**Graphs**

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

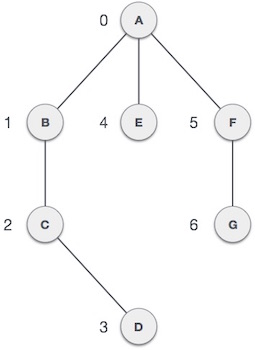
V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

**Graph Data Structure**

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

* **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
* **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
* **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
* **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



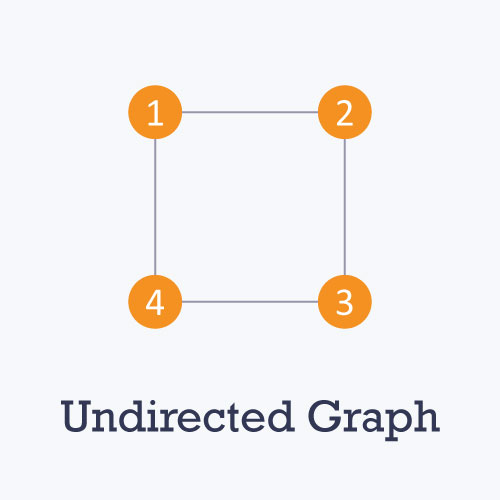
**Basic Operations**

Following are basic primary operations of a Graph −

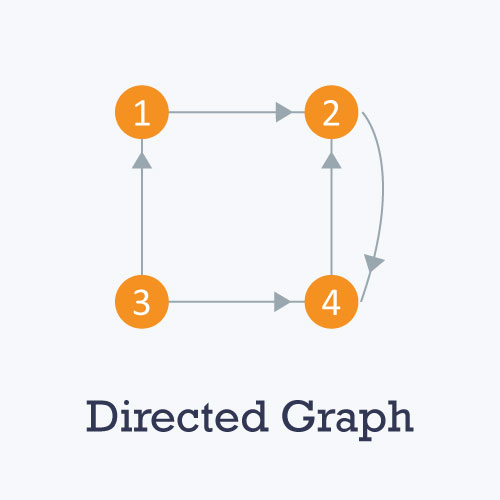
* **Add Vertex** − Adds a vertex to the graph.
* **Add Edge** − Adds an edge between the two vertices of the graph.
* **Display Vertex** − Displays a vertex of the graph.

**Types of graphs**

* Undirected: An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.

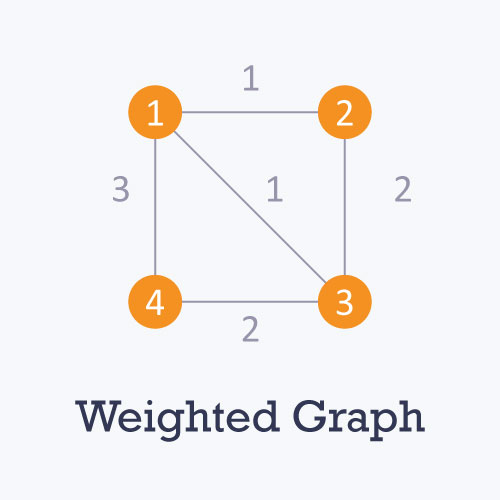


* Directed: A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.



* Weighted: In a weighted graph, each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it. If you want to go from vertex 1 to vertex 3, you can take one of the following 3 paths:
  + 1 -> 2 -> 3
  + 1 -> 3
  + 1 -> 4 -> 3

Therefore the total cost of each path will be as follows: - The total cost of 1 -> 2 -> 3 will be (1 + 2) i.e. 3 units - The total cost of 1 -> 3 will be 1 unit - The total cost of 1 -> 4 -> 3 will be (3 + 2) i.e. 5 units

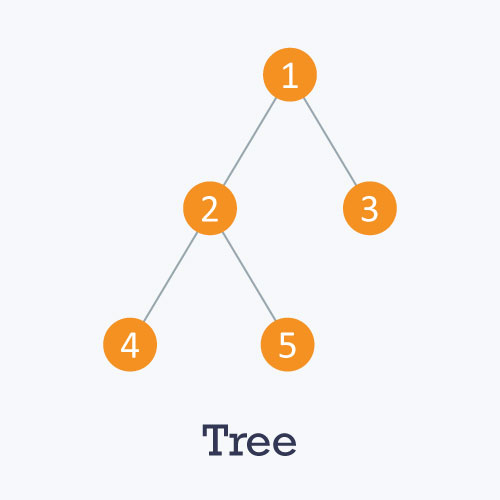


* Cyclic: A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex. That path is called a cycle. An acyclic graph is a graph that has no cycle.

A **tree** is an undirected graph in which any two vertices are connected by only one path. A tree is an acyclic graph and has N - 1 edges where N is the number of vertices. Each node in a graph may have one or multiple parent nodes. However, in a tree, each node (except the root node) comprises exactly one parent node.

***Note***: A root node has no parent.

A tree cannot contain any cycles or self loops, however, the same does not apply to graphs.



**Graph representation**

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

**Adjacency matrix**

An adjacency matrix is a **VxV** binary matrix **A**. Element Ai,j is 1 if there is an edge from vertex i to vertex j else Ai,j is 0.

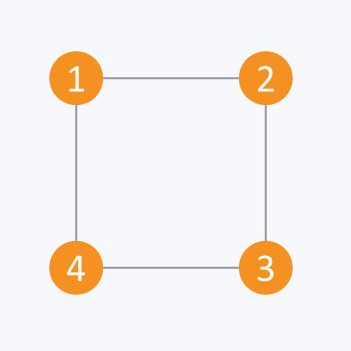
***Note***: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in Ai,j, the weight or cost of the edge will be stored.

In an undirected graph, if Ai,j = 1, then Aj,i = 1. In a directed graph, if Ai,j = 1, then Aj,i may or may not be 1.

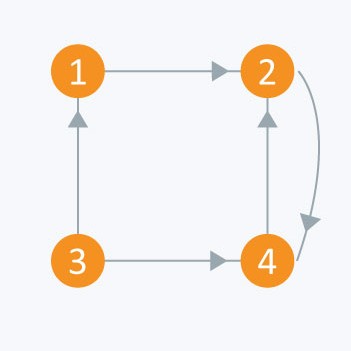
Adjacency matrix provides **constant time access** **(O(1) )** to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is **O(**V2**)**.

