SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

AN AUTONOMOUS INSTITUTION

Accredited by NBA–AICTE and Accredited by NAAC–UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

COURSE NAME : 19CS307- DATA STRUCTURES

II YEAR / III SEMESTER

UNIT-IV

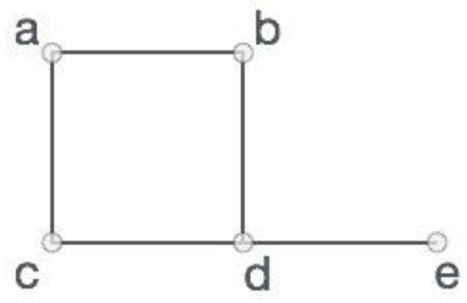
NON LINEAR DATA STRUCTURES – GRAPHS

GRAPHS

Definition

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}

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled. Edges may be directed or undirected.

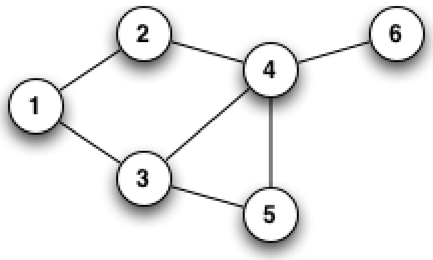
Example

Face book uses a graph data structure that consists of a group of entities and their relationships. On Face book, every user, photo, post, page, place, etc. that has data is represented with a node. Every edge from one node to another represents their relationships, friendships, ownerships, tags, etc. Whenever a user posts a photo, comments on a post, etc., a new edge is created for that relationship. Both nodes and edges have meta-data associated with them.

Types of Graph

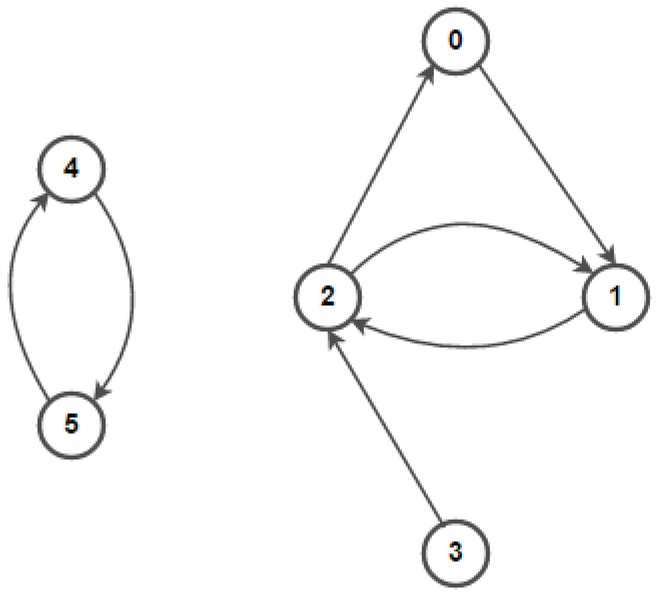
1. Undirected graph

An undirected graph (graph) is a graph in which edges have no orientation. The edge (x, y) is identical to edge (y, x), i.e., they are not ordered pairs. The maximum number of edges possible in an undirected graph without a loop is n×(n-1)/2.



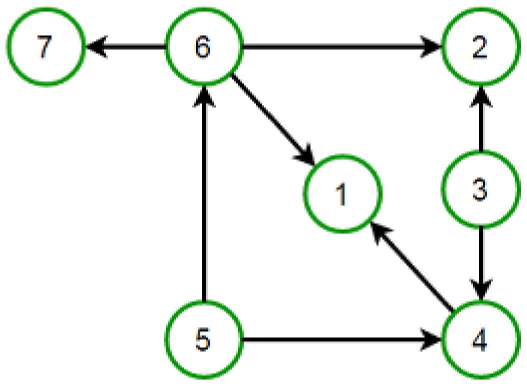
2. Directed graph

A Directed graph (digraph) is a graph in which edges have orientations, i.e., The edge (x, y) is not identical to edge (y, x).



3. Directed Acyclic Graph (DAG)

A Directed Acyclic Graph (DAG) is a directed graph that contains no cycles.

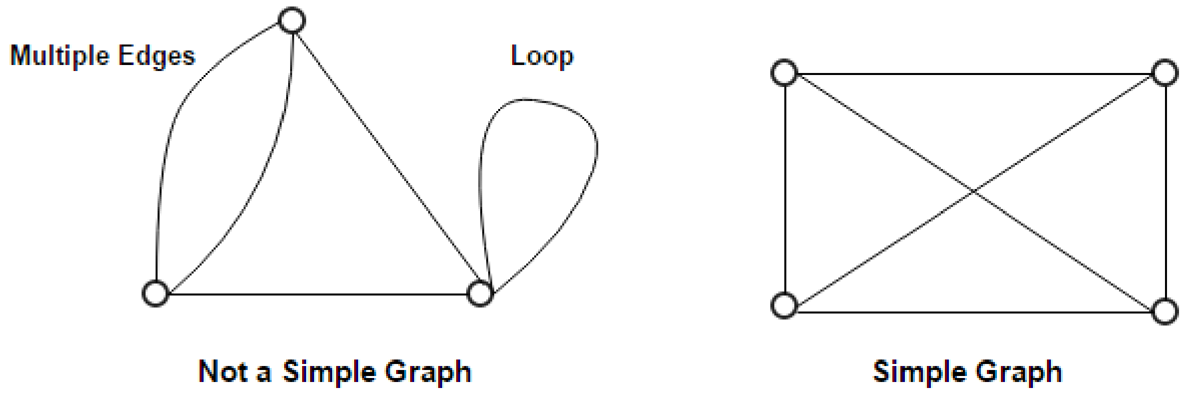


4. Multi graph

A multigraph is an undirected graph in which multiple edges (and sometimes loops) are allowed. Multiple edges are two or more edges that connect the same two vertices. A loop is an edge (directed or undirected) that connects a vertex to itself; it may be permitted or not.

5. Simple graph

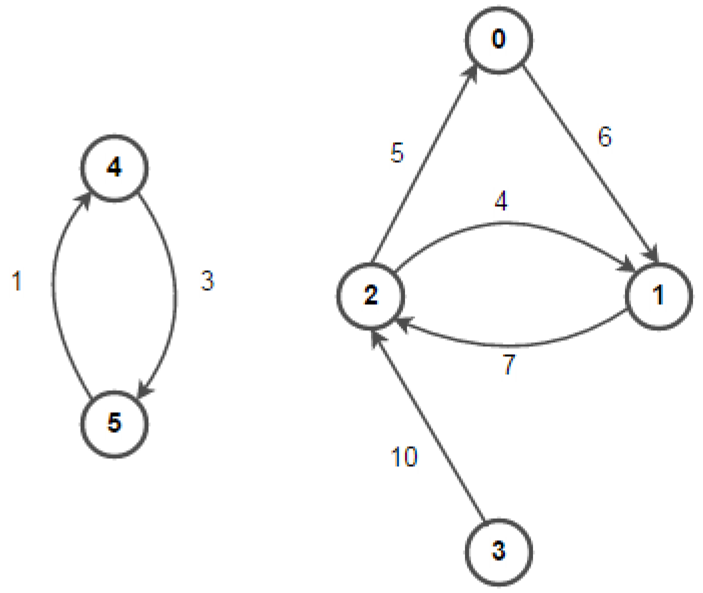
A simple graph is an undirected graph in which both multiple edges and loops are disallowed as opposed to a multigraph. In a simple graph with n vertices, every vertex's degree is at most n-1.

