Data Structure
Algorithms & Applications

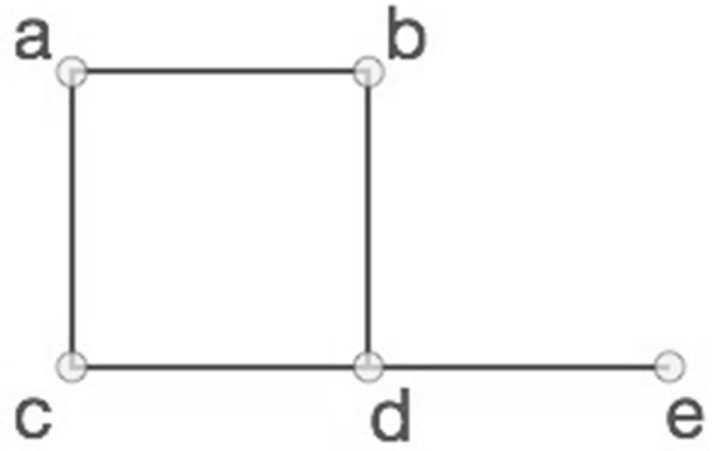
CT-159

Prepared by
Muhammad Kamran

Graph

- Graph Data Structure is a **non-linear data** structure consisting of vertices and edges.
- 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.

Graph



- In the above graph :

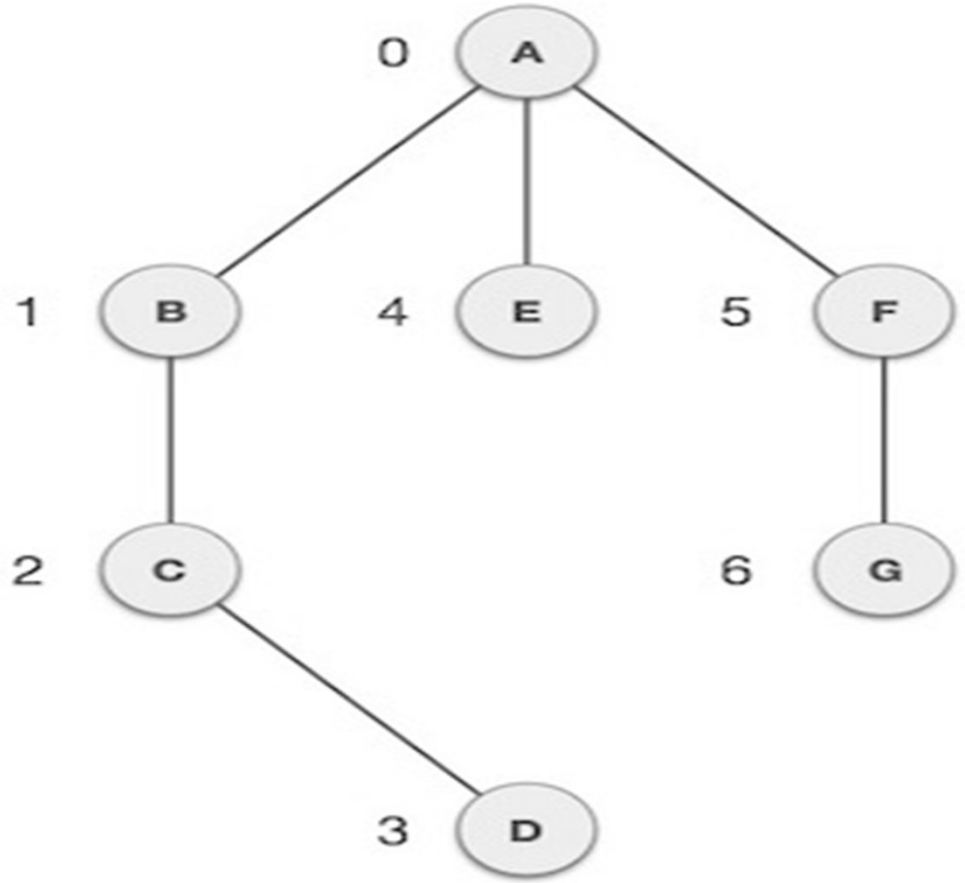
  V = {a, b, c, d, e}

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

Graph

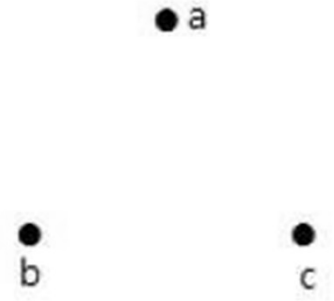
- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labelled circle represents vertices. Thus, A to G are vertices. We can represent them using an array. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents 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. 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.

Graph



Null Graph

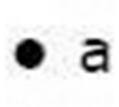
- **A graph having no edges** is called a Null Graph.



- In the above graph, there are three vertices named ‘a’, ‘b’, and ‘c’, but there are no edges among them. Hence it is a Null Graph.

Trivial Graph

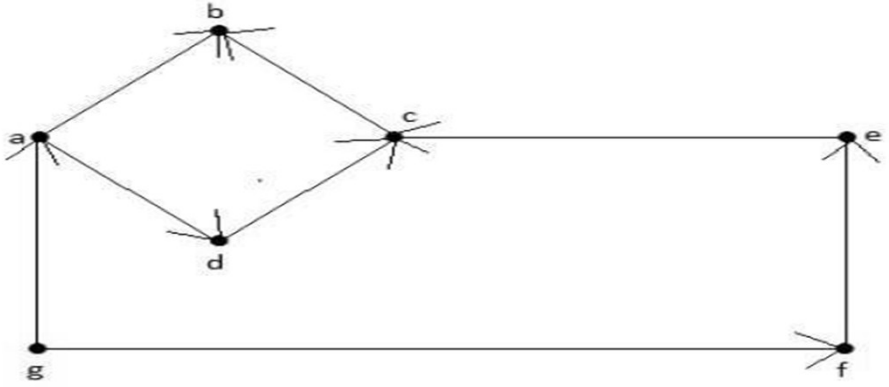
- A **graph with only one vertex** is called a Trivial Graph.

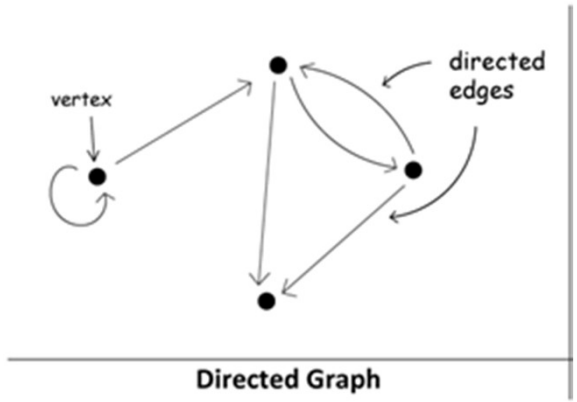


- In the above shown graph, there is only one vertex ‘a’ with no other edges. Hence it is a Trivial graph.

Directed Graph

- In a directed graph, each edge has a direction.

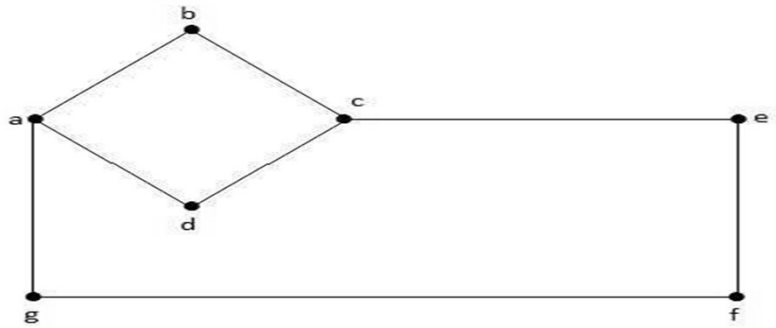


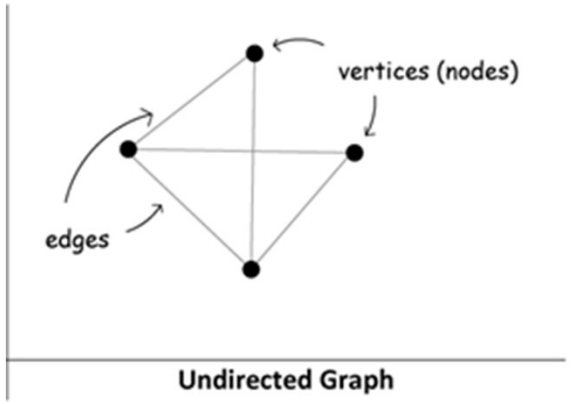


- In the above graph, we have seven vertices ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, and ‘g’, and eight edges ‘ab’, ‘cb’, ‘dc’, ‘ad’, ‘ec’, ‘fe’, ‘gf’, and ‘ga’. As it is a directed graph, each edge bears an arrow mark that shows its direction. Note that in a directed graph, ‘ab’ is different from ‘ba’.