The adjacency matrix of the following graph is:  
**i/j** : **1 2 3 4**  
**1** : 0 1 0 1  
**2** : 1 0 1 0  
**3** : 0 1 0 1  
**4** : 1 0 1 0



The adjacency matrix of the following graph is:

**i/j**: **1 2 3 4**  
**1** : 0 1 0 0  
**2** : 0 0 0 1  
**3** : 1 0 0 1  
**4** : 0 1 0 0



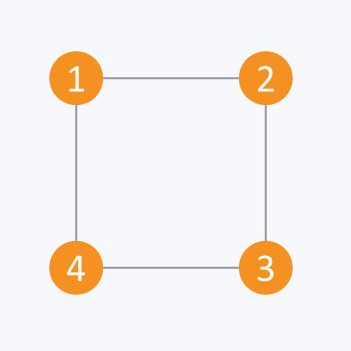
**Adjacency list**

The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array **Ai** is a list, which contains all the vertices that are adjacent to vertex i.

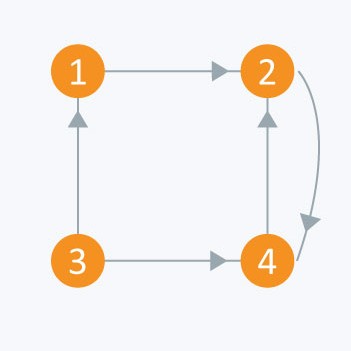
For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list Ai then vertex i will be in list Aj.

The space complexity of adjacency list is **O(V + E)** because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.

**Note:** A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.



Consider the same undirected graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2 → 4  
A2 → 1 → 3  
A3 → 2 → 4  
A4 → 1 → 3  


Consider the same directed graph from an adjacency matrix. The adjacency list of the graph is as follows:

A1 → 2  
A2 → 4  
A3 → 1 → 4  
A4 → 2

**Difference Between BFS and DFS**

Difference between BFS and DFS is that BFS proceeds level by level while DFS follows first a path form the starting to the ending node (vertex), then another path from the start to end, and so on until all nodes are visited. Furthermore, BFS uses the queue for storing the nodes whereas DFS uses the stack for traversal of the nodes.

BFS and DFS are the traversing methods used in searching a graph. Graph traversal is the process of visiting all the nodes of the graph. A graph is a group of Vertices ‘V’ and Edges ‘E’ connecting to the vertices.

**Comparison Chart**

|  |  |
| --- | --- |
| **BFS** | **DFS** |
| BFS finds the shortest path to the destination. | DFS goes to the bottom of a subtree, then backtracks. |
| The full form of BFS is Breadth-First Search. | The full form of DFS is Depth First Search. |
| It uses a queue to keep track of the next location to visit. | It uses a stack to keep track of the next location to visit. |
| BFS traverses according to tree level. | DFS traverses according to tree depth. |
| It is implemented using FIFO list. | It is implemented using LIFO list. |
| It requires more memory as compare to DFS. | It requires less memory as compare to BFS. |
| This algorithm gives the shallowest path solution. | This algorithm doesn't guarantee the shallowest path solution. |
| There is no need of backtracking in BFS. | There is a need of backtracking in DFS. |
| You can never be trapped into finite loops. | You can be trapped into infinite loops. |
| If you do not find any goal, you may need to expand many nodes before the solution is found. | If you do not find any goal, the leaf node backtracking may occur. |

Definition of BFS

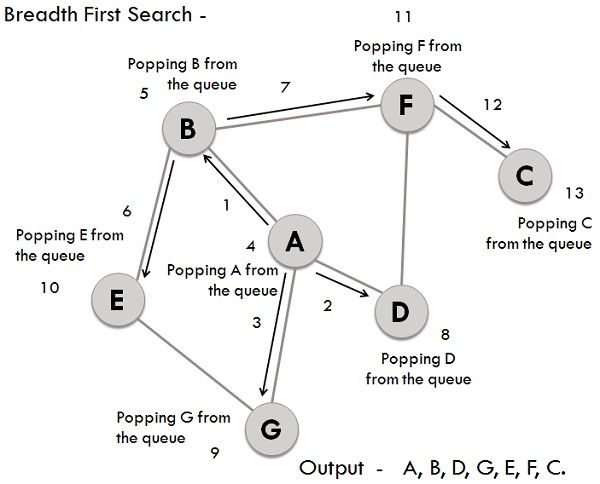
**Breadth First Search (BFS)** is the traversing method used in graphs. It uses a queue for storing the visited vertices. In this method the emphasize is on the vertices of the graph, one vertex is selected at first then it is visited and marked. The vertices adjacent to the visited vertex are then visited and stored in the queue sequentially.

Similarly, the stored vertices are then treated one by one, and their adjacent vertices are visited. A node is fully explored before visiting any other node in the graph, in other words, it traverses shallowest unexplored nodes first.

Example

We have a graph whose vertices are A, B, C, D, E, F, G. Considering A as starting point. The steps involved in the process are:

* Vertex A is expanded and stored in the queue.
* Vertices B, D and G successors of A, are expanded and stored in the queue meanwhile Vertex A removed.
* Now B at the front end of the queue is removed along with storing its successor vertices E and F.
* Vertex D is at the front end of the queue is removed, and its connected node F is already visited.
* Vertex G is removed from the queue, and it has successor E which is already visited.
* Now E and F are removed from the queue, and its successor vertex C is traversed and stored in the queue.
* At last C is also removed and the queue is empty which means we are done.
* The generated Output is – A, B, D, G, E, F, C.

Applications

BFS is also better at finding the **shortest path** in the graph could be seen as a network.

