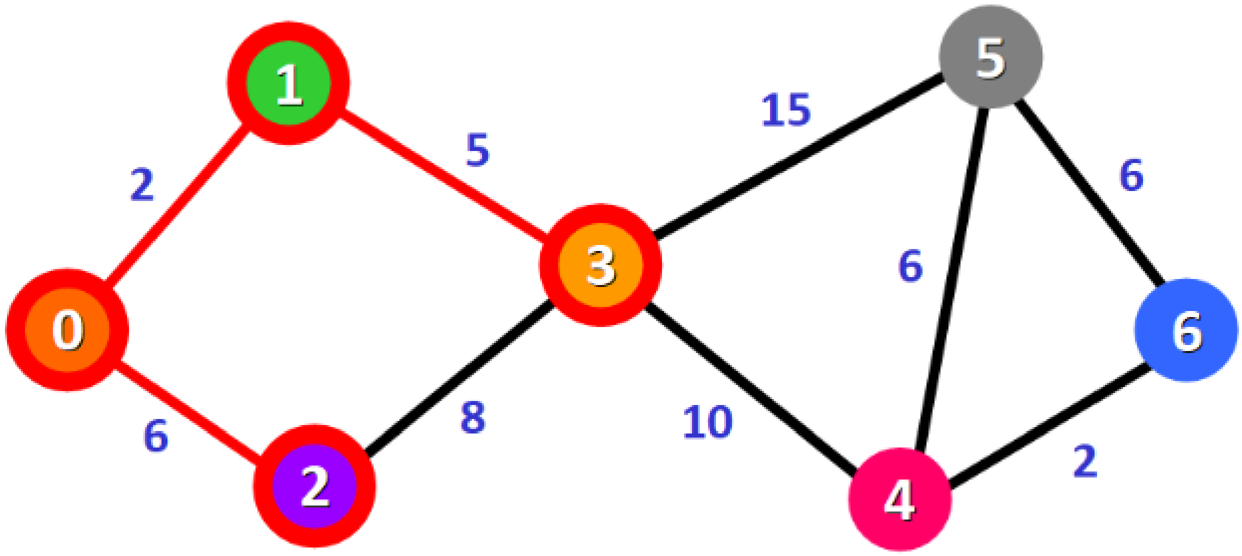
Node 3 already has a distance in the list that was recorded previously (**7,** see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path 0 -> 1 -> 3.

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



Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path 0 -> 1 -> 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 -> 1 -> 3.



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

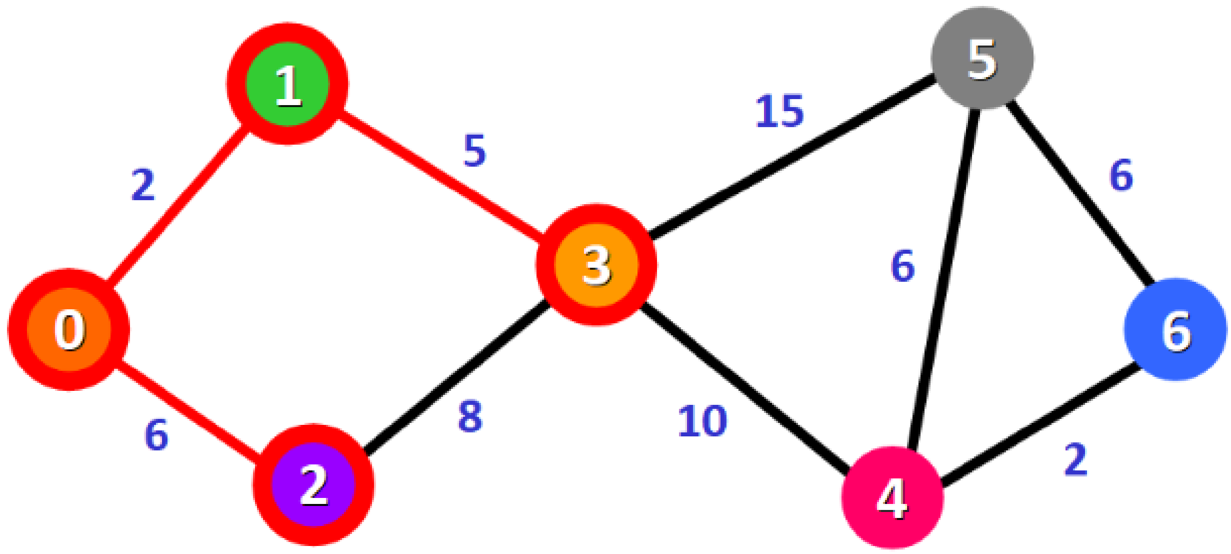
Distance:

0: 0
1: $\cancel{\infty}$ **2**
2: $\cancel{\infty}$ **6**
3: $\cancel{\infty}$ **7**
4: ∞
5: ∞
6: ∞

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

Now we repeat the process again.