Non-Directed Graph

- A non-directed graph contains edges but the edges are not directed ones.

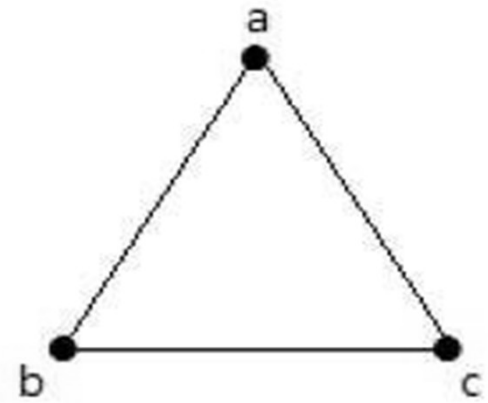




- In this graph, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’ are the vertices, and ‘ab’, ‘bc’, ‘cd’, ‘da’, ‘ag’, ‘gf’, ‘ef’ are the edges of the graph. Since it is a non-directed graph, the edges ‘ab’ and ‘ba’ are same. Similarly other edges also considered in the same way.

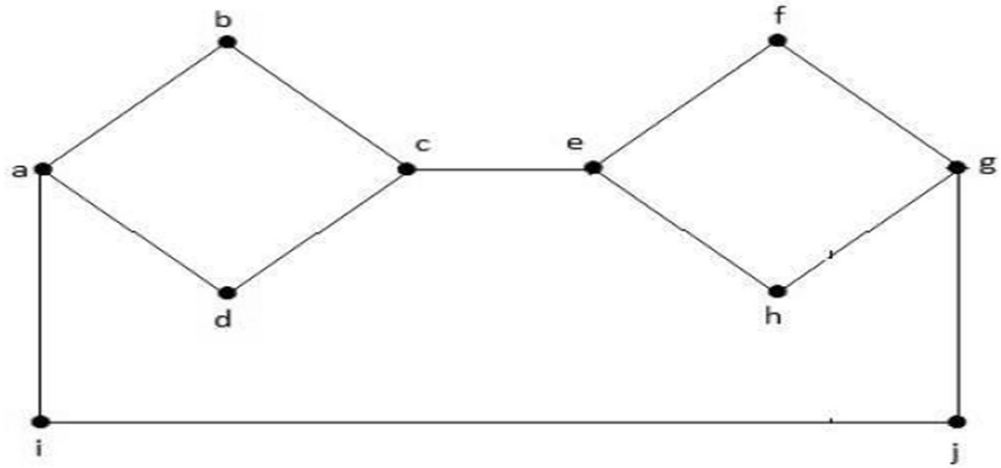
Simple Graph

- A graph **with no loops** and no **parallel edges** is called a simple graph.
- The maximum number of edges possible in a single graph with ‘n’ vertices is ${}^{n}C_{2}$ where ${}^{n}C_{2} = n(n – 1)/2$.



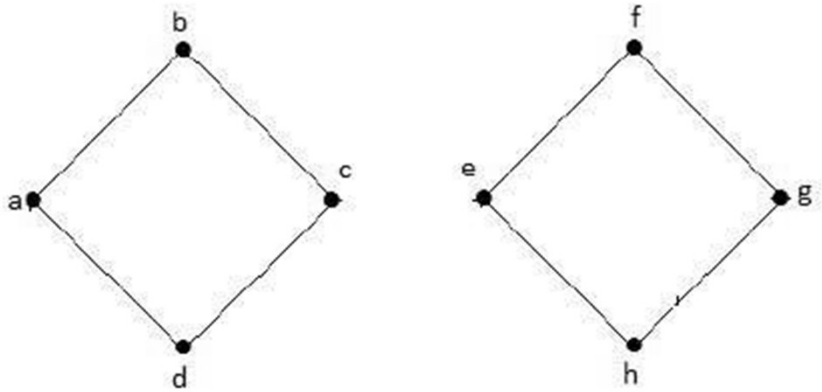
Connected Graph

- A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph. So that we can say that it is connected to some other vertex at the other side of the edge.

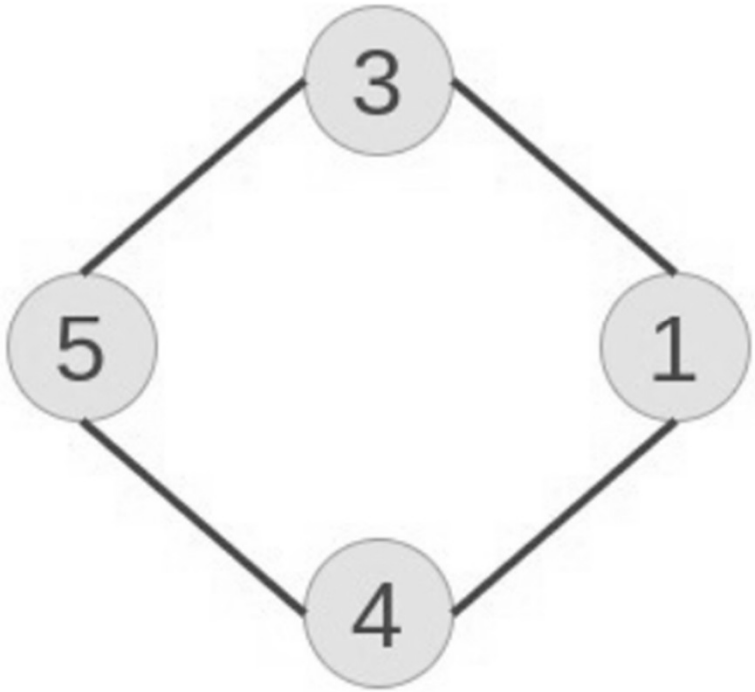


Disconnected Graph

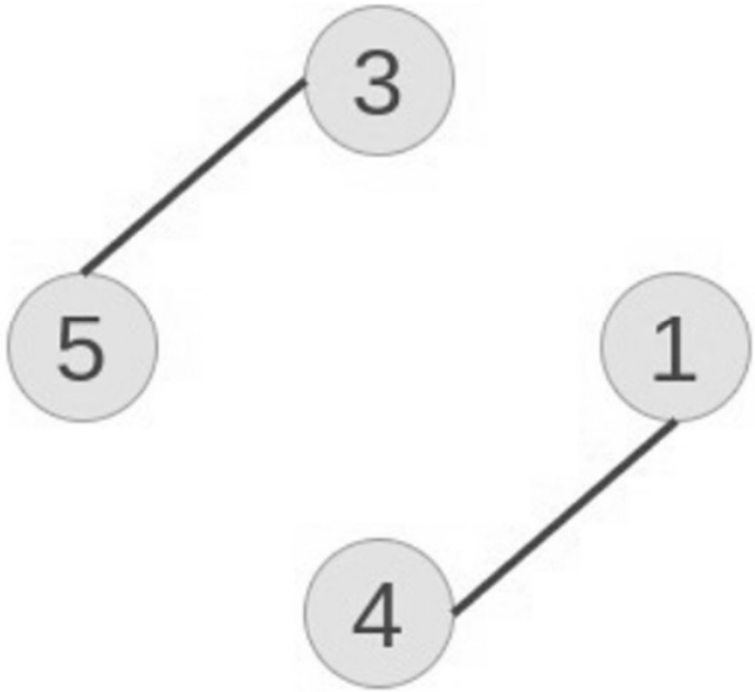
- A graph G is disconnected, if it does not contain at least two connected vertices.



Connected vs Disconnected



Connected Graph

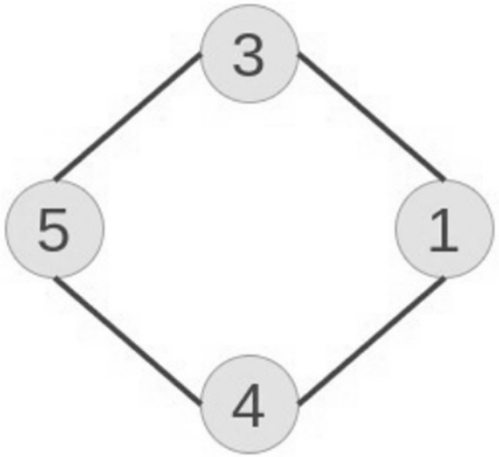


Disconnected Graph

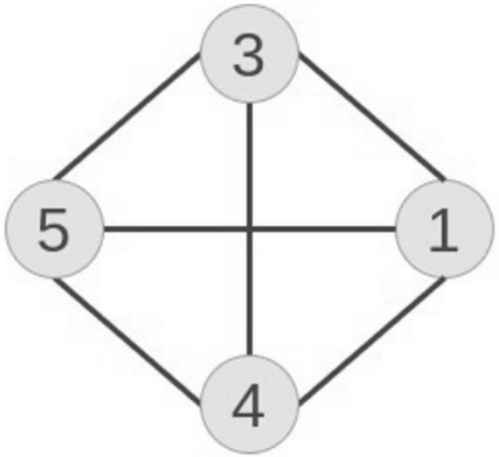
Regular Vs Complete Graph

The graph in which the degree of every vertex is equal to K is called K regular graph.