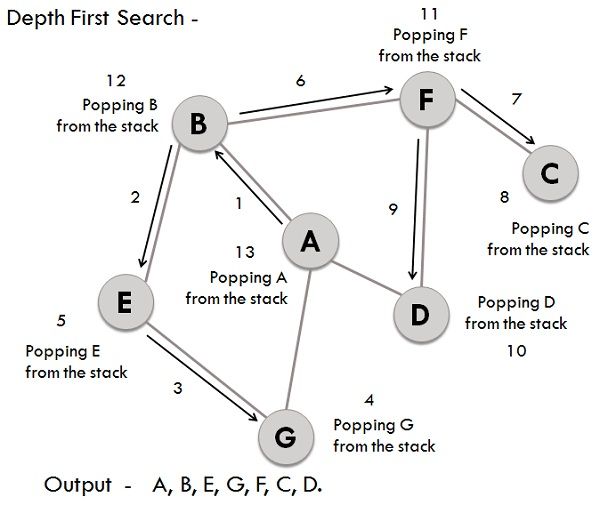
Definition of DFS

**Depth First Search (DFS)** traversing method uses the stack for storing the visited vertices. DFS is the edge based method and works in the recursive fashion where the vertices are explored along a path (edge). The exploration of a node is suspended as soon as another unexplored node is found and the deepest unexplored nodes are traversed at foremost. DFS traverse/visit each vertex exactly once and each edge is inspected exactly twice.

Example

Similar to BFS lets take the same graph for performing DFS operations, and the involved steps are:

* Considering A as the starting vertex which is explored and stored in the stack.
* B successor vertex of A is stored in the stack.
* Vertex B have two successors E and F, among them alphabetically E is explored first and stored in the stack.
* The successor of vertex E, i.e., G is stored in the stack.
* Vertex G have two connected vertices, and both are already visited, so G is popped out from the stack.
* Similarly, E s also removed.
* Now, vertex B is at the top of the stack, its another node(vertex) F is explored and stored in the stack.
* Vertex F has two successor C and D, between them C is traversed first and stored in the stack.
* Vertex C only have one predecessor which is already visited, so it is removed from the stack.
* Now vertex D connected to F is visited and stored in the stack.
* As vertex D doesn’t have any unvisited nodes, therefore D is removed.
* Similarly, F, B and A are also popped.
* The generated output is – A, B, E, G, F, C, D.

Application

The applications of DFS includes the inspection of **two edge connected** graph, **strongly connected** graph, **acyclic graph**, and **topological order**.

A graph is called two edges connected if and only if it remains connected even if one of its edges is removed. This application is very useful, in computer networks where the failure of one link in the network will not affect the remaining network, and it would be still connected.

Strongly connected graph is a graph in which there must exist a path between ordered pair of vertices. DFS is used in the directed graph for searching the path between every ordered pair of vertices. DFS can easily solve connectivity problems.

Key Differences Between BFS and DFS

1. BFS is vertex-based algorithm while DFS is an edge-based algorithm.
2. Queue data structure is used in BFS. On the other hand, DFS uses stack or recursion.
3. Memory space is efficiently utilized in DFS while space utilization in BFS is not effective.
4. BFS is optimal algorithm while DFS is not optimal.
5. DFS constructs narrow and long trees. As against, BFS constructs wide and short tree.

**Conclusion**

BFS and DFS, both of the graph searching techniques have similar running time but different space consumption, DFS takes linear space because we have to remember single path with unexplored nodes, while BFS keeps every node in memory.

DFS yields deeper solutions and is not optimal, but it works well when the solution is dense whereas BFS is optimal which searches the optimal goal at first.

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.

**Dijkstra's Algorithm**

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

**Algorithm Steps:**

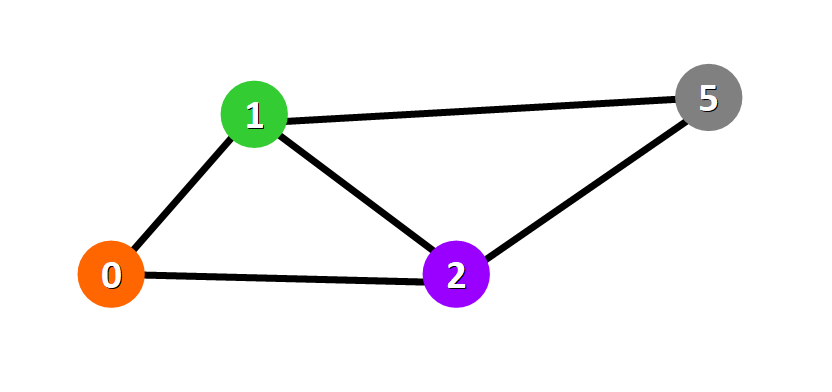
* Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
* Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
* Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
* Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex  
  with the new distance to the priority queue.
* If the popped vertex is visited before, just continue without using it.
* Apply the same algorithm again until the priority queue is empty.
* Time Complexity of Dijkstra's Algorithm is O(V2) but with min-priority queue it drops down to O(V+ElogV).
* However, if we have to find the shortest path between all pairs of vertices, both of the above methods would be expensive in terms of time. Discussed below is another alogorithm designed for this case.