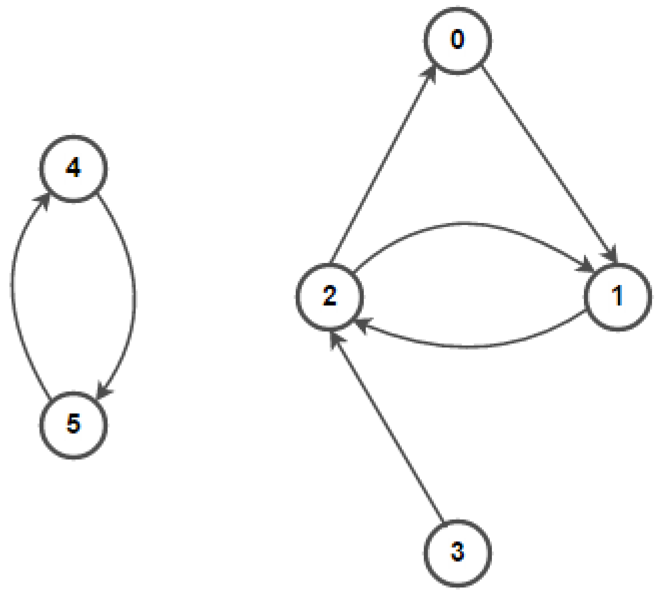


6. Weighted and Unweighted graph

A weighted graph associates a value (weight) with every edge in the graph. We can also use words cost or length instead of weight.

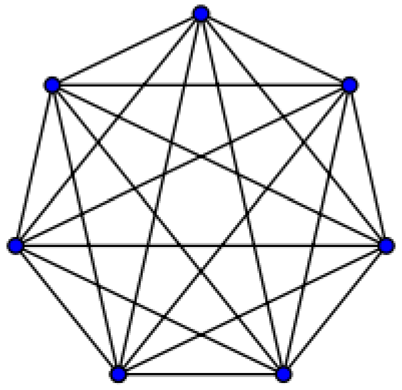
An unweighted graph does not have any value (weight) associated with every edge in the graph. In other words, an un weighted graph is a weighted graph with all edge weight as 1. Unless specified otherwise, all graphs are assumed to be un weighted by default.





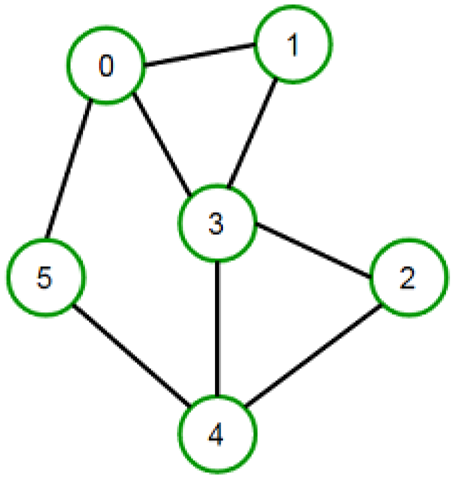
7. Complete graph

A complete graph is one in which every two vertices are adjacent: all edges that could exist are present.



8. Connected graph

A Connected graph has a path between every pair of vertices. In other words, there are no unreachable vertices. A disconnected graph is a graph that is not connected.



Relationship between number of edges and vertices

For a simple graph with m edges and n vertices, if the graph is

- directed, then m = n×(n-1)
- undirected, then m = n×(n-1)/2
- connected, then m = n-1

Graph Terminology

- An **edge** is (together with vertices) one of the two basic units out of which graphs are constructed. Each edge has two vertices to which it is attached, called its endpoints.
- Two vertices are called **adjacent** if they are end points of the same edge.
- Outgoing edges of a vertex are directed edges that the vertex is the origin.
- Incoming edges of a vertex are directed edges that the vertex is the destination.
- The **degree** of a vertex in a graph is the total number of edges incident to it.
- In a directed graph, the out-degree of a vertex is the total number of outgoing edges, and the in-degree is the total number of incoming edges.
- A vertex with in-degree zero is called a source vertex, while a vertex with out-degree zero is called a sink vertex.
- An isolated vertex is a vertex with degree zero, which is not an endpoint of an edge.
- **Path** is a sequence of alternating vertices and edges such that the edge connects each successive vertex.
- **Cycle** is a path that starts and ends at the same vertex.

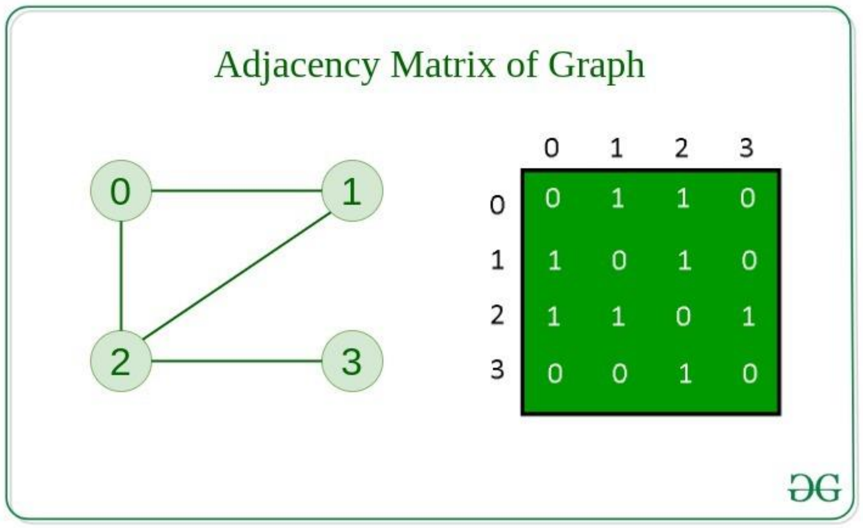
Representation of Graph

There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

1. **Adjacency Matrix**

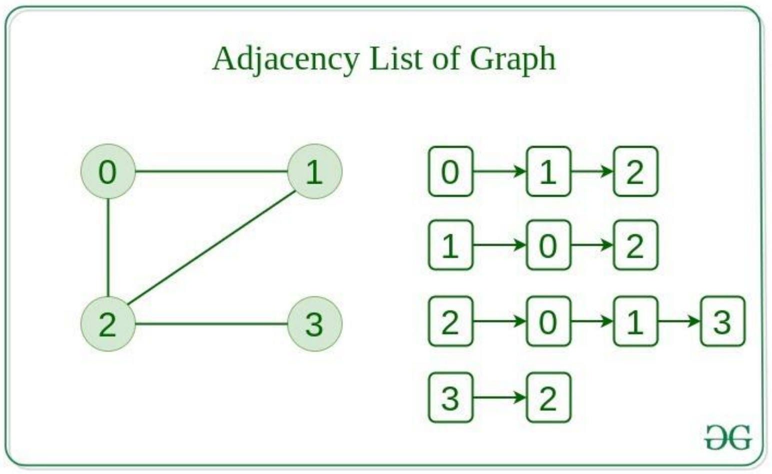
In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 0 & 0 & 1 & 0 \end{array}$$

2. **Adjacency List**

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.



$$\begin{tikzcd} 0 \arrow[r] & 1 \arrow[r] & 2 \\ 1 \arrow[r] & 0 \arrow[r] & 2 \\ 2 \arrow[r] & 0 \arrow[r] & 1 \arrow[r] & 3 \\ 3 \arrow[r] & 2 \end{tikzcd}$$

Basic Operations on Graphs

- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph.

Usage of graphs

1. **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.

2. In **Facebook,** users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.

3. In **World Wide Web,** web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.

4. In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

Graph Traversal Technique in Data Structure

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops.