The graph in which from each node there is an edge to each other node.



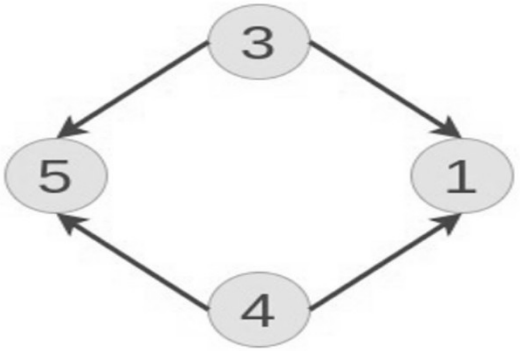
2-Regular



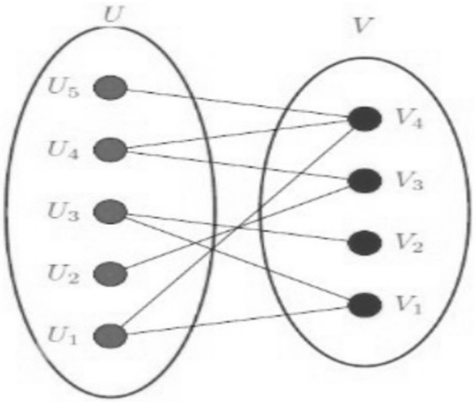
Complete Graph

Directed Acyclic Vs Bipartite Graph

- A Directed Graph that does not contain any cycle.
- A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.

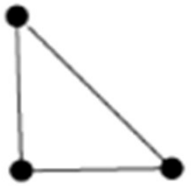


Directed Acyclic Graph



Bipartite Graph

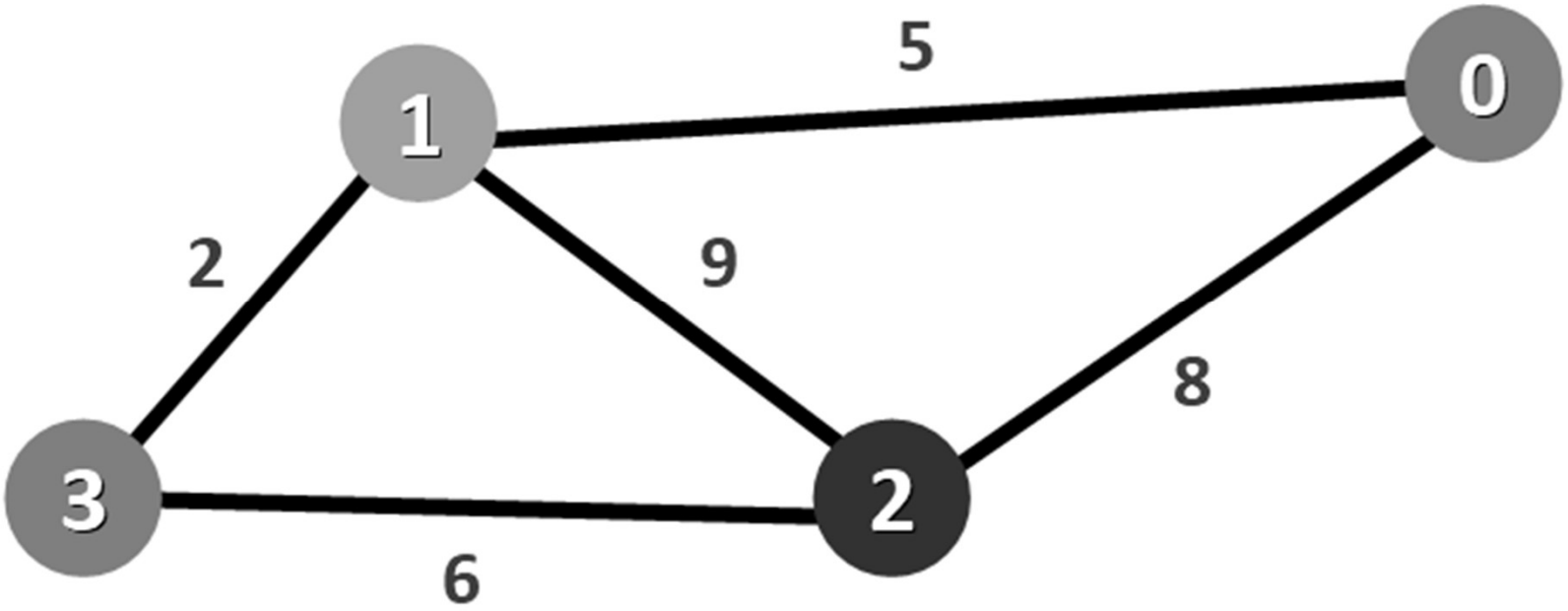
Other types of Graph

Graph Type	Multiple Edges	Loops	Directed Edges	Undirected Edges
Simple graph	No	No		
Multigraph	Yes	Yes & No		
Pseudograph	Yes	Yes	No	
Mixed graph	yes	yes		

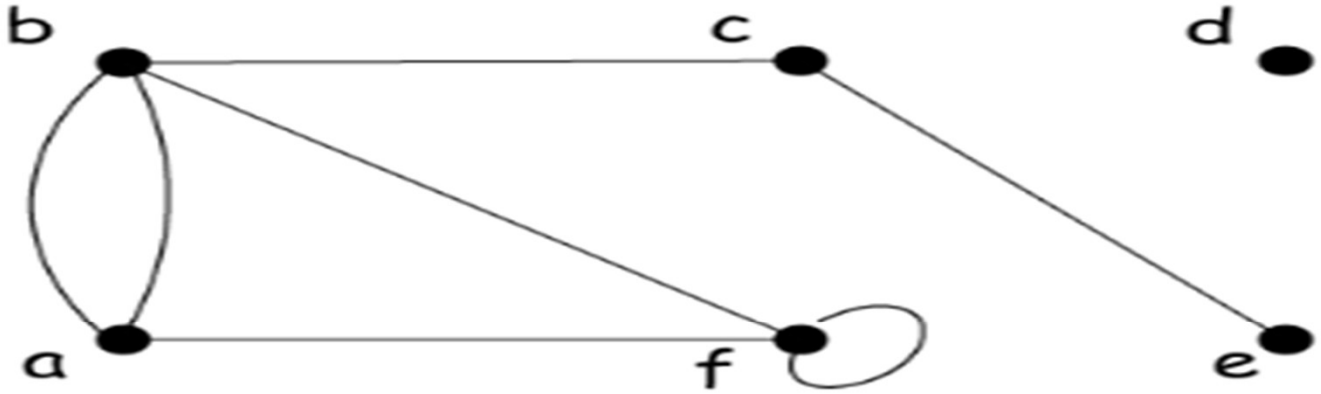
Weighted Graphs

- 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.

Weighted Graphs



let's identify the degree and neighborhood for each vertex

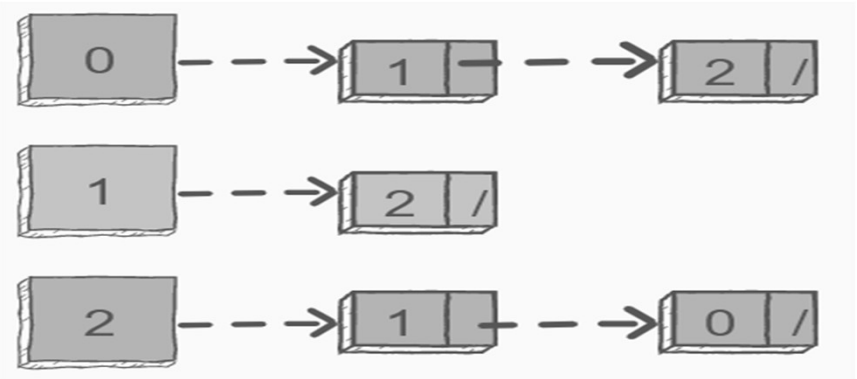


Degree	Neighborhood
$\deg(a)=3$	$N(a)=\{b,f\}$
$\deg(b)=4$	$N(b)=\{a,c,f\}$
$\deg(c)=2$	$N(c)=\{b,e\}$
$\deg(d)=0$ *(isolated)*	$N(d)=\varnothing$
$\deg(e)=1$ *(pendant)*	$N(d)=\{c\}$
$\deg(f)=4$	$N(f)=\{a,b,f\}$

Graph Representation

Adjacency List :

- To create an Adjacency list, an array of lists is used. The size of the array is equal to the number of nodes.
- A single index, array[i] represents the list of nodes adjacent to the *i*th node.