**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:

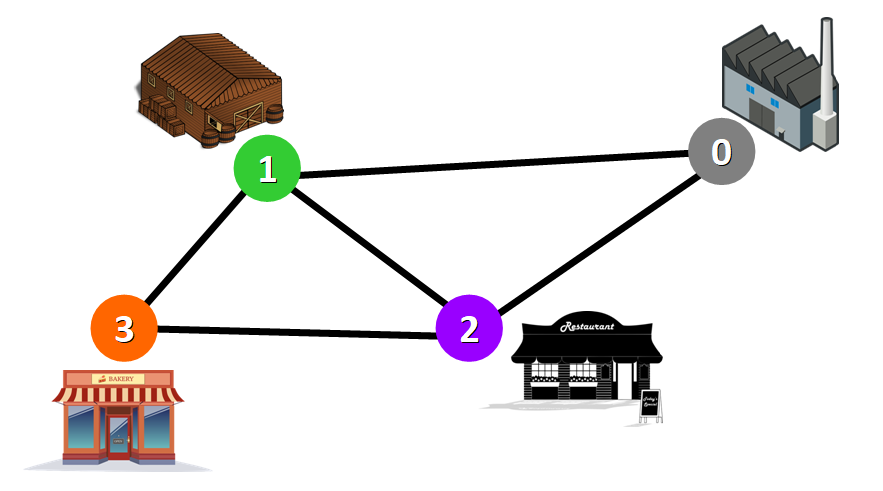


**Nodes**are represented with colored circles and **edges**are represented with lines that connect these circles.

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

**Applications**

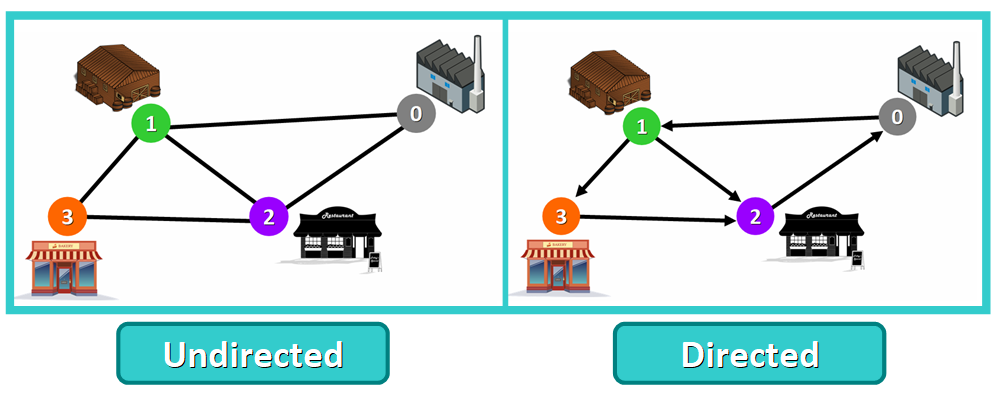
Graphs are directly applicable to real-world scenarios. For example, we could use graphs to model a transportation network where nodes would represent facilities that send or receive products and edges would represent roads or paths that connect them (see below).

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.

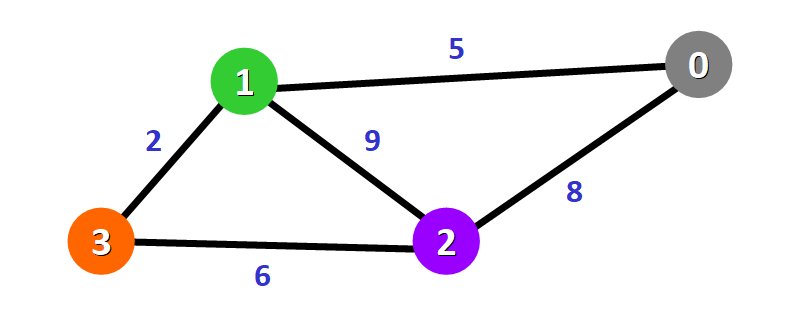


**💡 Tip:** in this article, we will work with **undirected**graphs.

**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.

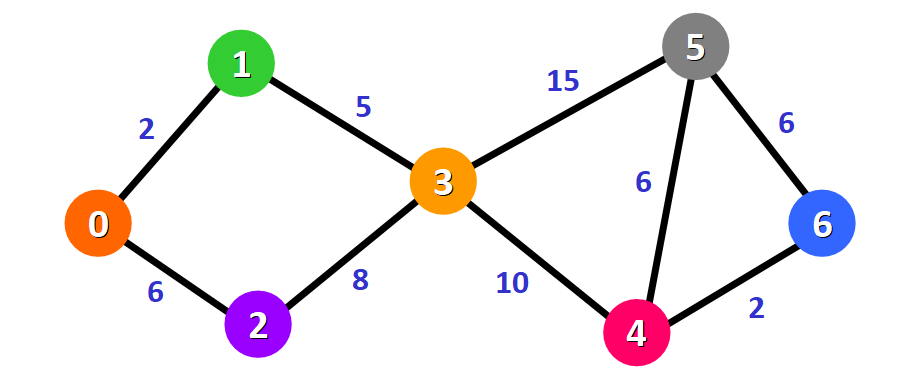
**Basics of Dijkstra's Algorithm**

* 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.

**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:



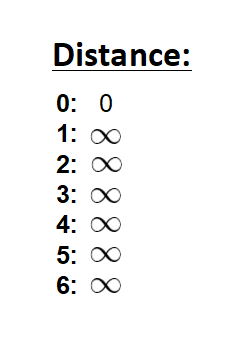
The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

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

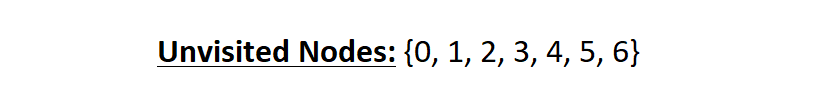
We will have the shortest path from node 0 to node 1, from node 0 to node 2, from node 0 to node 3, and so on for every node in the graph.

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.

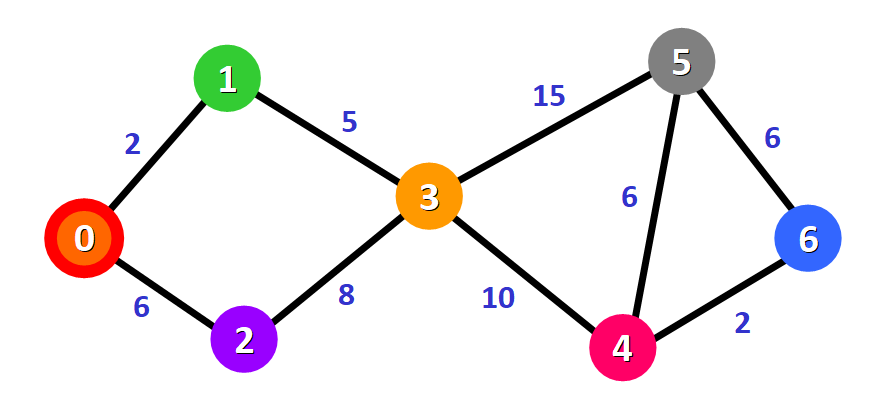


We also have this list (see below) to keep track of the nodes that have not been visited yet (nodes that have not been included in the path):