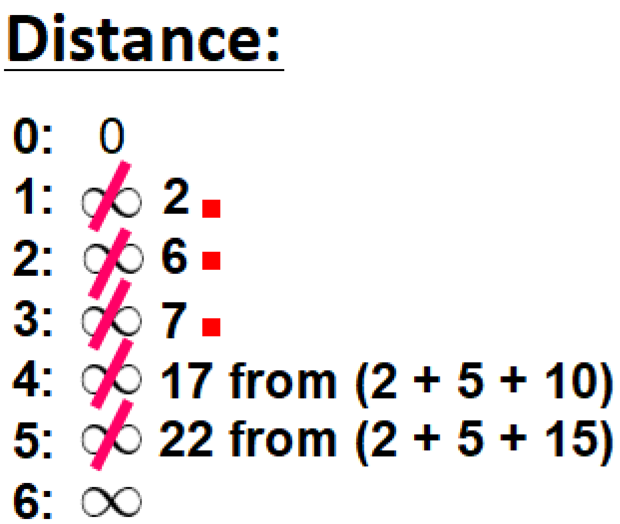
We need to check the new adjacent nodes that we have not visited so far. This time, these nodes are node 4 and node 5 since they are adjacent to node 3.



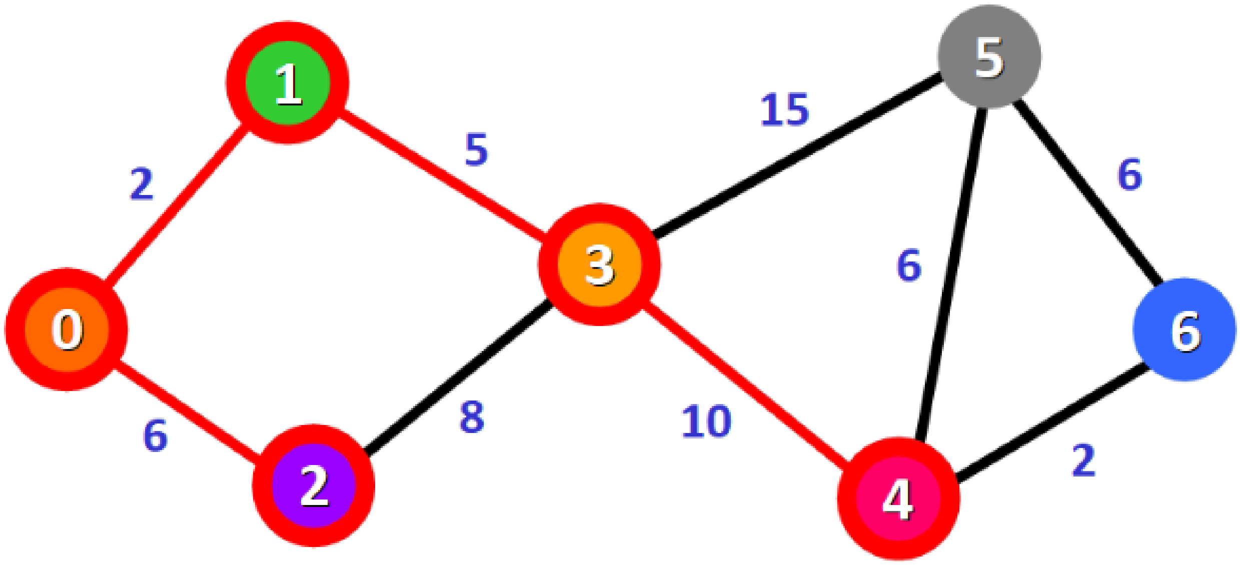
We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

- 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).**



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:

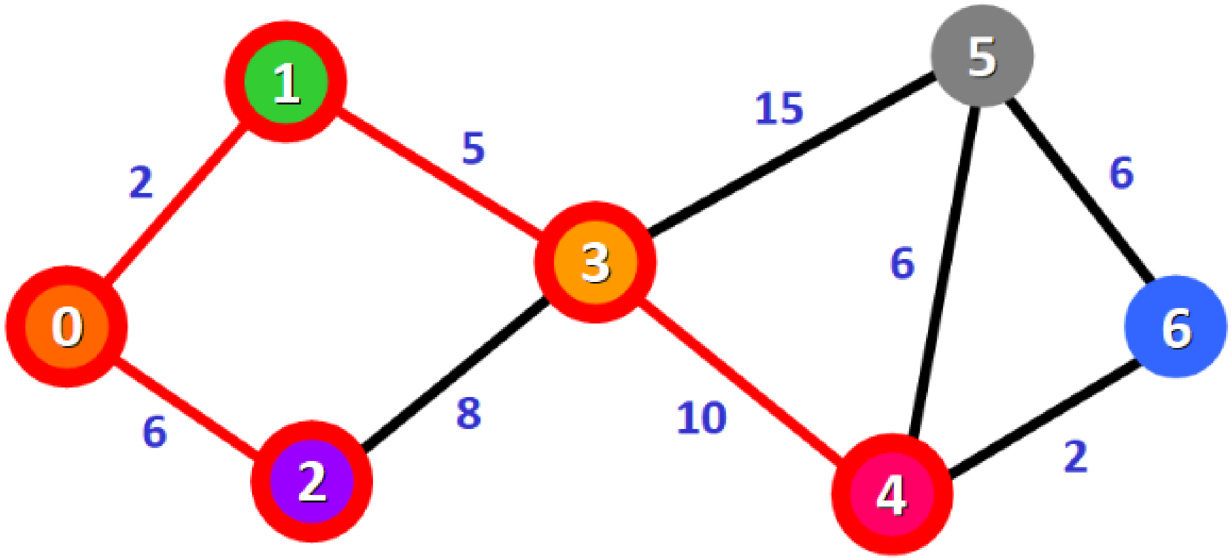
Distance:

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 -> 1 -> 3 -> 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 -> 1 -> 3 -> 4 -> 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 -> 1 -> 3 -> 4 -> 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.

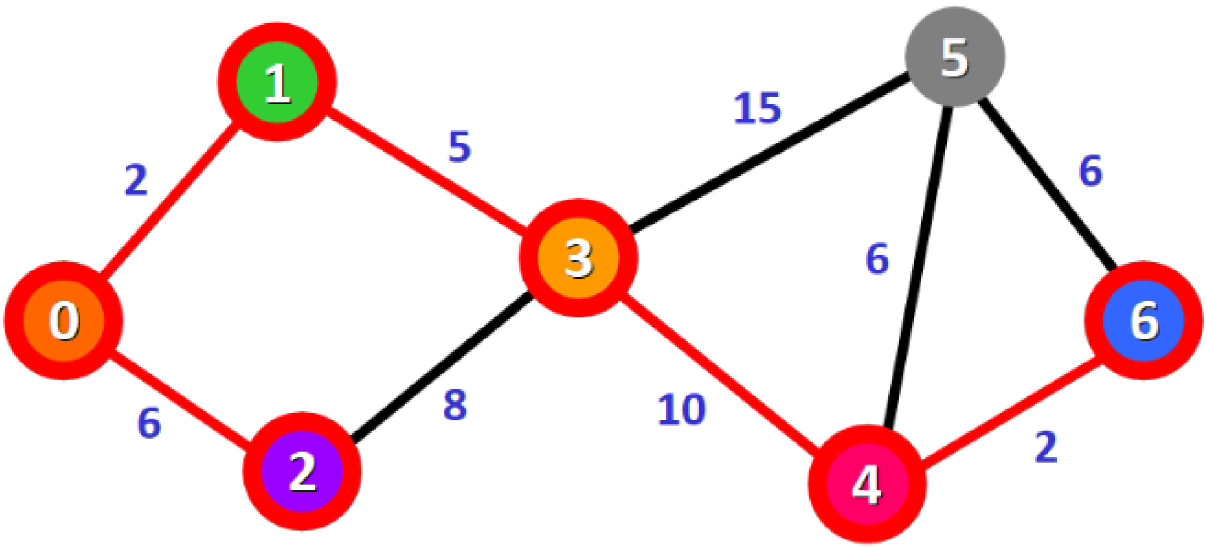
Distance:

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

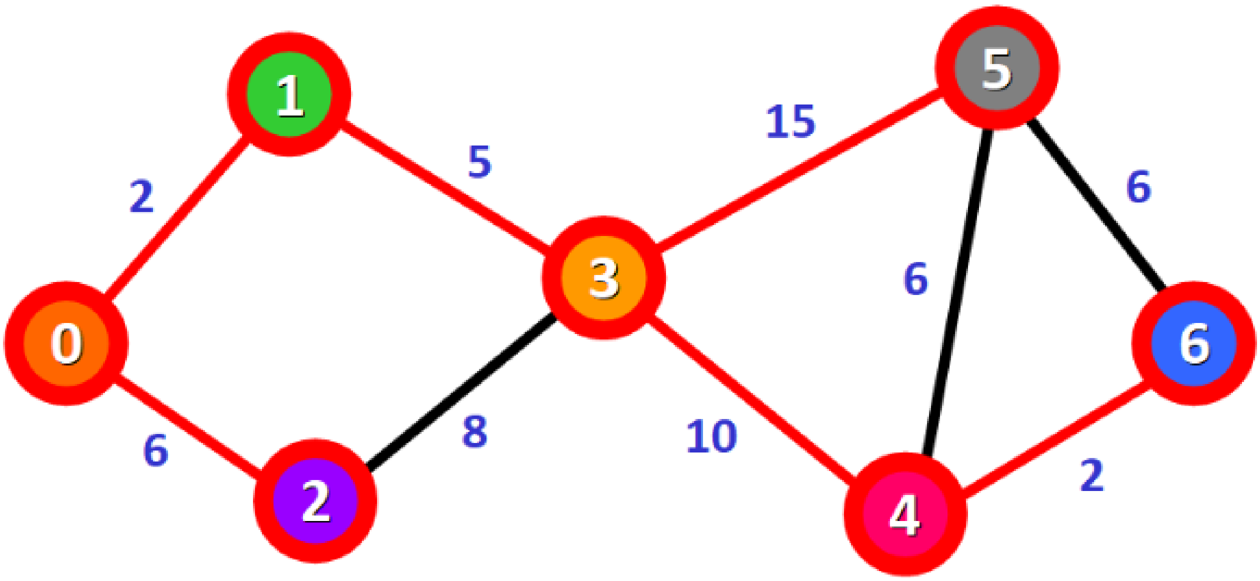
There are three different paths that we can take to reach node 5 from the nodes that have been added to the path:

- 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**.



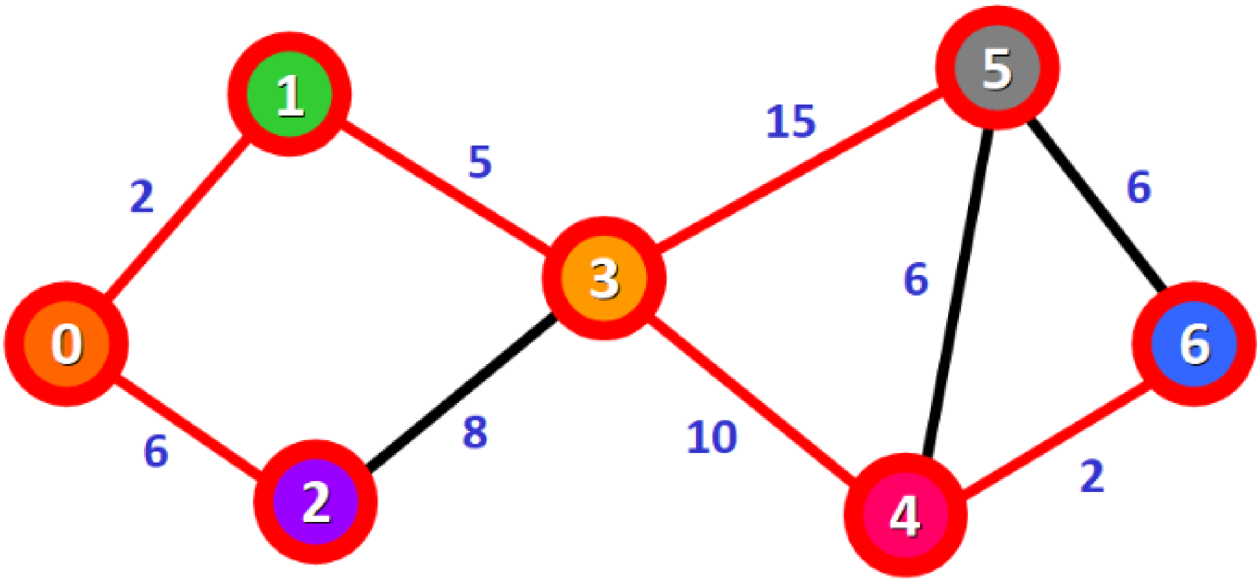
We mark the node as visited and cross it off from the list of unvisited nodes:

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.