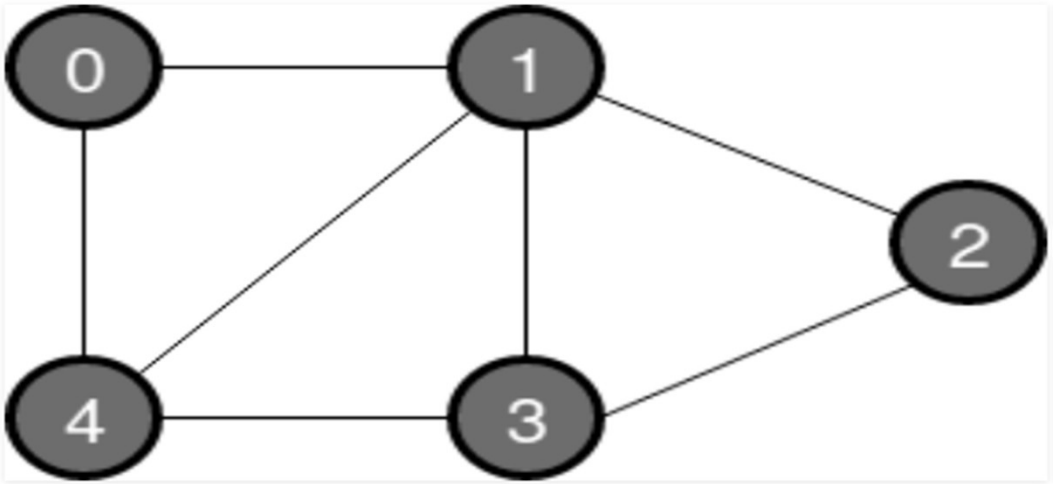


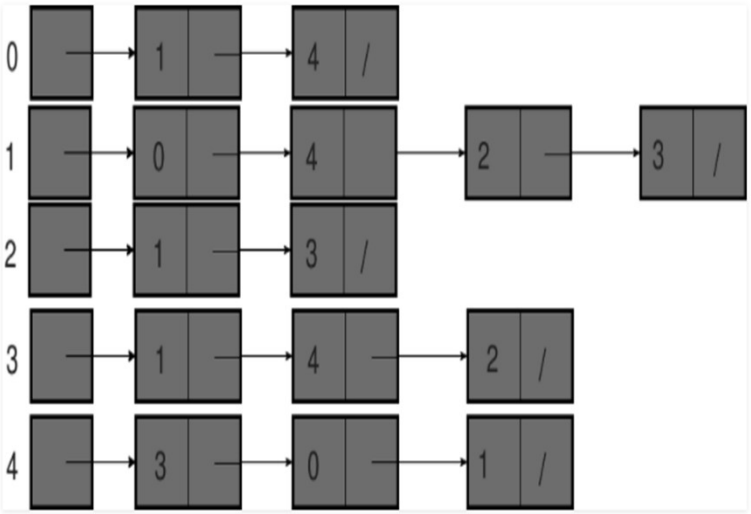
Graph Representation

Adjacency Matrix :

- An Adjacency Matrix is a 2D array of size V x V where V is the number of nodes in a graph. A slot matrix[i][j] = 1 indicates that there is an edge from node *i* to node *j*.

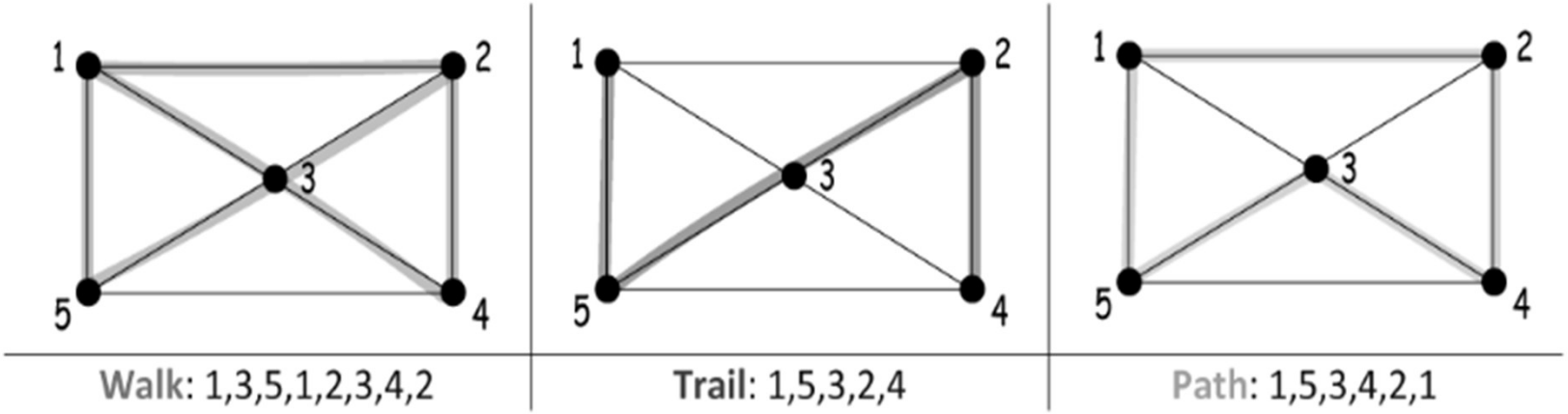
Graph Representations





	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Walk Vs Trails Vs Paths



Walk is a finite path that joins a sequence of vertices where vertices and edges can be repeated. In other words, every time you "traverse" a graph, you get a walk.

Trail is a walk in which all the edges are distinct, but a vertex can be repeated.

Path is a trail that joins a sequence of vertices and distinct edges where no vertex nor edge is repeated, and vertices are listed in order.

Graph Traversals

- Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.
- The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.
- During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS)

- There are many ways to traverse graphs. BFS is the most commonly used approach.
- BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer-wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

Breadth First Search (BFS)

- **Initialization:** Enqueue the given source vertex into a queue and mark it as visited.

- **Exploration:** While the queue is not empty:

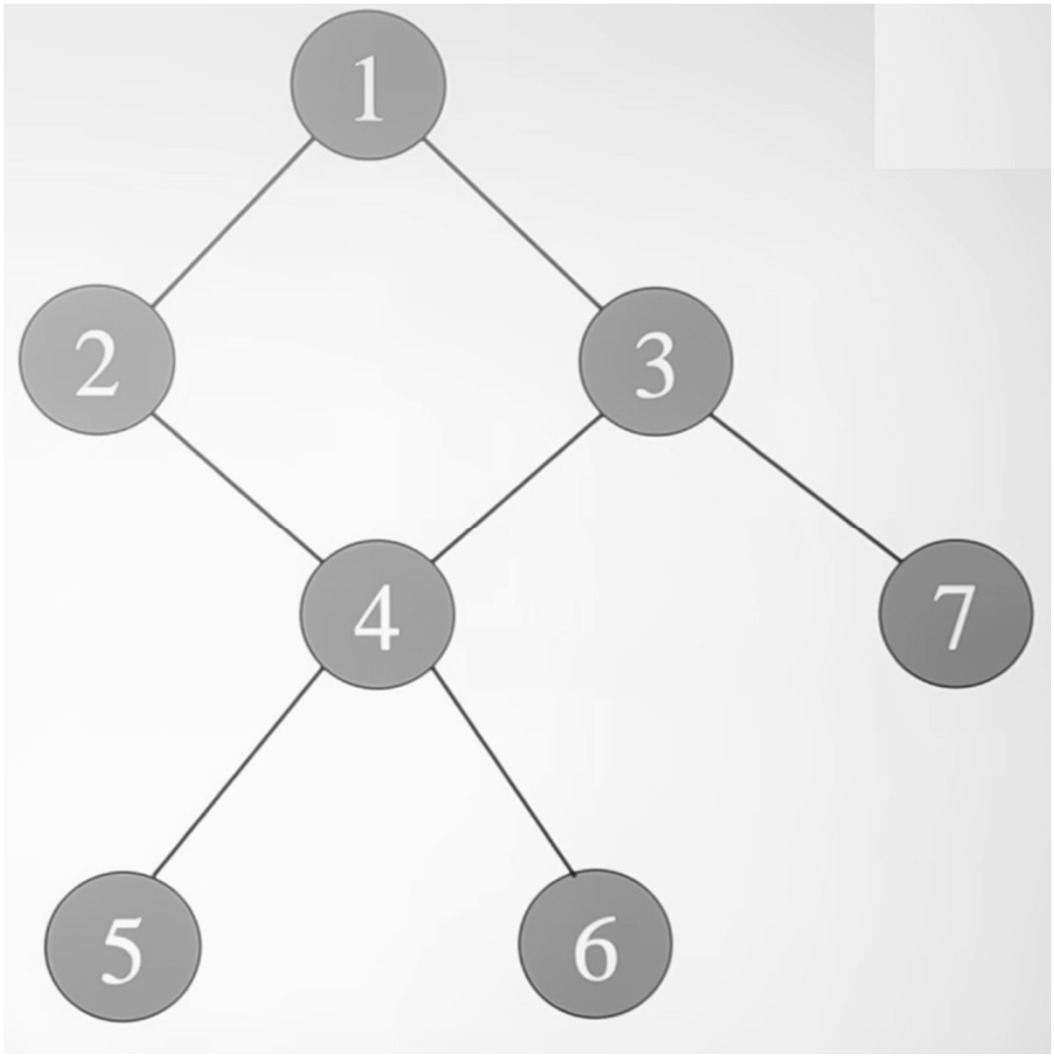
  - Dequeue a node from the queue and visit it (e.g., print its value).