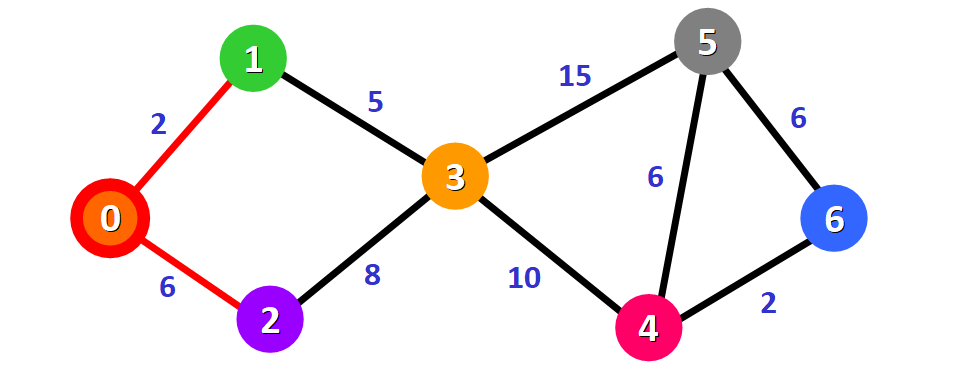


**💡 Tip:**Remember that the algorithm is completed once all nodes have been added to the path.

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:

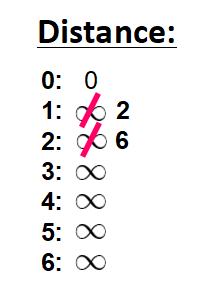
https://www.freecodecamp.org/news/content/images/2020/06/image-87.png

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:

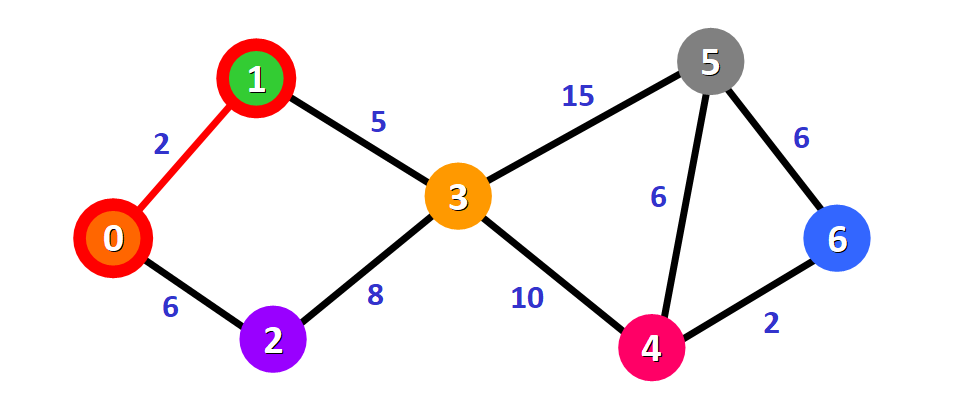


After updating the distances of the adjacent nodes, we need to:

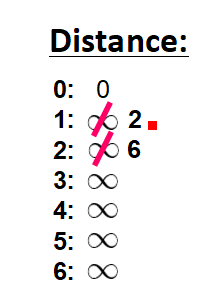
* Select the node that is closest to the source node based on the current known distances.
* Mark it as visited.
* Add it to the path.

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.

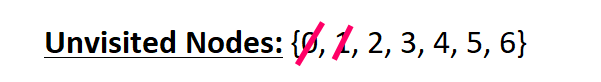
In the diagram, we can represent this with a red edge:



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:

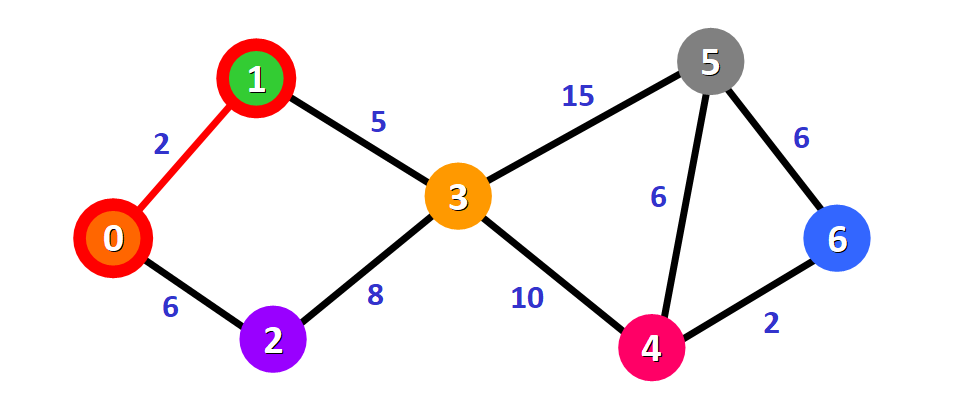


We cross it off from the list of unvisited nodes:

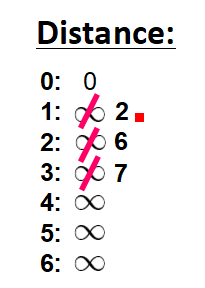


Now we need to analyze the new adjacent nodes to find the shortest path to reach them. We will only analyze the nodes that are adjacent to the nodes that are already part of the shortest path (the path marked with red edges).

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



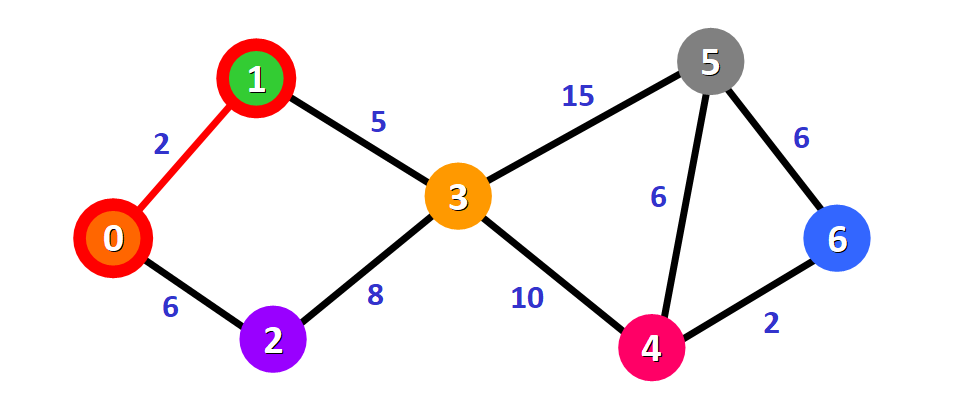
Since we already have the distance from the source node to node 2 written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node 3):



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

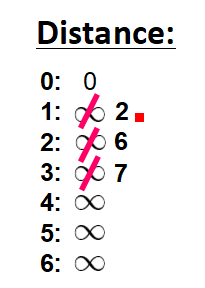
To find the distance from the source node to another node (in this case, node 3), we add the weights of all the edges that form the shortest path to reach that node:

* **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).