There are two graph traversal techniques

1. Breadth-first search
2. Depth-first search

Breadth-first search

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree
The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and

- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

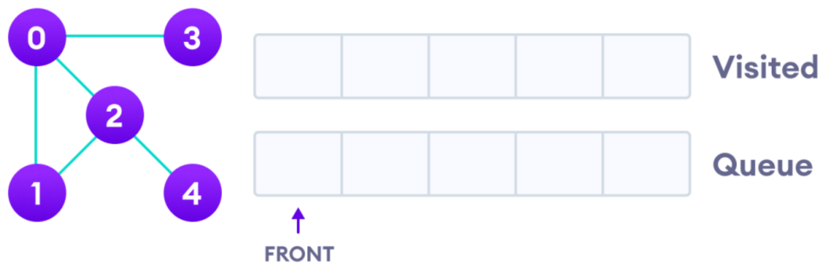
The algorithm works as follows:

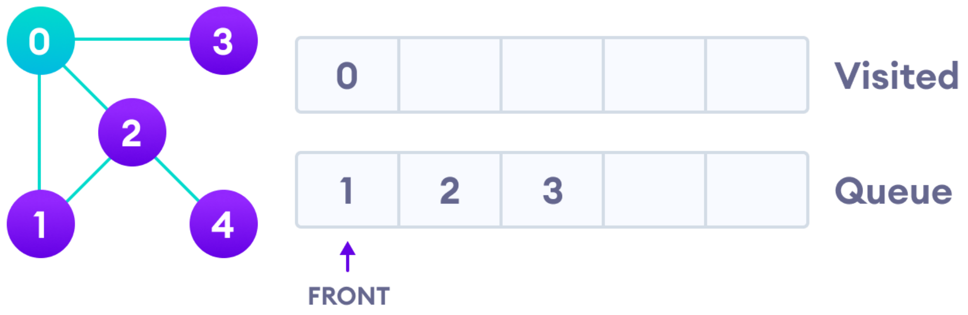
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

BFS example

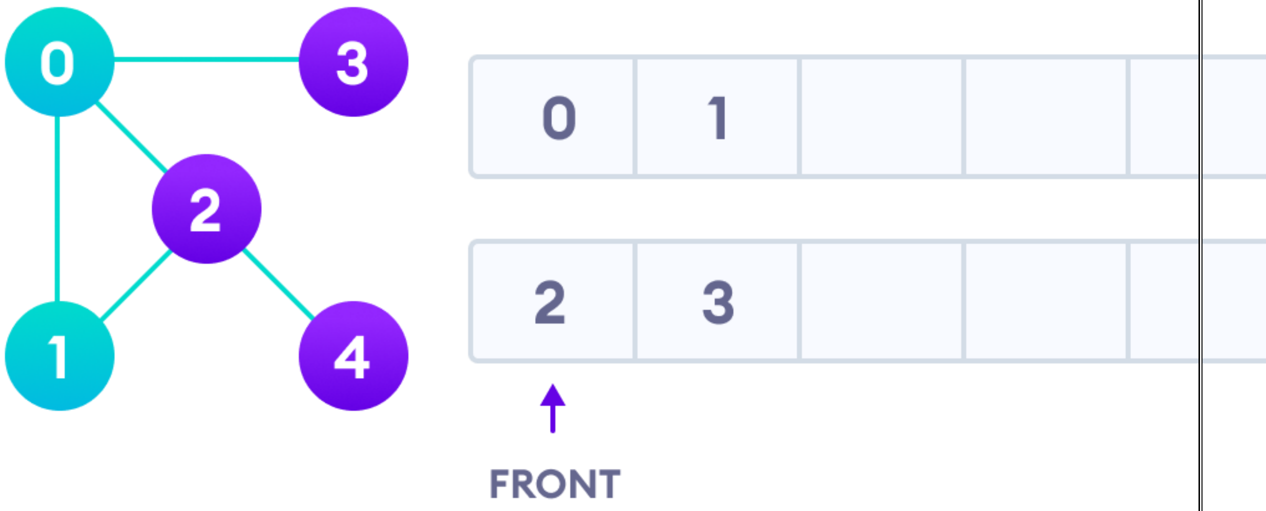
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.





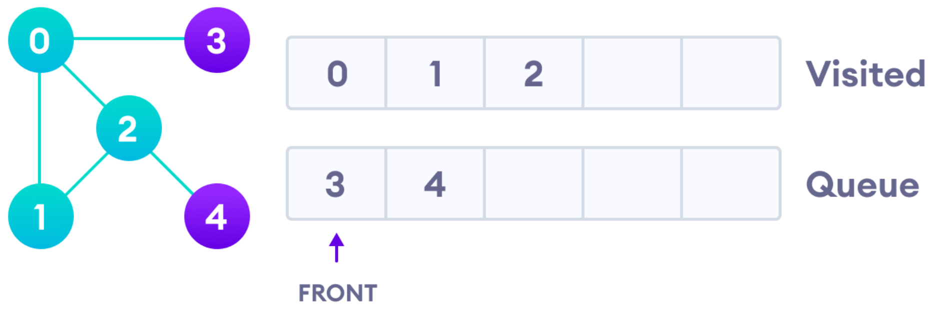
Visit start vertex and add its adjacent vertices to queue

Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

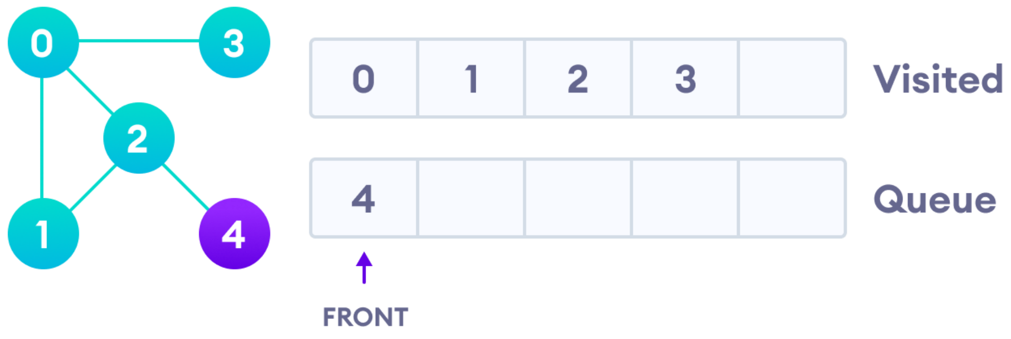


Visit the first neighbour of start node 0, which is 1

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

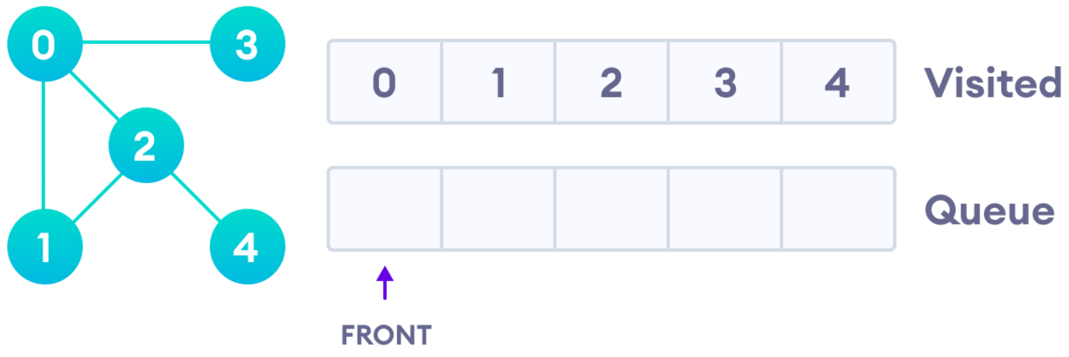


Visit 2 which was added to queue earlier to add its neighbors remains in the queue



4

Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Visit last remaining item in the queue to check if it has unvisited neighbors

Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS pseudo code/Routine

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
remove the head u of Q
mark and enqueue all (unvisited) neighbours of u
```

BFS Algorithm Applications

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree

Depth First Search (DFS)

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

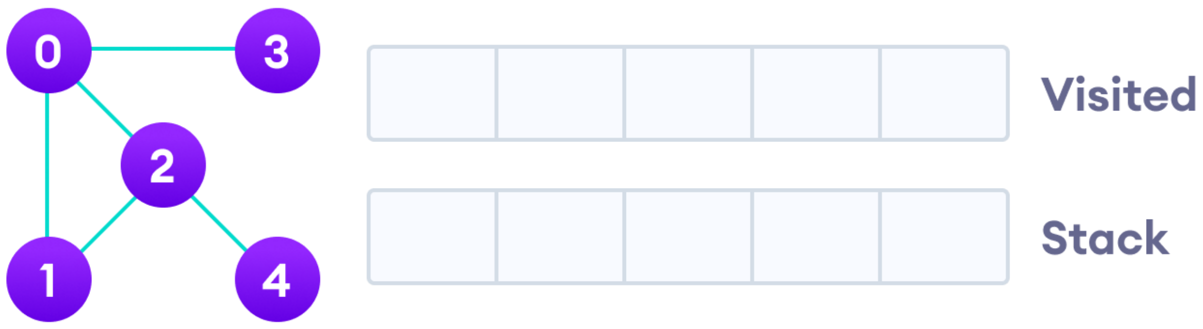
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

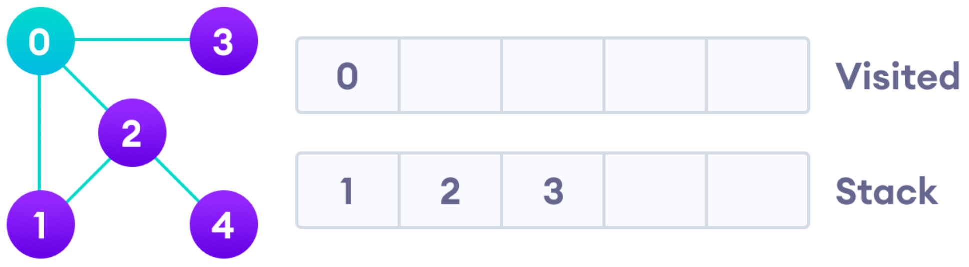
Depth First Search Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



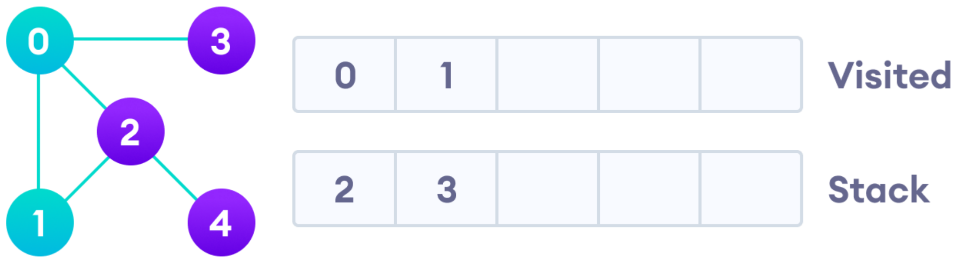
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



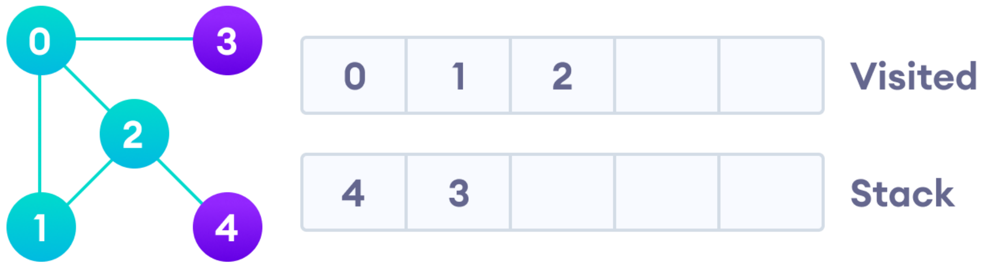
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

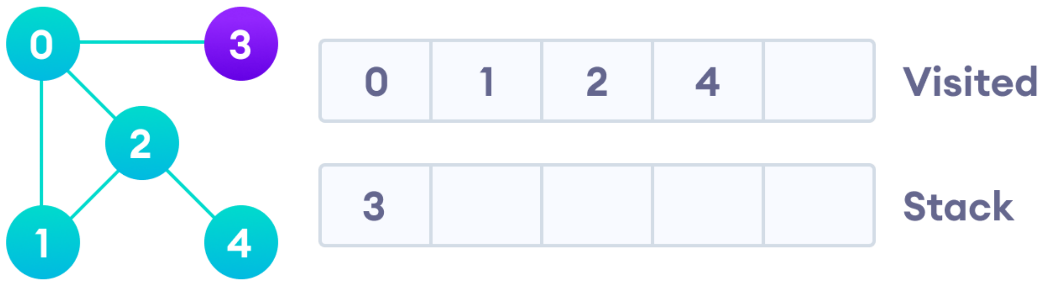


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

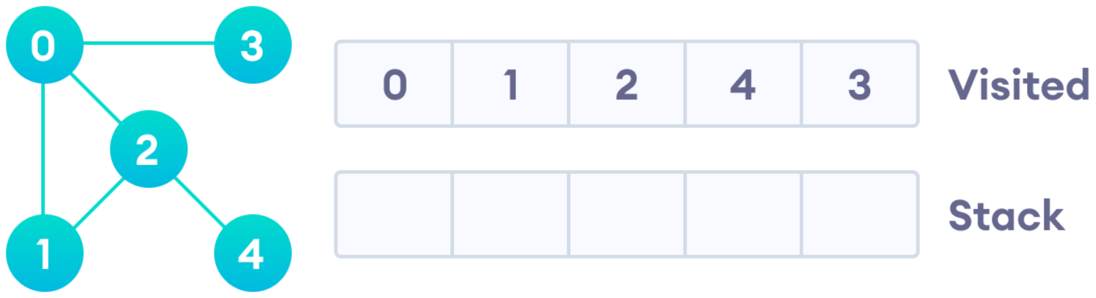


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS Pseudocode (recursive implementation)/Routine

The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node. This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
```

```
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

Application of DFS Algorithm

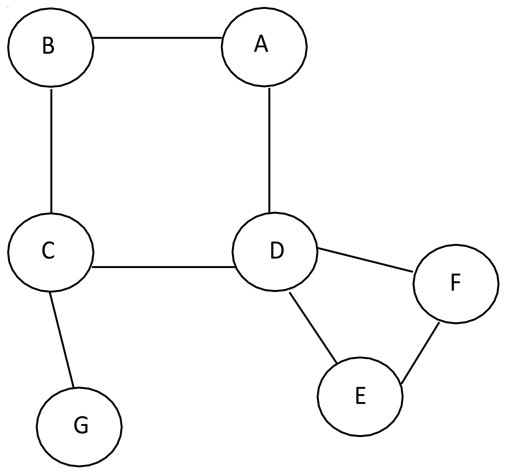
1. For finding the path
2. For finding the strongly connected components of a graph
3. For detecting cycles in a graph

BICONNECTIVITY

A connected undirected graph is biconnected if there are no vertices whose removal discon- nects the rest of the graph.

ARTICULATION POINTS OR CUT VERTEX

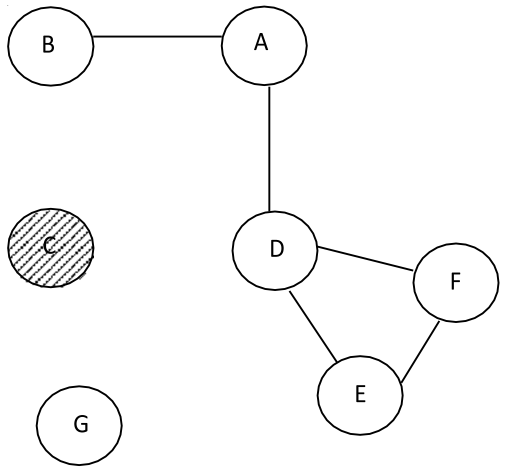