  - For each unvisited neighbor of the dequeued node:

    1. Enqueue the neighbor into the queue.
    2. Mark the neighbor as visited.

- **Termination:** Repeat step 2 until the queue is empty.

Breadth First Search (BFS)



Depth First Search (DFS)

- The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.
- Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse.
- All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

Depth First Search (DFS)

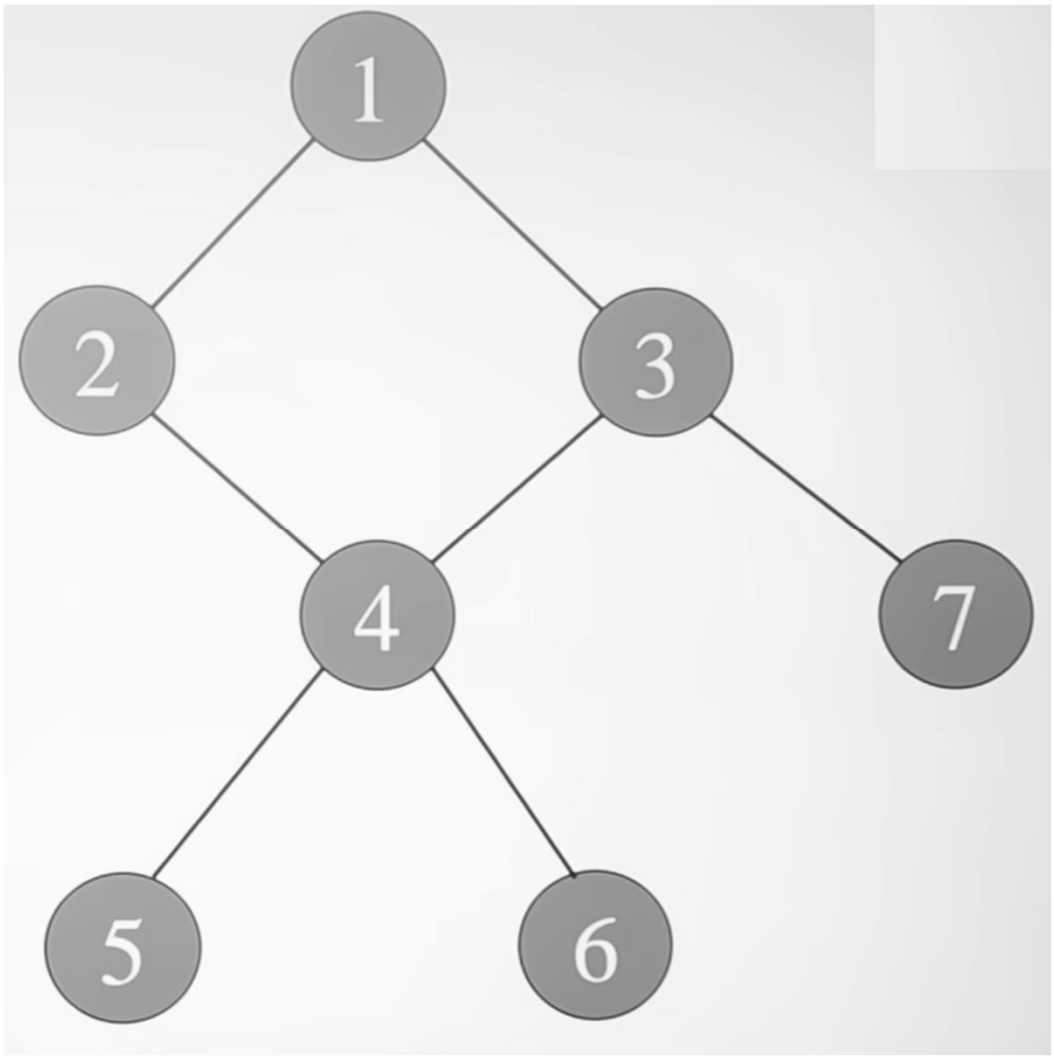
- This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

1 - Pick a starting node and push all its adjacent nodes into a stack.

2 - Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

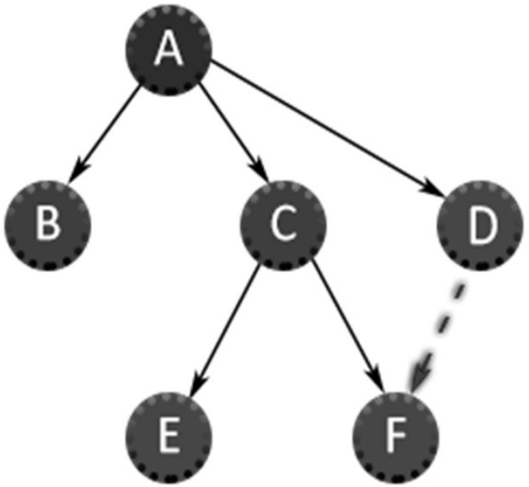
3 - Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once.

Depth First Search (DFS)



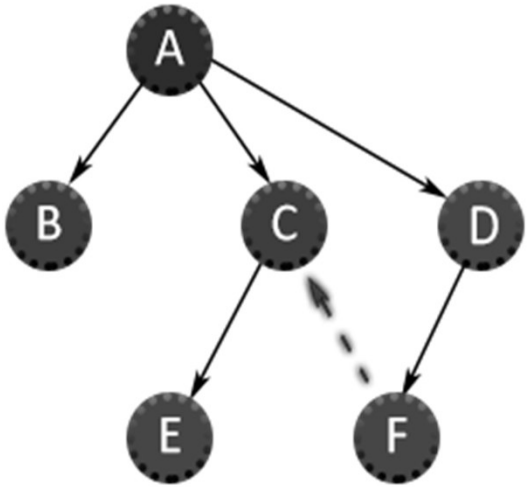
Graph Traversal

BFS



A B C D E F

DFS



A D F C E B

What is Dijkstra's algorithm?

Dijkstra's Algorithm allows you to calculate the **shortest** path between one node of your choice and every other node in a graph.

Algorithm

1 - Mark all nodes as *unvisited*.

2 - Mark the selected **initial** node with a *current* distance of **0** and the rest with **infinity**.

3 - Set the **initial** *node* as **current** *node*.

4 - For the **current** *node*, consider all of its **unvisited** neighbors and calculate their distances by adding the *current* distance of **current** *node* to the *weight* of the *edge* connecting **neighbor** *node* and **current** node.

5 - Compare the newly calculated distance to the current distance assigned to the **neighboring** *node* and set is as the **new** *current* distance of **neighboring** *node*.