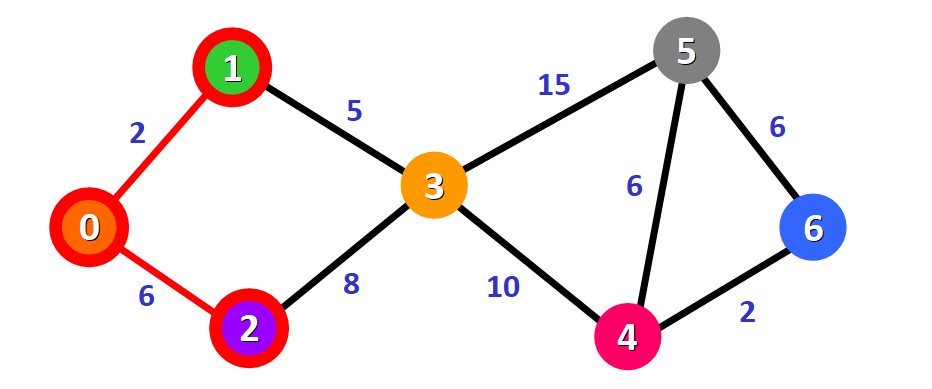


Now that we have the distance to the adjacent nodes, we have to choose which node will be added to the path. We must select the **unvisited**node with the shortest (currently known) distance to the source node.

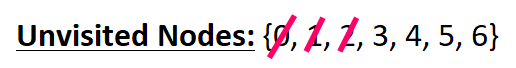
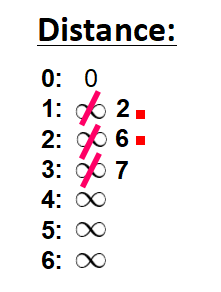
From the list of distances, we can immediately detect that this is node 2 with distance **6**:



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

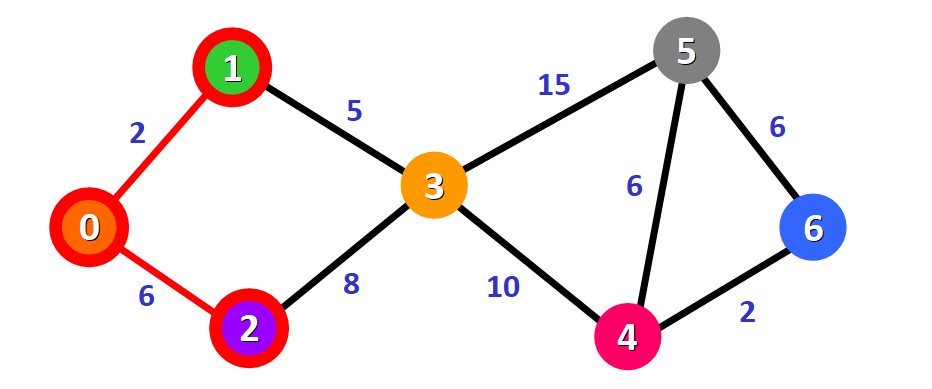


We also mark it as visited by adding a small red square in the list of distances and crossing it off from the list of unvisited nodes:



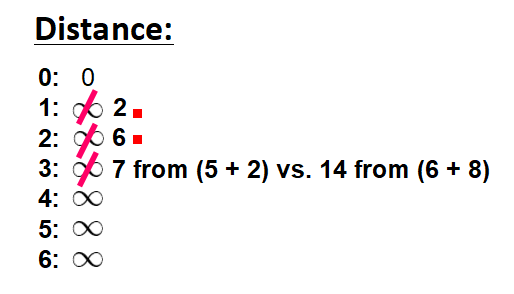
Now we need to repeat the process to find the shortest path from the source node to the new adjacent node, which is node 3.

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.



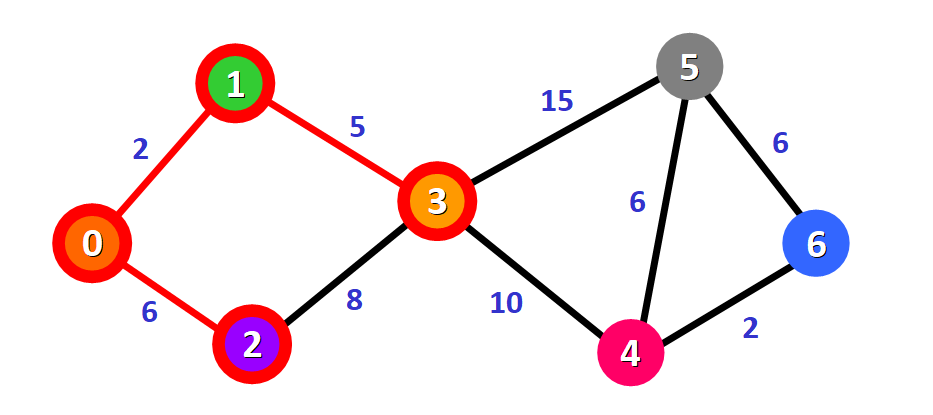
Node 3 already has a distance in the list that was recorded previously (**7,**see the list below). This distance was the result of a previous step, where we added the weights 5 and 2 of the two edges that we needed to cross to follow the path 0 -> 1 -> 3.

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**.