The vertices whose removal would disconnect the graph are known as articulation points.



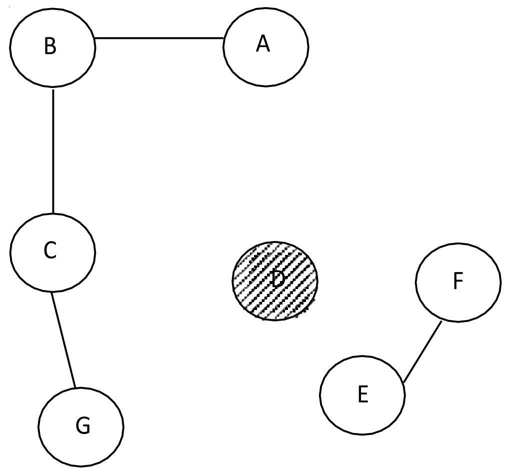
Connected Undirected Graph

Here the removal of 'C' vertex will disconnect G from the graph.

Similarly removal of 'D' vertex will disconnect E & F from the graph. Therefore 'C' & 'D' are articulation points.



(a) Removal of vertex ‘C’



(b) Removal of vertex ‘D’

The graph is not biconnected, if it has articulation points.

Depth first search provides a linear time algorithm to find all articulation points in a con- nected graph.

Steps to find Articulation Points :

Step 1 : Perform Depth first search, starting at any vertex

Step 2 : Number the vertex as they are visited, as Num (v).

Step 3 : Compute the lowest numbered vertex for every vertex v in the Depth first spanning tree, which we call as low (w), that is reachable from v by taking zero or more tree edges and then possibly one back edge. By definition, Low(v)is the minimum of

(i) Num (v)

(ii) The lowest Num (w) among all back edges (v, w)

(iii) The lowest Low (w) among all tree edges (v, w)

Step 4 : (i) The root is an articulation if and only if it has more than two children.

(ii) Any vertex v other than root is an articulation point if and only if v has same child w such that Low (w) Num (v), The time taken to compute this algorithm an a |gr|ap|h |is 0 E + V .

Note

For any edge (v, w) we can tell whether it is a tree edge or back edge merely by checking
Num (v) and Num (w).

If Num (w) > Num (v) then the edge is a back edge.

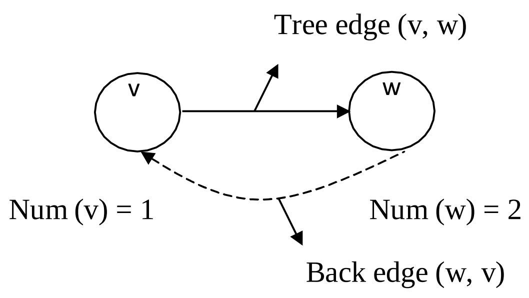
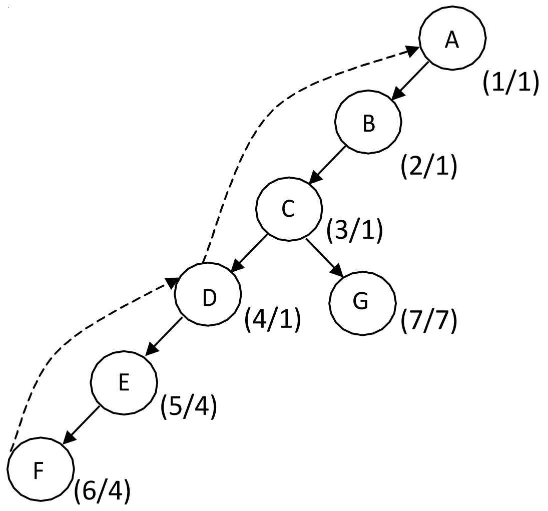


Fig. 4.8.3