◆ In Summary

- Graphs are used to model connections between objects, people, or entities. They have two main elements: nodes and edges. Nodes represent objects and edges represent the connections between these objects.
- Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph.
- This algorithm uses the weights of the edges to find the path that minimizes the total distance (weight) between the source node and all other nodes.

What's the difference between Dijkstra and BFS ?

Dijkstra's algorithm and Breadth-First Search (BFS) are both algorithms used for graph traversal and pathfinding, but they serve different purposes and have different characteristics:

1. **Purpose:**
    - Dijkstra's algorithm is used to find the shortest path between a single source vertex and all other vertices in a weighted graph with non-negative edge weights. It's commonly used in routing and network optimization problems.
    - BFS is used to traverse or search a graph or tree data structure level by level, starting from a given source vertex. It's used to find the shortest path in an unweighted graph or to explore all vertices reachable from a given source vertex.
2. **Algorithm:**

    - Dijkstra's algorithm is a greedy algorithm that iteratively selects the vertex with the smallest tentative distance from the source vertex and updates the distances to its adjacent vertices accordingly. It maintains a priority queue (min heap) to efficiently select the next vertex with the smallest tentative distance.
    - BFS explores vertices in layers, starting from the source vertex and visiting all of its neighbors before moving to the next level of vertices. It typically uses a queue data structure to keep track of the vertices to visit next.
3. **Edge Weights:**
    - Dijkstra's algorithm works with graphs where edges have non-negative weights. It assumes that smaller edge weights represent shorter distances.
    - BFS can work with both unweighted and weighted graphs, but it doesn't consider edge weights during traversal. It simply explores vertices in increasing order of distance from the source.
4. **Optimality:**
    - Dijkstra's algorithm guarantees finding the shortest path from the source vertex to all other vertices in the graph, given that the graph satisfies the requirements (non-negative edge weights).
    - BFS finds the shortest path in terms of the number of edges, not necessarily the shortest path in terms of edge weights. In an unweighted graph, the shortest path found by BFS is always optimal.

In summary, Dijkstra's algorithm is specifically designed for finding the shortest path in weighted graphs with non-negative edge weights, while BFS is a more general-purpose algorithm for traversing graphs and finding shortest paths in unweighted graphs.

We don't have "visited" array to record if a node visited or not in Dijkstra's algorithm, then how we avoid duplicate visit of a node ?

In Dijkstra's algorithm, we ensure that each node is visited only once by maintaining a data structure called a priority queue (often implemented using a min-heap) to store the vertices along with their tentative distances from the source vertex. This priority queue helps us select the next vertex with the smallest tentative distance efficiently.

Here's how Dijkstra's algorithm avoids duplicate visits of nodes:

1. **Initialization:** Initialize the priority queue with the source vertex and set its tentative distance to 0. Initialize the tentative distance of all other vertices to infinity.
2. **Main Loop:** While the priority queue is not empty, repeatedly extract the vertex with the smallest tentative distance from the priority queue. Let's call this vertex **current**.
3. **Relaxation:** For each neighbor **neighbor** of **current**, calculate its tentative distance from the source vertex through **current** (tentative distance of **neighbor** = tentative distance of **current** + weight of edge **(current, neighbor)**). If this new tentative distance is smaller than the current tentative distance of **neighbor**, update **neighbor**'s tentative distance and update its entry in the priority queue.

4. **Duplicate Avoidance:** **When extracting a vertex from the priority queue, ensure that the vertex has not been processed before. This is done by checking whether the vertex's tentative distance is already final. If the tentative distance is final, it means the vertex has been processed, and there's no need to process it again. (Note: minPQ + Dijkstra relaxation inequality help on check if we have to visit or revisit the node)**
5. **Finalization:** Once all vertices have been processed (i.e., their tentative distances are final), terminate the algorithm.

By maintaining the priority queue and updating the tentative distances of vertices during relaxation steps, Dijkstra's algorithm ensures that each vertex is visited only once, and it's only processed when its tentative distance is not yet final. This effectively avoids duplicate visits of nodes while traversing the graph.