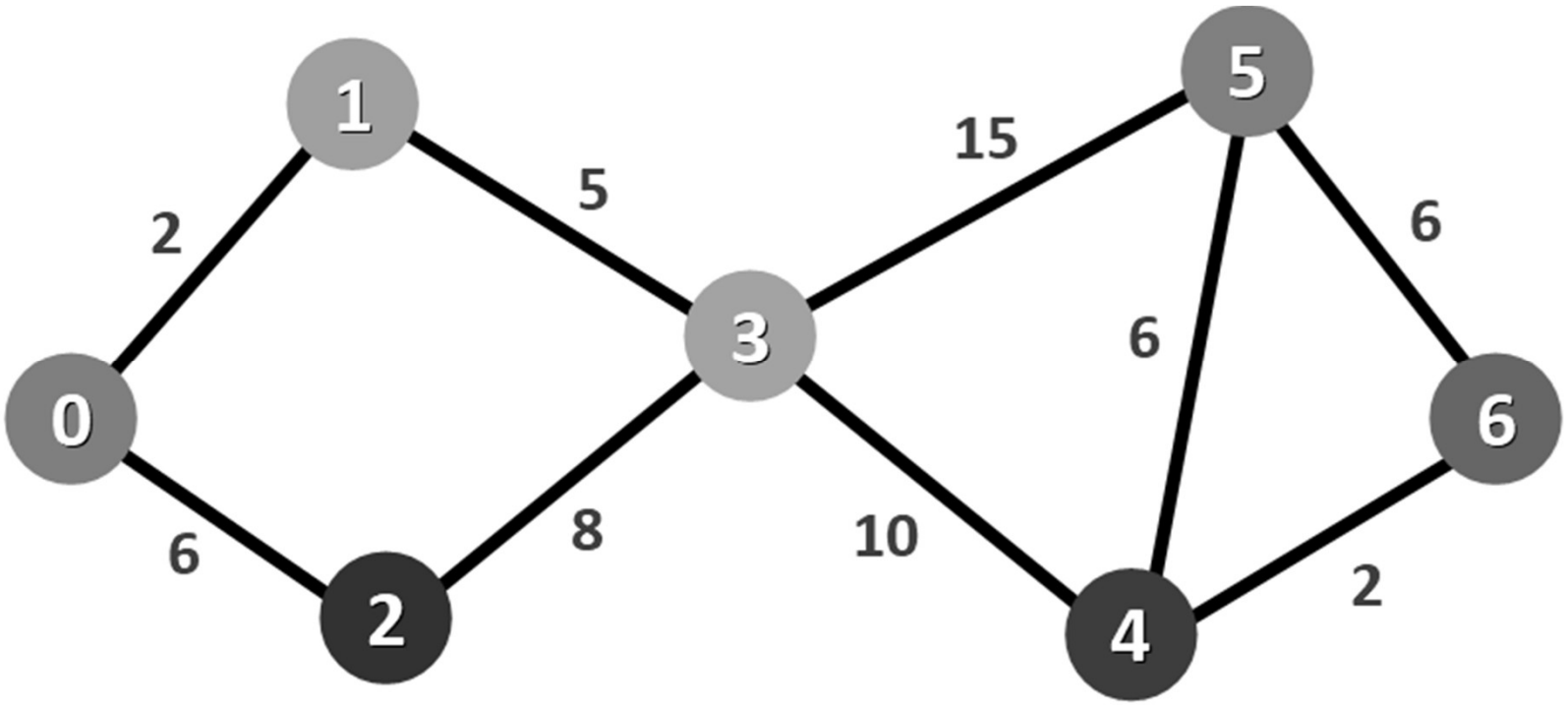
Algorithm

6 - When done considering all of the **unvisited** *neighbors* of the **current** *node*, mark the **current** node as **visited**.

7 - If the **destination** *node* has been marked **visited** then stop. The algorithm has finished.

8 - Otherwise, select the **unvisited** node that is marked with the **smallest** distance, set it as the new **current node**, and go back to **step 4**.