```
void AssignLow (Vertex V)
{
    Vertex W;
    Low [V] = Num [V]; /* Rule 1 */
    for each W adjacent to V
    {
        If (Num [W] > Num [V]) /* forward edge */
        {
            Assign Low (W);
            If (Low [W]> = Num [V])
                Printf ("% V is an articulation pt \n", V);
            Low[V] = Min (Low [V],      Low[W]); /* Rule 3*/
        }
        else
        if (parent [V] ! = W) /* Back edge */
            Low [V] = Min (Low [V], Num [W])); /* Rule 2*/
```

ROUTINE TO COMPUTE LOW AND TEST FOR ARTICULATION POINTS



Depth First Tree With Num and Low.

Low can be computed by performing a postorder traversal of the depth - first spanning tree. (ie)

Low (F) = Min (Num (F), Num (D))

/* Since there is no tree edge & only one back edge */

= Min (6, 4) = 4

Low (F) = 4

Low (E) = Min (Num (E), Low (F))

/* there is no back edge */.

= Min (5, 4) = 4

Low (D) = Min (Num (D), Low (E), Num (A))

= Min (4,4,1) = 1

Low (D) = 1

Low (G) = Min (Num (G)) = 7 /* Since there is no tree edge & back edge */

Low (C) = Min (Num (C), Low (D), Low (G))

= Min (3,1,7) = 1

Low (C) = 1 .

lll^{ly} Low (B) = Min (Num (B), Low (C))

= Min (2,1) = 1

Low (A) = Min (Num (A), Low (B))

= Min (1, 1) = 1

Low (A) = 1.

From fig (4.8) it is clear that Low (G) > Num (C) (ie) 7 > 3 /* if Low (W) $\geq$ Num (V)*/ the 'v' is an articulation pt Therefore 'C' is an articulation point.

lll^{ly} Low (E) = Num (D), Hence D is an articulation point.

EULER'S CIRCUITEuler path

A graph is said to be containing an Euler path if it can be traced in 1 sweep without lifting the pencil from the paper and without tracing the same edge more than once. Vertices may be passed through more than once. The starting and ending points need not be the same.

Euler circuit

An Euler circuit is similar to an Euler path, except that the starting and ending points must be the same.

It is interesting that Euler never published an algorithm for finding an Euler circuit, but only provided a method of determining if one existed or not. In a note from Ed Sandifer he states, "In his paper on the Konigsberg Bridge Problem, all he says about finding such paths is that if you remove all double edges, then it will be easy to find a solution".

Euler went on to generalize this mode of thinking, laying a foundation for graph theory. Using modern vocabulary, we make the following definitions and prove at theorem:

Definition:

A network is a figure made up of points (vertices) connected by non-intersecting curves (arcs).

Definition:

A vertex is called odd if it has an odd number of arcs leading to it, otherwise it is called even.

Definition:

An Euler path is a continuous path that passes through every are once and only once.

Theorem:

If a network has more than two odd vertices, it does not have an Euler path.

Euler also proved this:

Theorem:

If a network has two or zero odd vertices, it has atleast one Euler path. In particular, if a network has exactly two odd vertices, then its Euler paths can only start on one of the odd vertices, and end on the other -- a type of Euler path called an Euler circuit.

Problem

The seven bridges of konigsberg

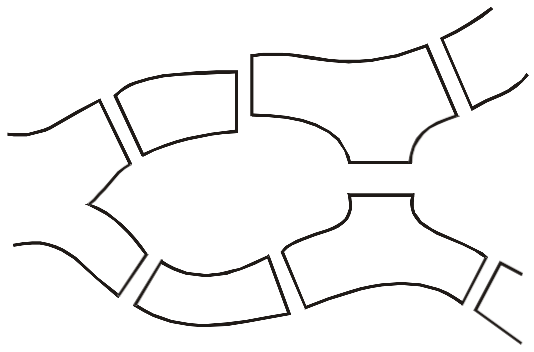


Figure 4.9

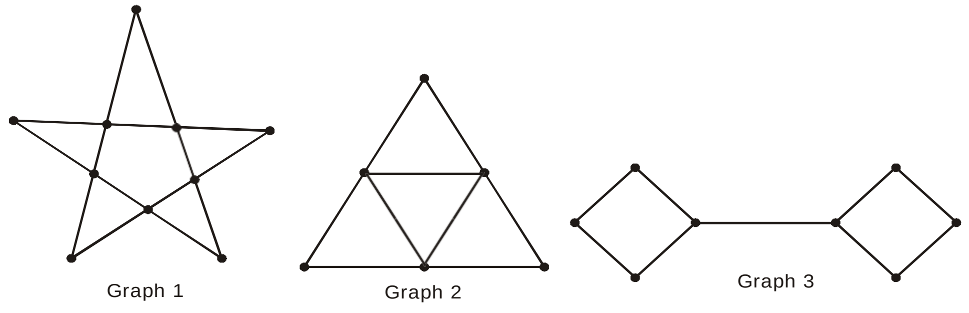
The river Pregel separates the city of Konigsberg into 4 separate regions and the regions are connected by 7 bridges. In the summer evenings, the citizens of the country would like to have a walk around the whole city. Some curious citizens wondered whether it is possible to begin at one of the regions, cross each bridge exactly once and return to the same starting point. Can the citizen's suggestion be made possible?