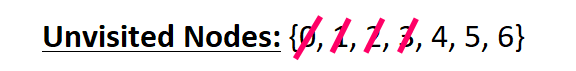
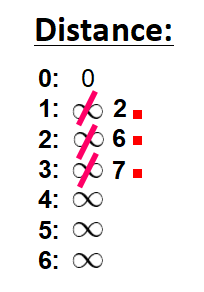


Clearly, the first (existing) distance is shorter (7 vs. 14), so we will choose to keep the original path 0 -> 1 -> 3. **We only update the distance if the new path is shorter.**

Therefore, we add this node to the path using the first alternative: 0 -> 1 -> 3.

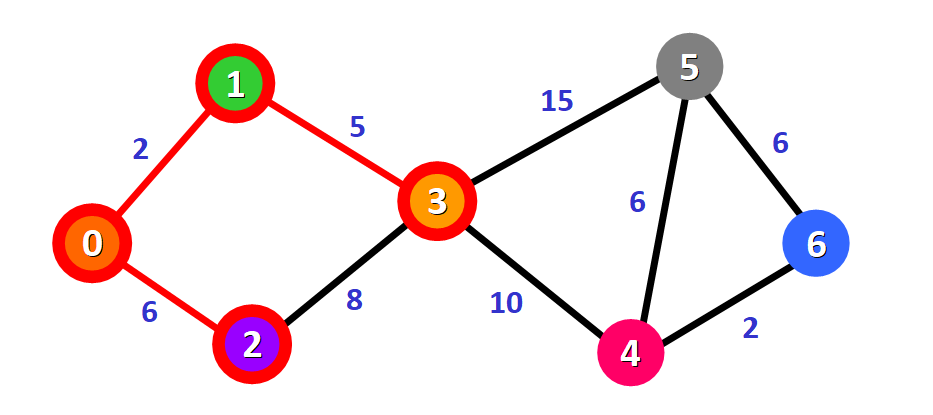


We mark this node as visited and cross it off from the list of unvisited nodes:



Now we repeat the process again.

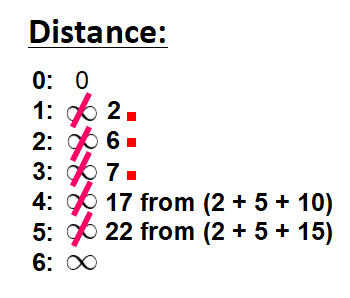
We need to check the new adjacent nodes that we have not visited so far. This time, these nodes are node 4 and node 5 since they are adjacent to node 3.



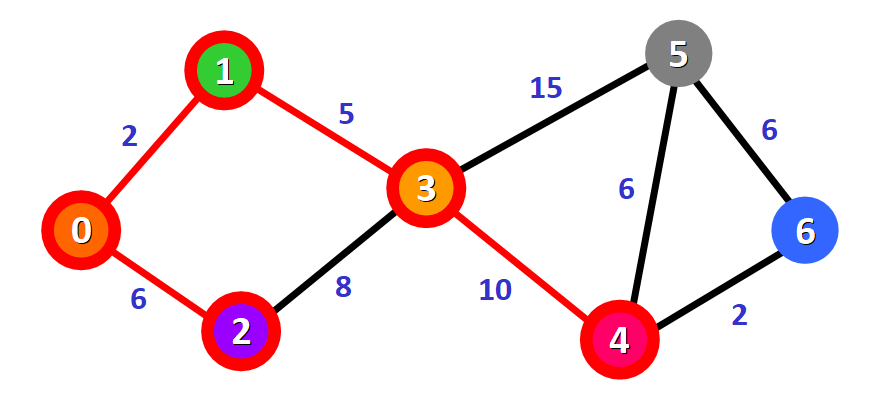
We update the distances of these nodes to the source node, always trying to find a shorter path, if possible:

* **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.

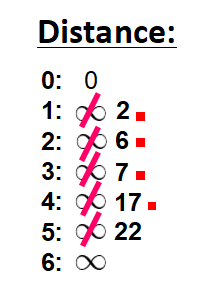
**💡 Tip:** Notice that we can only consider extending the shortest path (marked in red). We cannot consider paths that will take us through edges that have not been added to the shortest path (for example, we cannot form a path that goes through the edge 2 -> 3).



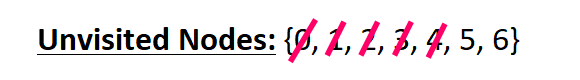
We need to choose which unvisited node will be marked as visited now. In this case, it's node 4 because it has the shortest distance in the list of distances. We add it graphically in the diagram:



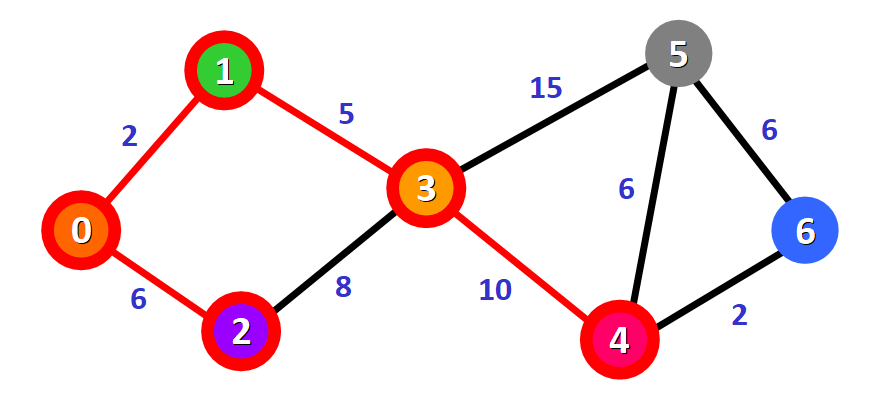
We also mark it as "visited" by adding a small red square in the list:



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



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.



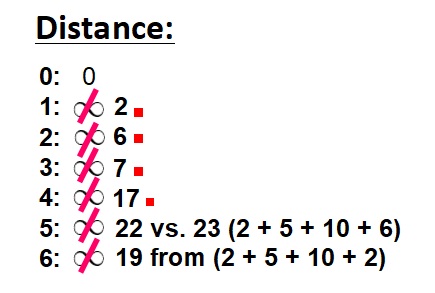
**For node 5:**

* 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).

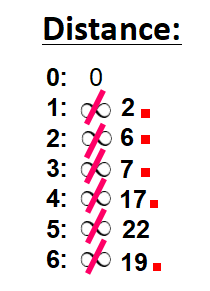
Clearly, the first path is shorter, so we choose it for node 5.