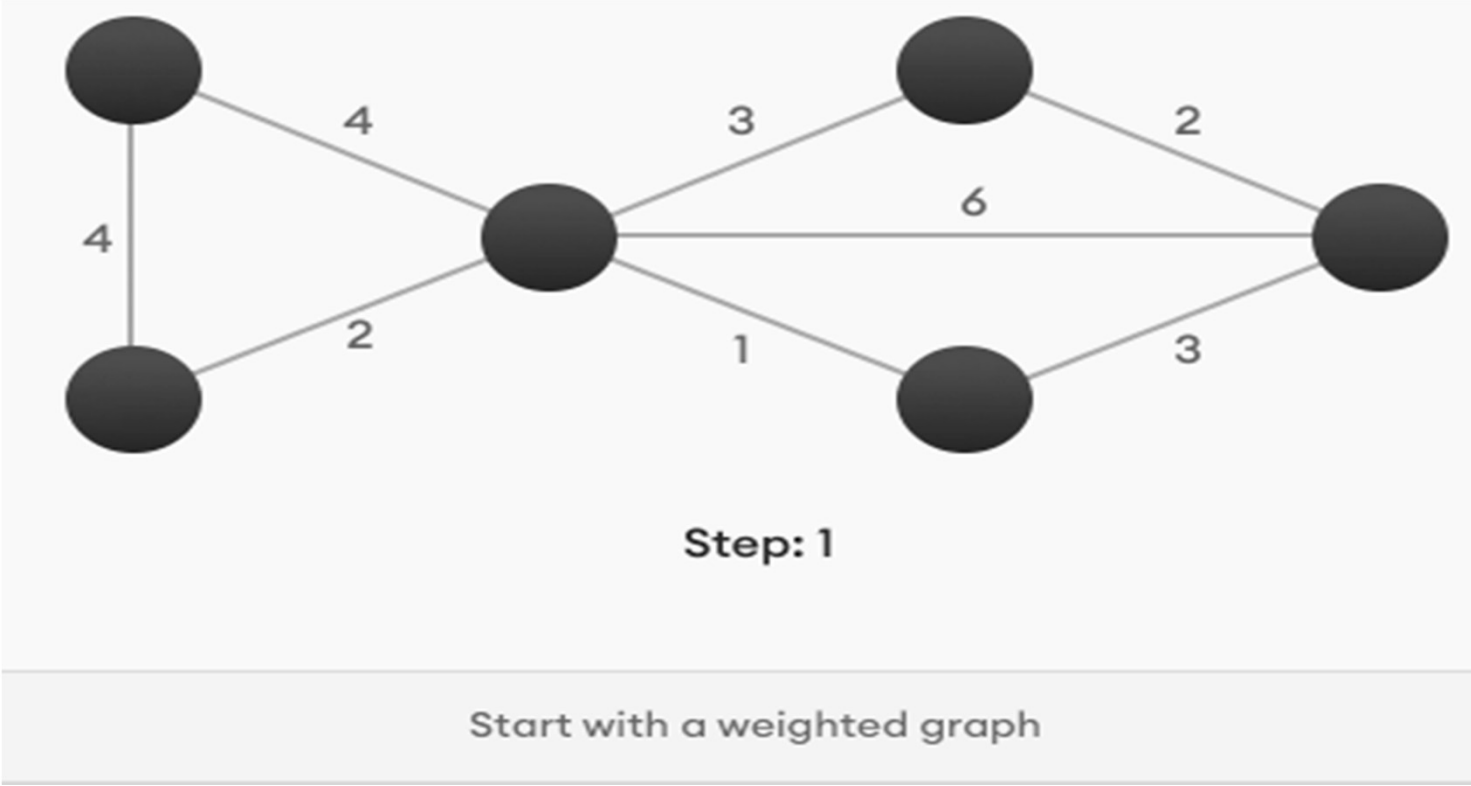
Dijkstra's algorithm



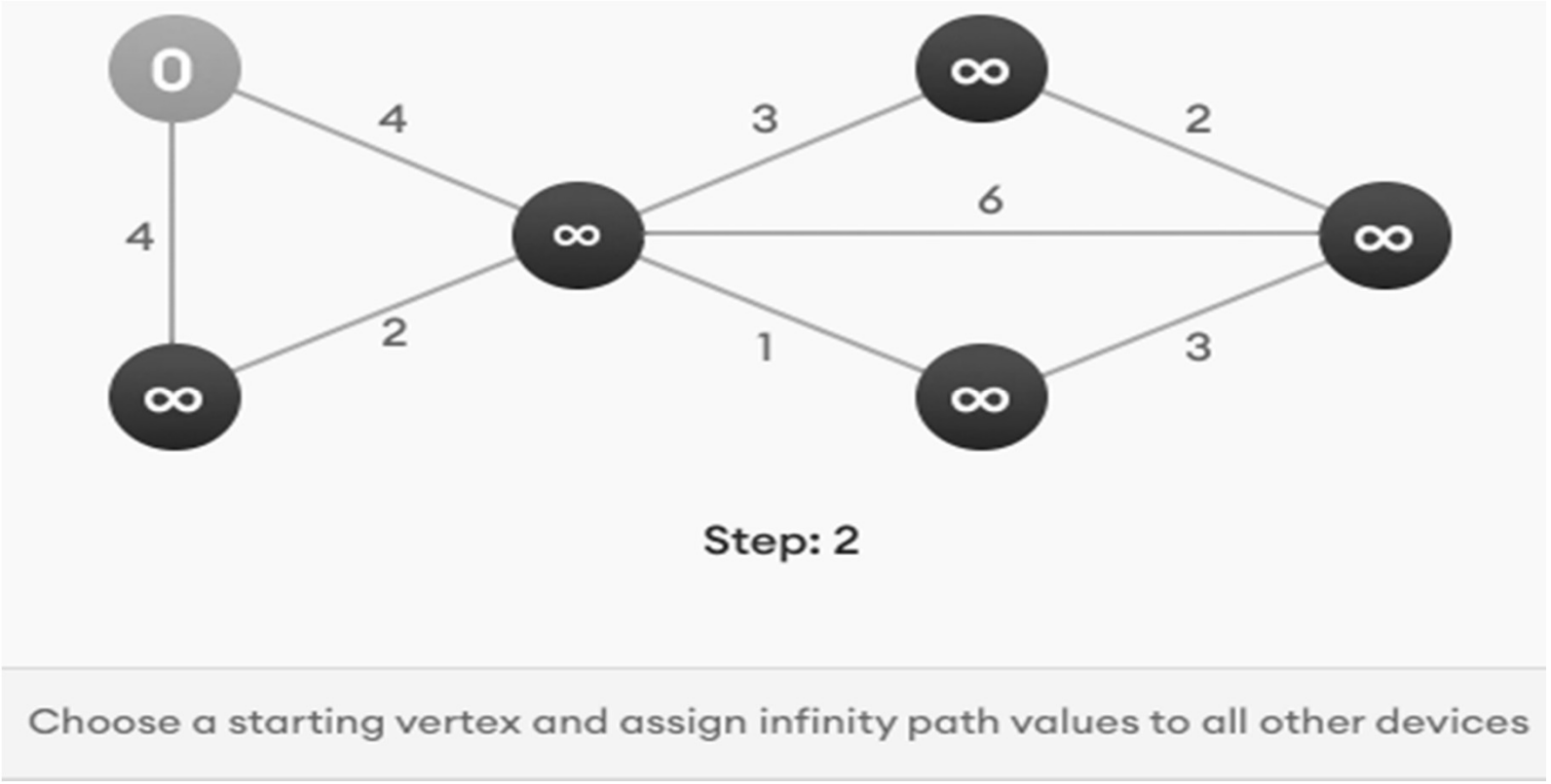
Uses of Dijkstra's Algorithm

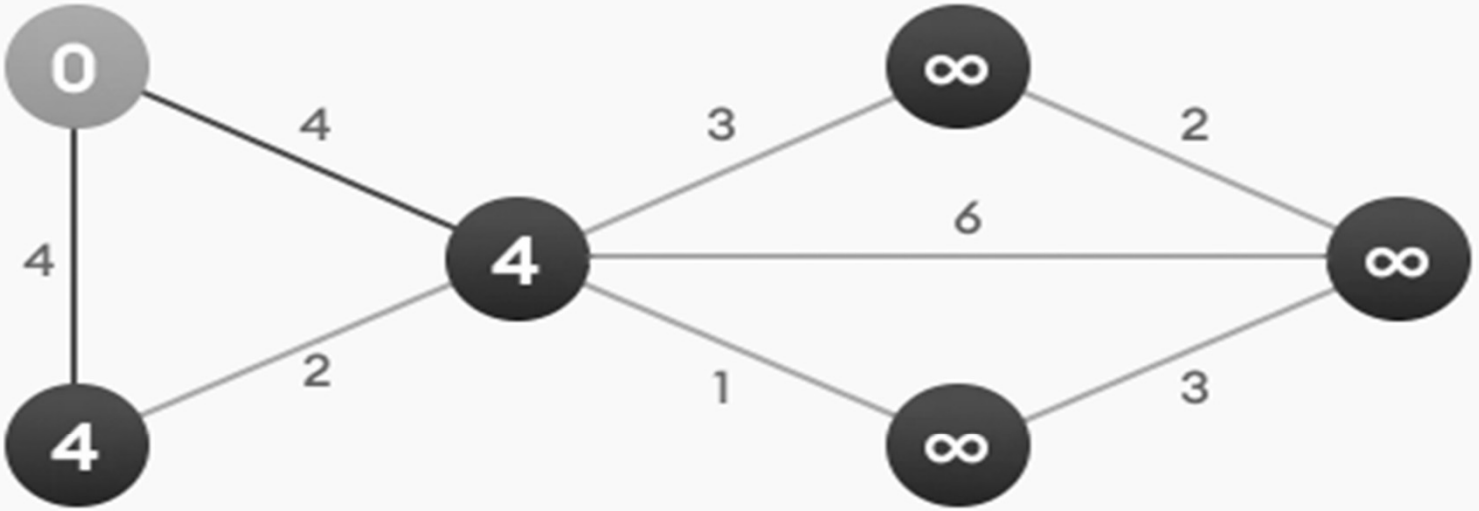
- Dijkstra's Algorithm has several real-world use cases, including the following:
- Map applications.
- Finding locations of a map referring to vertices of a graph.
- IP routing to find Open Shortest Path First.
- Telephone networks.

Another Example



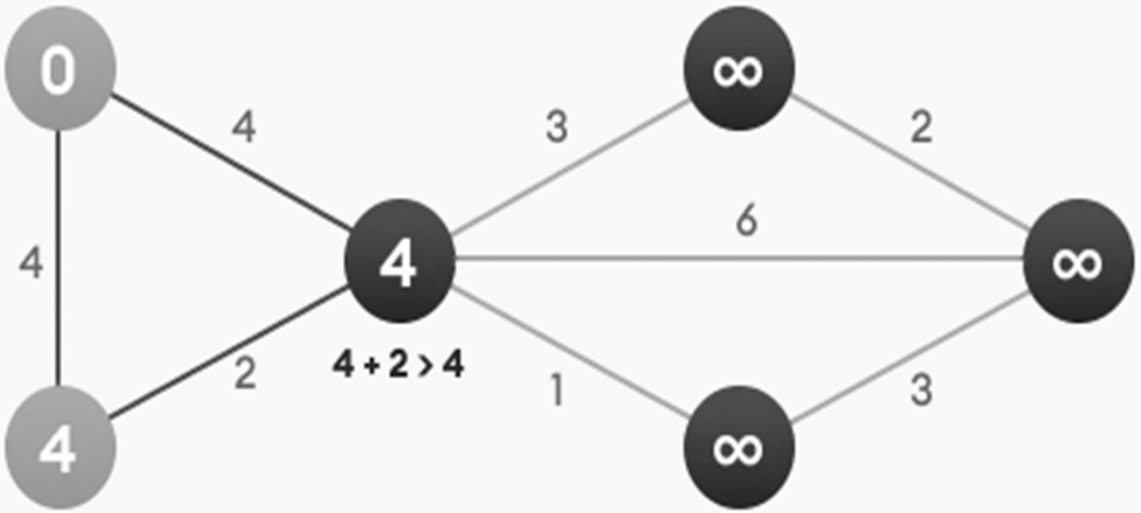
Example Continue . .