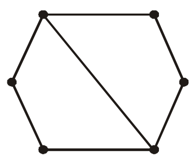
Solution:

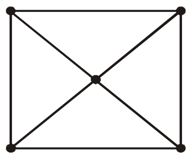
We can observe that each vertex has an odd number of edges. For example, vertex A is of degree 5 and vertex B is of degree 3. Therefore the citizen's suggestion is impossible. As each edge can be used only once and all vertices are odd, it is impossible to re-enter any vertex again after leaving it, and this makes starting and ending at the same point impossible.

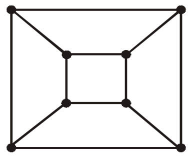
Problems

For each of the networks below, determine whether it has an Euler path. If it does, find one.









Graph	*Number of odd vertices(vertices connected to an odd number of edges)*	*Number of even vertices (vertices connected to an even number of edges*	*What does the path contain? (Euler path = P; Euler circuit = C; Neither = N)*
1	0	10	C
2	0	6	C
3	2	6	P
4	2	4	P
5	4	1	N
6	8	0	N

From the above table, we can observe that:

1. A graph with all vertices being even contains an Euler circuit.
2. A graph with 2 odd vertices and some even vertices contains an Euler path.
3. A graph with more than 2 odd vertices does not contain any Euler path or circuit.

APPLICATIONS OF GRAPHS

1. **Social network graphs**: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in socialstructures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. **Transportation networks**. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. **Utility graphs**. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. **Document link graphs**. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

5. **Protein-protein interactions graphs**. Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions amongthem.
6. **Network packet traffic graphs**. Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
7. **Scene graphs.** In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.
8. **Finite element meshes**. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is calleda finite element mesh.
9. **Robot planning**. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomousvehicles.

10. **Neural networks.** Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons

Dijkstra's Algorithm

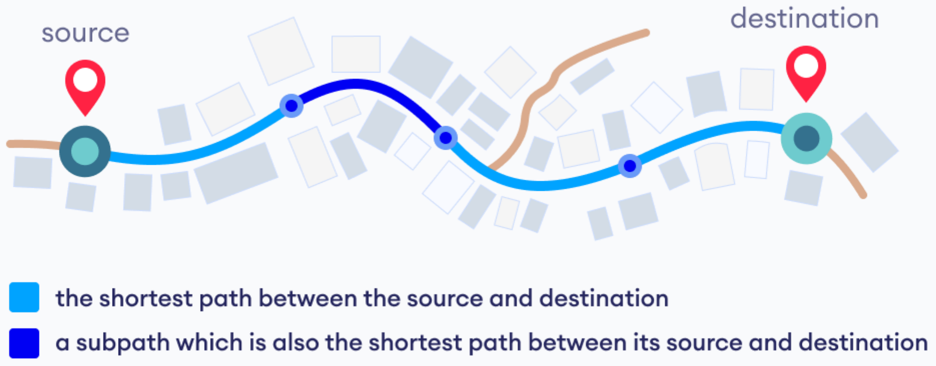
Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices

might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.


source
destination
the shortest path between the source and destination
a subpath which is also the shortest path between its source and destination

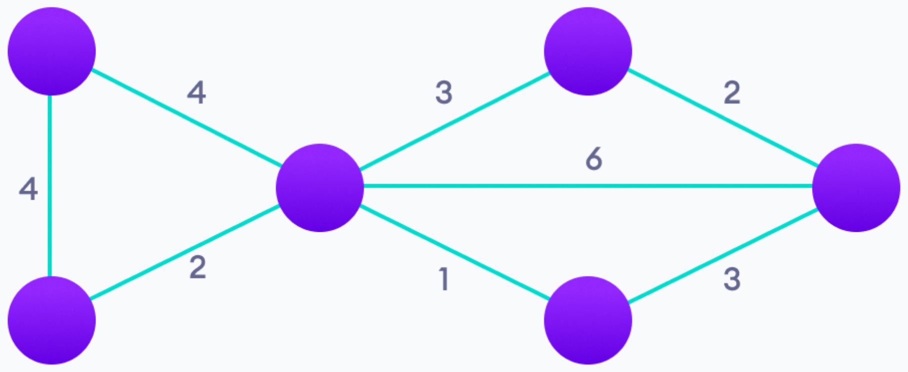
Each subpath is the shortest path

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

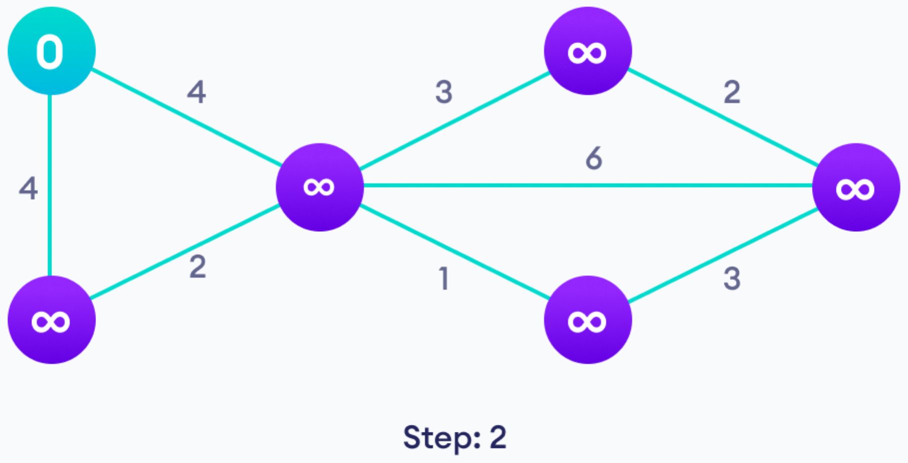
Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.

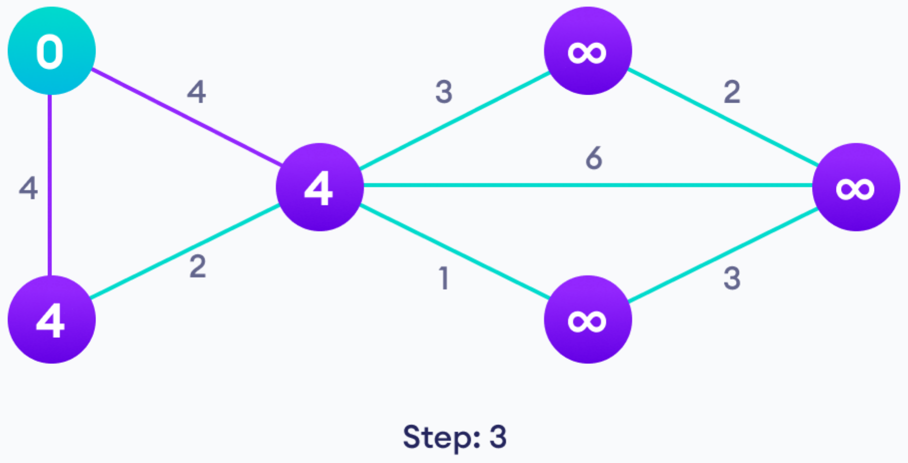


Step: 1

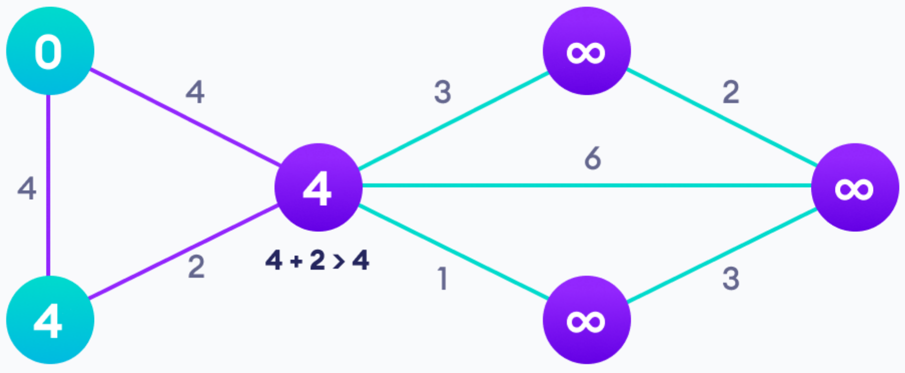
Start with a weighted graph



Choose a starting vertex and assign infinity path values to all other devices

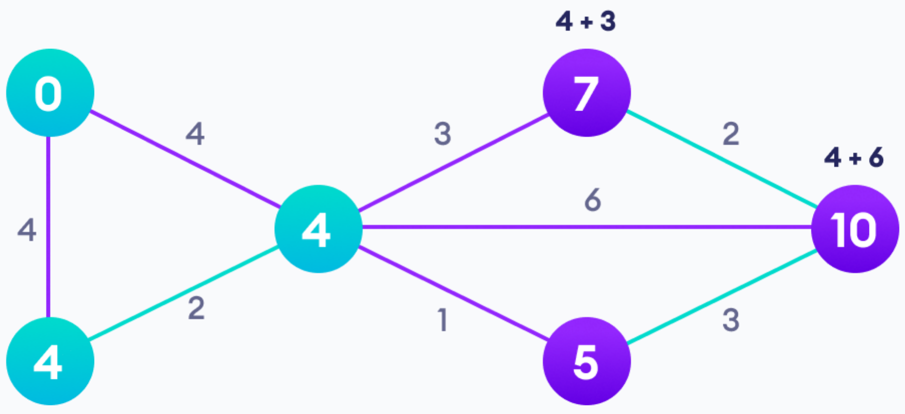


Go to each vertex and update its path length



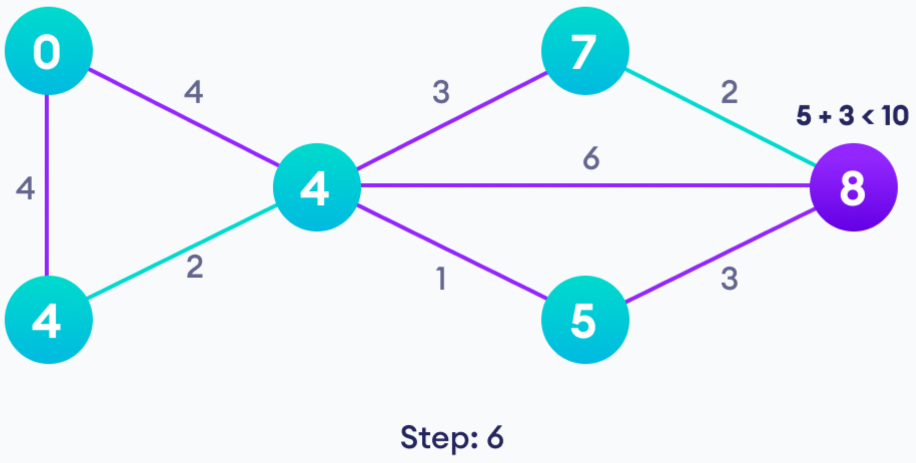
Step: 4

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

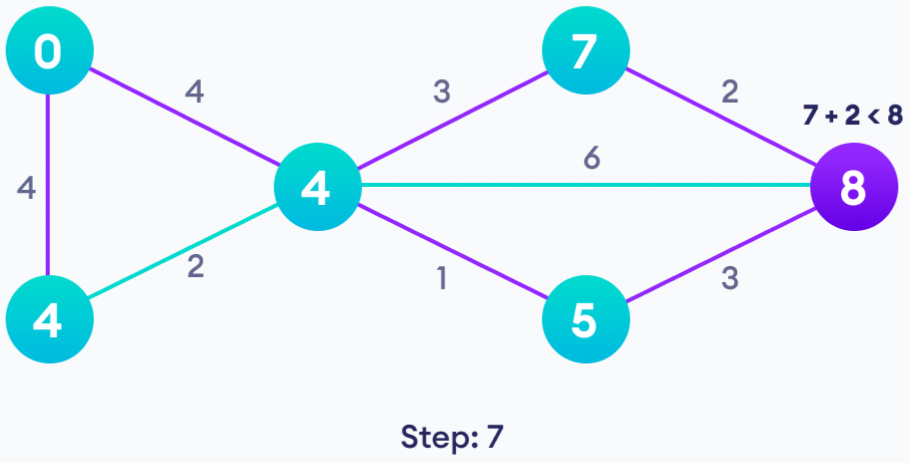


Step: 5

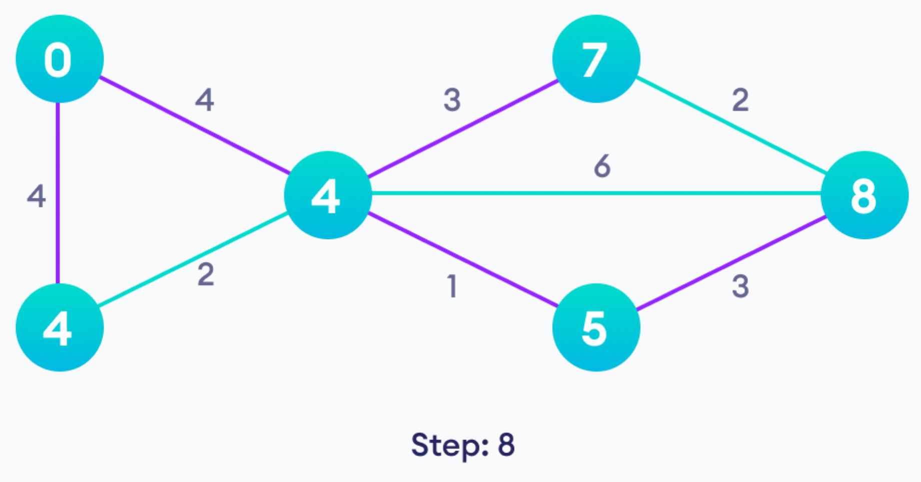
Avoid updating path lengths of already visited vertices



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



Notice how the rightmost vertex has its path length updated twice



Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
    U <- Extract MIN from Q
    for each unvisited neighbour V of U
        tempDistance <- distance[U] + edge_weight(U, V)
        if tempDistance < distance[V]
            distance[V] <- tempDistance
            previous[V] <- U
return distance[], previous[]
```

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

---

How Prim's algorithm works

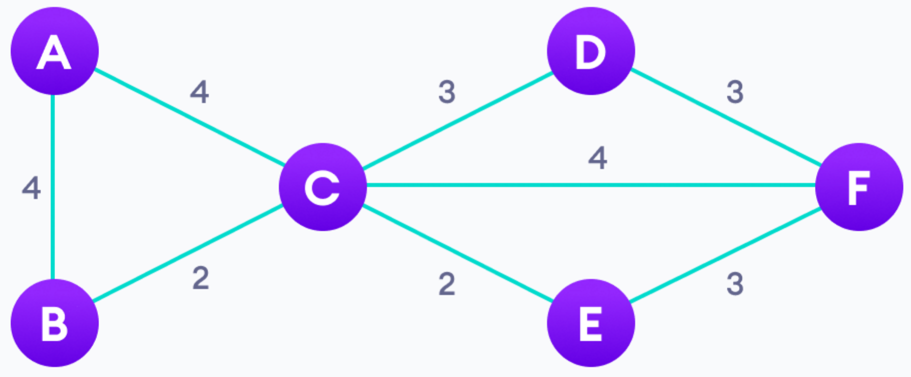
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

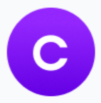
---

Example of Prim's algorithm



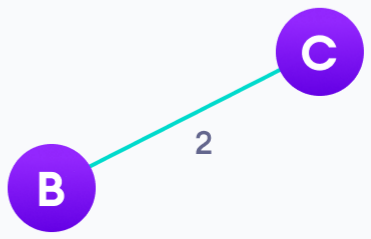
Step: 1

Start with a weighted graph



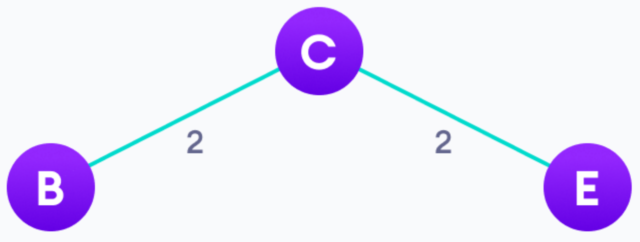
Step: 2

Choose a vertex



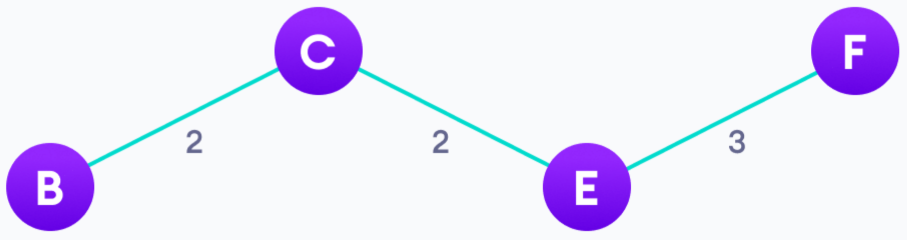
Step: 3

Choose the shortest edge from this vertex and add it



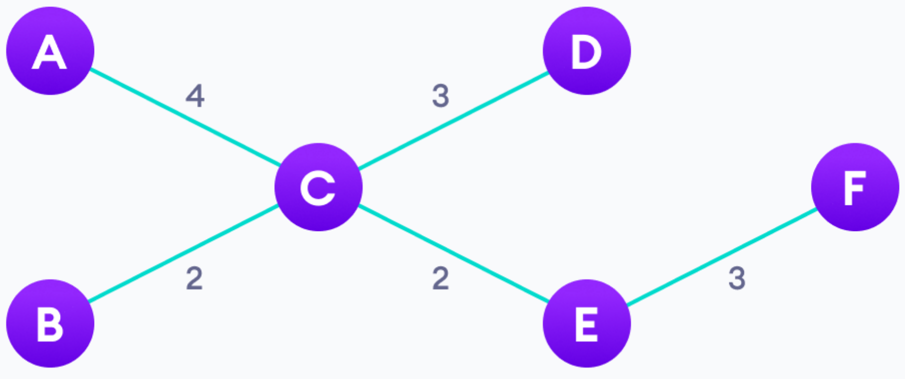
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Prim's Algorithm pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

```
T = ∅;
U = { 1 };
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
    T = T ∪ {(u, v)}
    U = U ∪ {v}
```

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

---

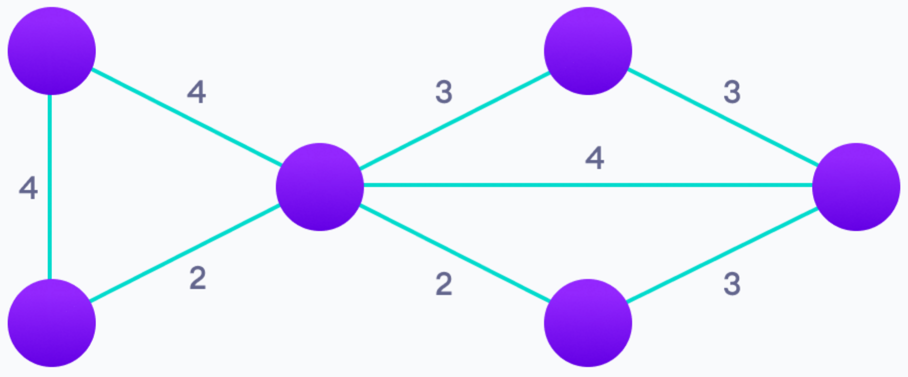
How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.
We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

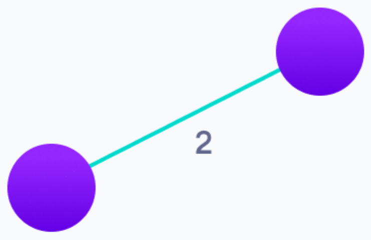
1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



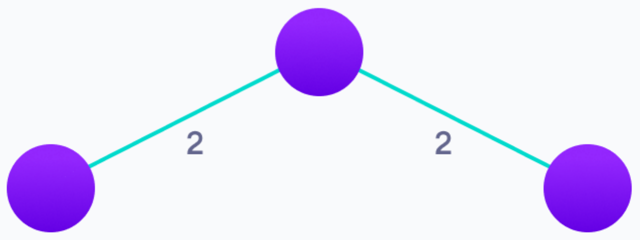
Step: 1

Start with a weighted graph



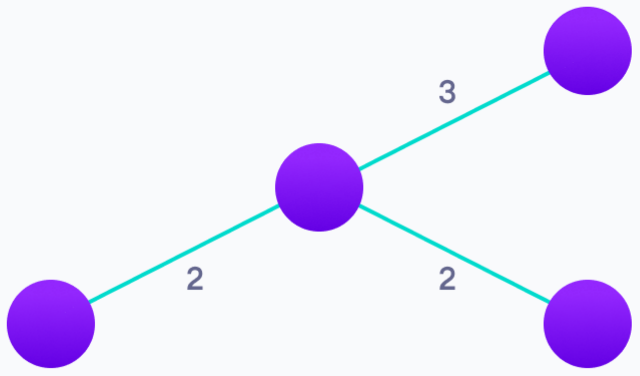
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



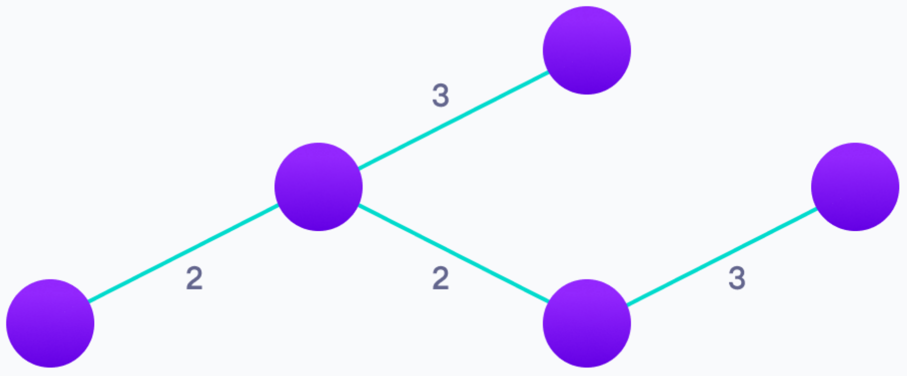
Step: 3

Choose the next shortest edge and add it



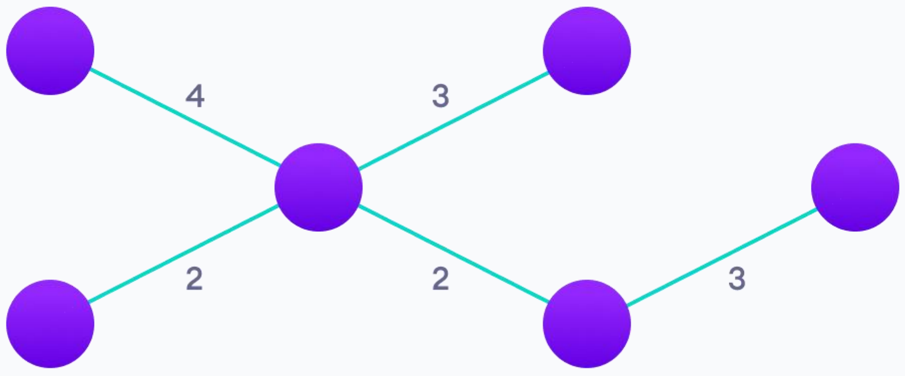
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called Union FInd. The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```