**For node 6:**

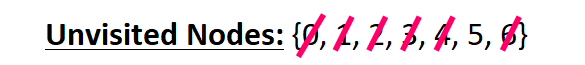
* The path available is 0 -> 1 -> 3 -> 4 -> 6, which has a distance of **19** from the source node (2 + 5 + 10 + 2).



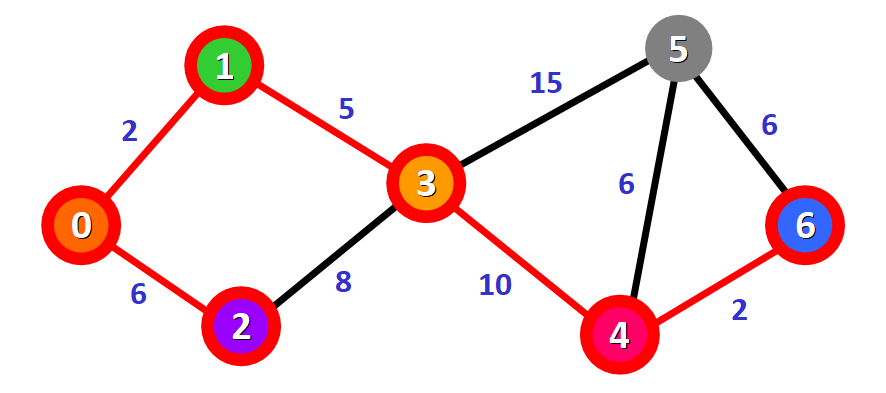
We mark the node with the shortest (currently known) distance as visited. In this case, node 6.



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



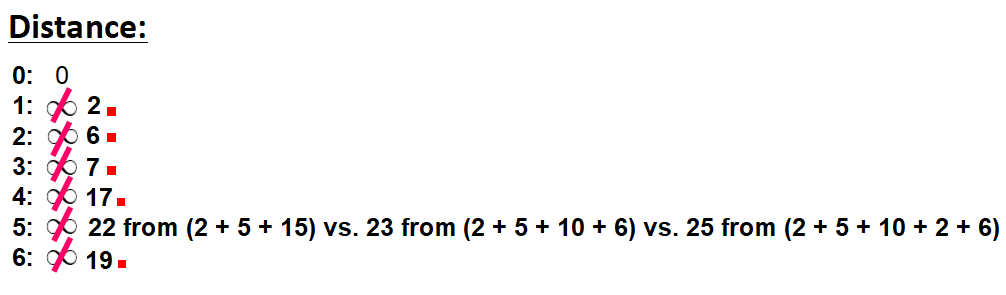
Now we have this path (marked in red):



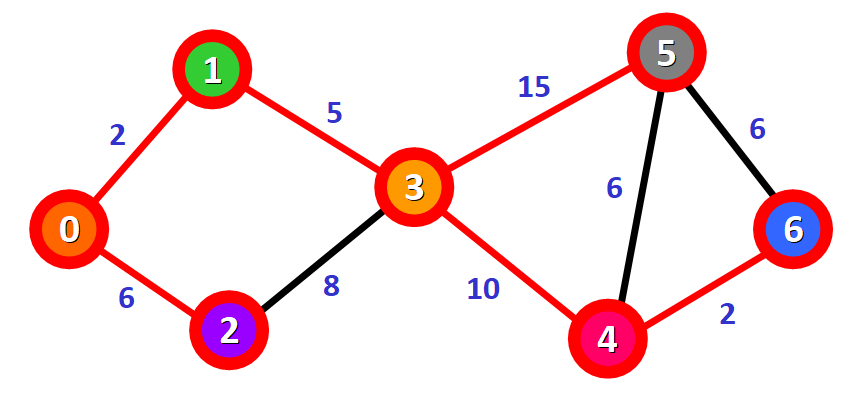
Only one node has not been visited yet, node 5. Let's see how we can include it in the path.

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

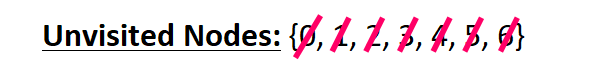
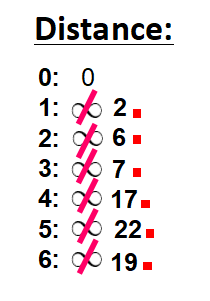
* **Option 1:**0 -> 1 -> 3 -> 5 with a distance of **22**(2 + 5 + 15).
* **Option 2:**0 -> 1 -> 3 -> 4 -> 5 with a distance of **23** (2 + 5 + 10 + 6).
* **Option 3:** 0 -> 1 -> 3 -> 4 -> 6 -> 5 with a distance of **25**(2 + 5 + 10 + 2 + 6).



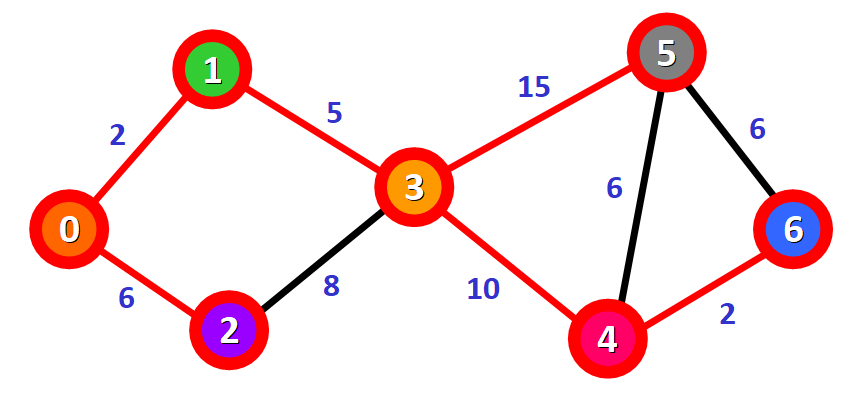
We select the shortest path: 0 -> 1 -> 3 -> 5 with a distance of **22**.



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



**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.

***HASHING TOPIC TO BE DONE FROM HAND WRITTEN NOTES MADE BY CHIRAG***