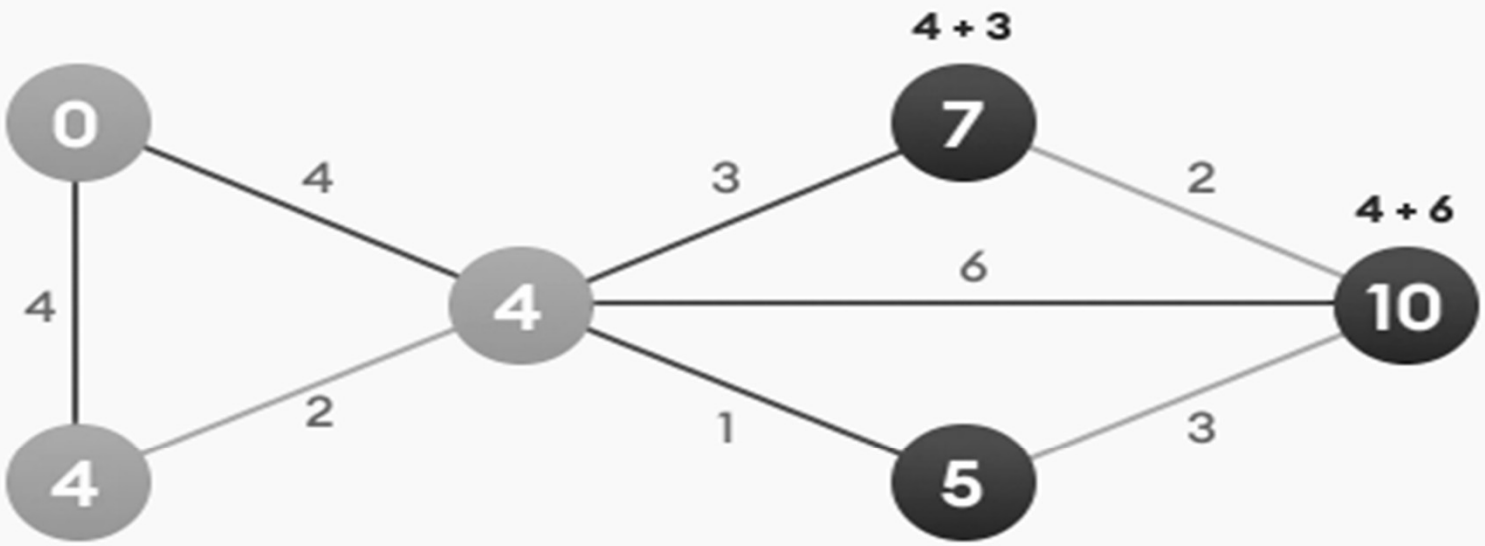
Step: 3

Go to each vertex and update its path length



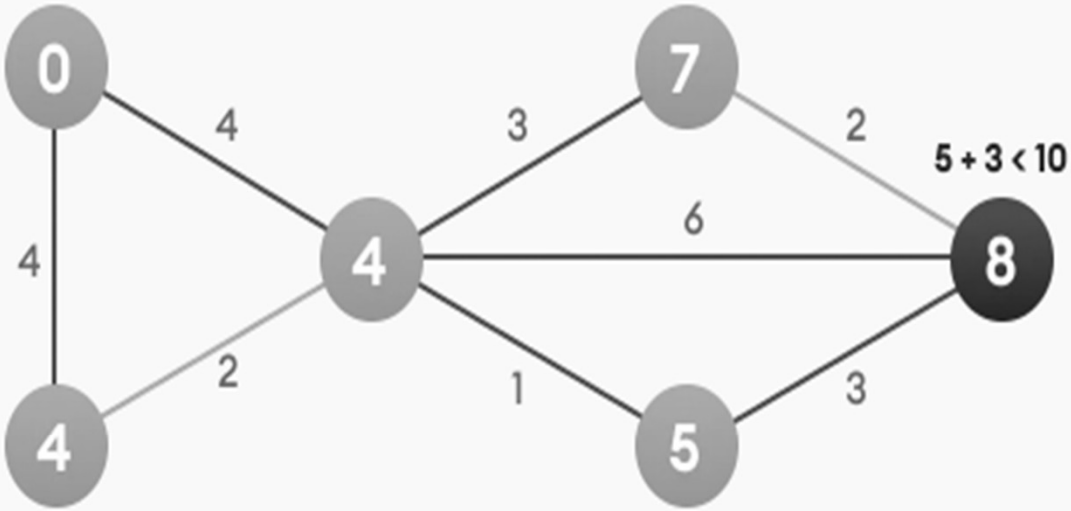
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



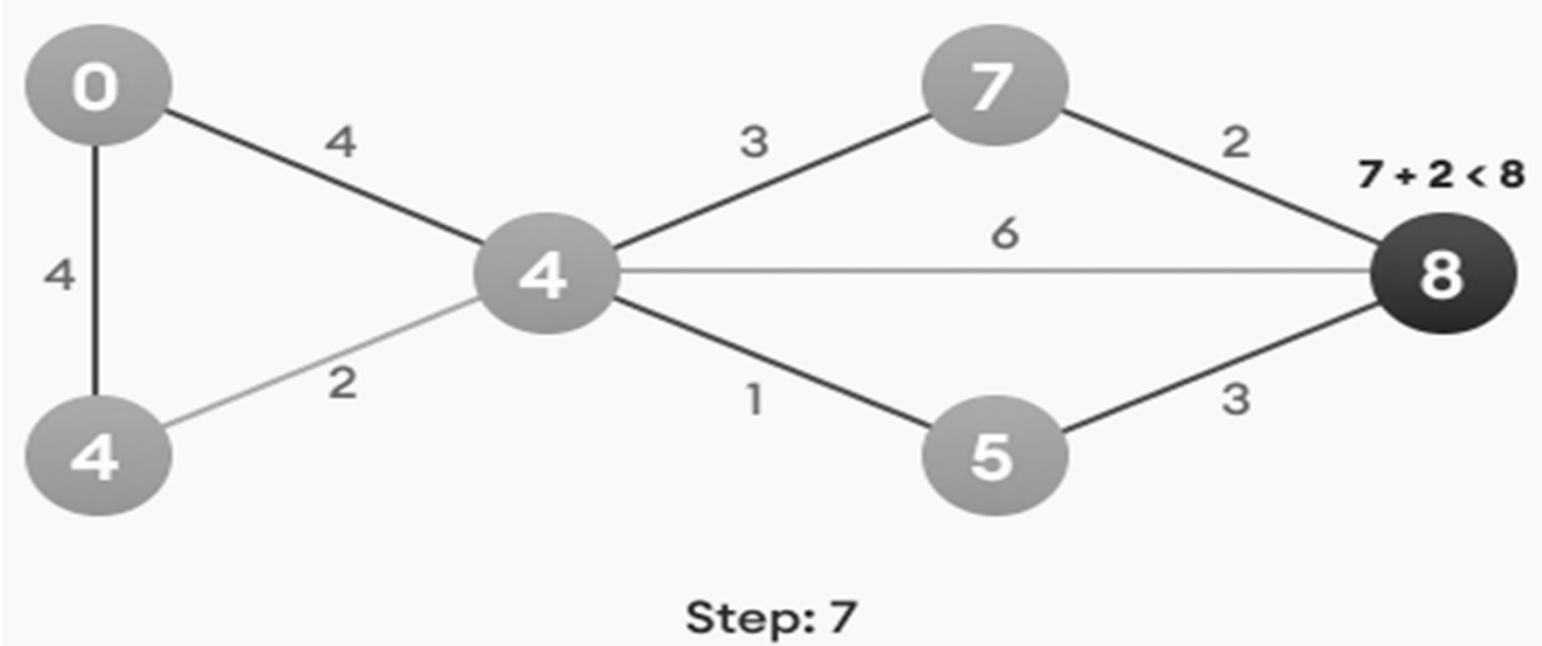
Step: 5

Avoid updating path lengths of already visited vertices



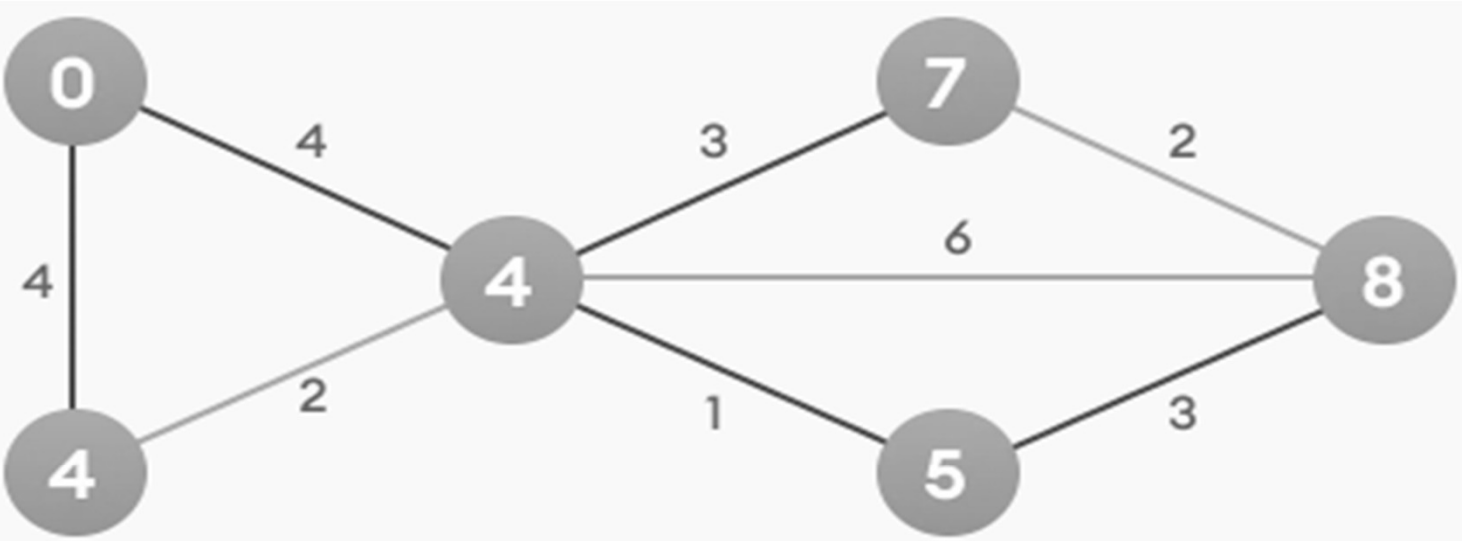
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited