# Week 15

# Lesson 29-30

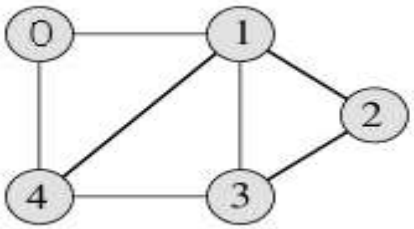
## Objectives

- Graph
  - What is a Graph?
  - Graph Vocabulary
  - Why we use Graph?
  - Real Life Examples
  - Types of Graph
    - Directed Graph
    - Undirected Graph
  - Representation of Graph
    - Adjacency Matrix
    - Adjacency List
    - Adjacency Matrix Vs. Adjacency List
  - Difference between Graph and Tree
    - Tree
    - Graph
  - Graph Operations
  - Graph Terminologies
    - Adjacent nodes
    - Path
    - Complete Graph
    - Weighted Graph
  - Depth First Search
  - Breadth First Search
  - Minimal Spanning Tree
    - Kruskal's Algorithm
    - Prims Algorithm

## What is Graph?

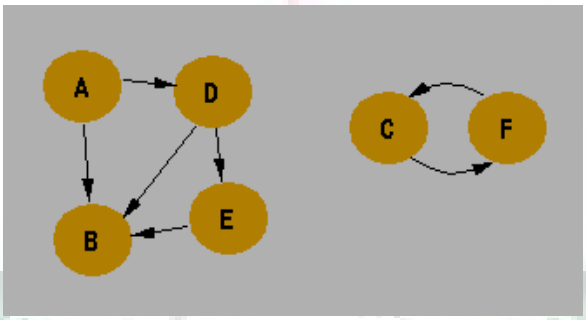
- Graph is a data structure.
- A person can use graph to save information.
- Graph is a data structure that consists of following two components:
  **1.** A finite set of nodes known as vertices.
  **2.** A finite set of edges that relate the nodes to each other.
- The set of edges describes relationships among the vertices.

Following is an example undirected graph with 5 vertices:

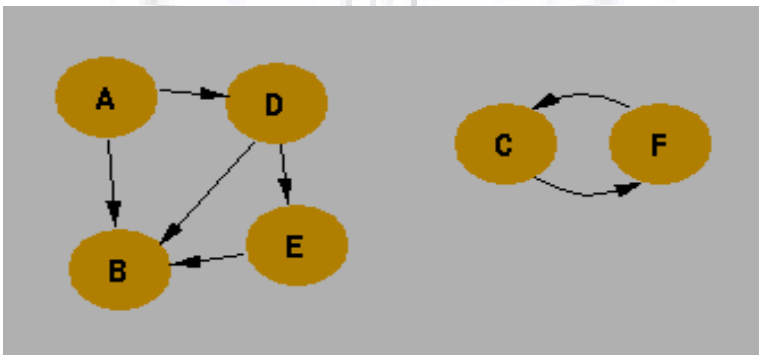



## Graph vocabulary:

- The letters are held in what are called the ***vertices*** of the graph.
- Vertices can be connected to other vertices.
- A connection between 2 vertices is called an ***edge***.
- This example graph is a ***directed graph***.
- This just means that each edge in the graph is ***unidirectional***,
- i.e., it goes from one vertex to another.
  For example, there is an *edge* from **D** to **B**,
  but there is in no edge representing the reverse relationship (from **B** to **D**).
- Here is a simple graph that stores letters:




- Also, all the vertices aren't connected in this example graph.
- I.e., there are connections between **A, B, D** and **E**,
  but there is no way to *get to* vertex **C** from any of those vertices.
  Thus, **A, B, D** and **E** from their own *component*.
  A second component is made up of **C** and **F**.




## Why we use graph?

- Some types of information are naturally represented in a graph.
- For example, the graph could be viewed as a map of what cities are connected by **train routes**.
  Viewing it this way;
  1. Each *vertex* represents a particular city
  2. Each *edge* represents whether there is a train route from one city to another.
  3. We can imagine that the edges are unidirectional
     since trains are only allowed to go in one direction on the track.
- The *edges* in our graph represent a very simple relationship.
- For example; one city is connected to another.
- We can store information at vertices (e.g., the city name)
- We can also store information at each edge.
  For example,
  we want to store the ***distance*** between cities at edges OR
  the ***time*** the trip takes OR
  the ***cost*** of the ticket OR
  all of those pieces of information.

# Real Life Example:

Graphs are used to represent many real life applications:

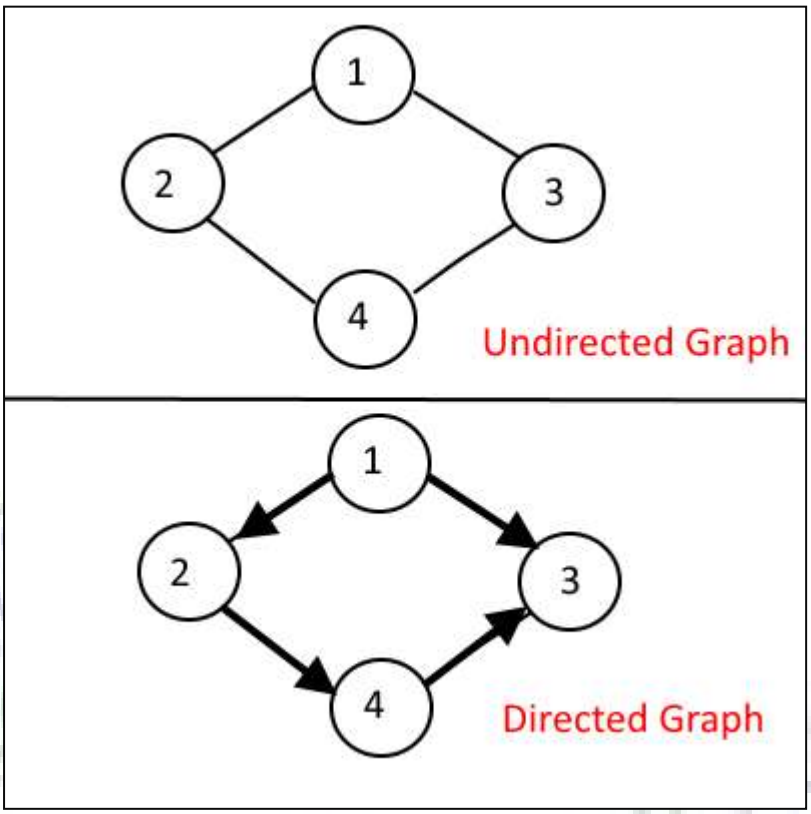
1. Train route
2. Networks
3. Network – can be path in a city, telephone network or circuit network
4. Social Networking – such as facebook; in facebook each person is a vertex (node). Each node is a structure and contains information like person id, name, gender and location.
5. Communication networks – i.e. Network topologies.
6. Computer networks and the Internet. Often nodes will represent end-systems or routers, while edges represent connections between these systems.
7. **Molecules:** Graphs can be used to model atoms and molecules for studying their interaction and structure among other things.

# Representation of graph:

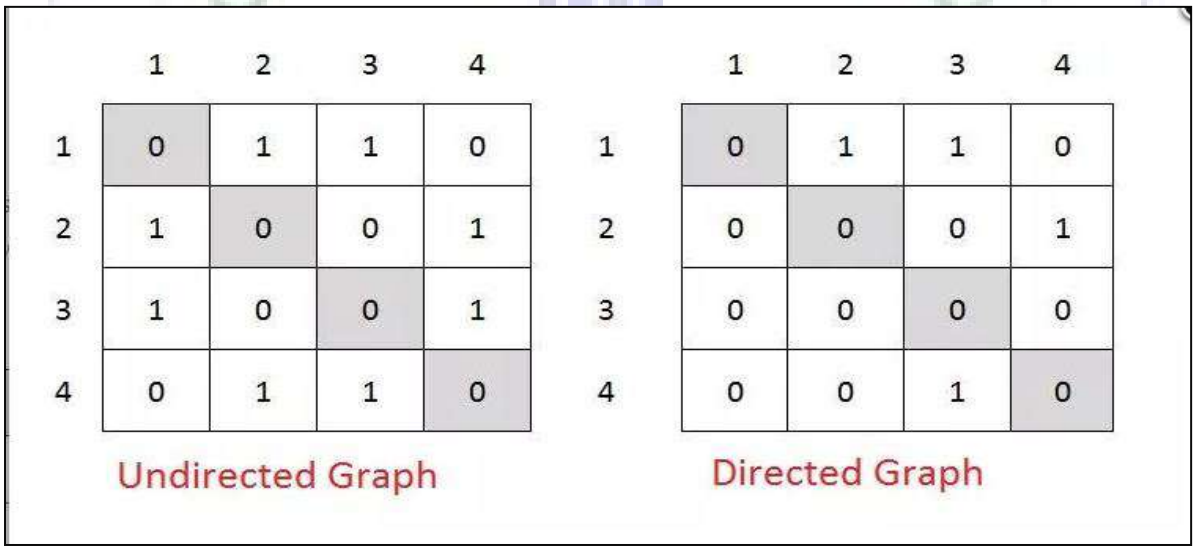
## 1. Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be A[][], a blockA[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs.
- If A[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

These are the directed and indirected graphs:




The adjacency matrix of the following diected and undirected graph is:




## Code:

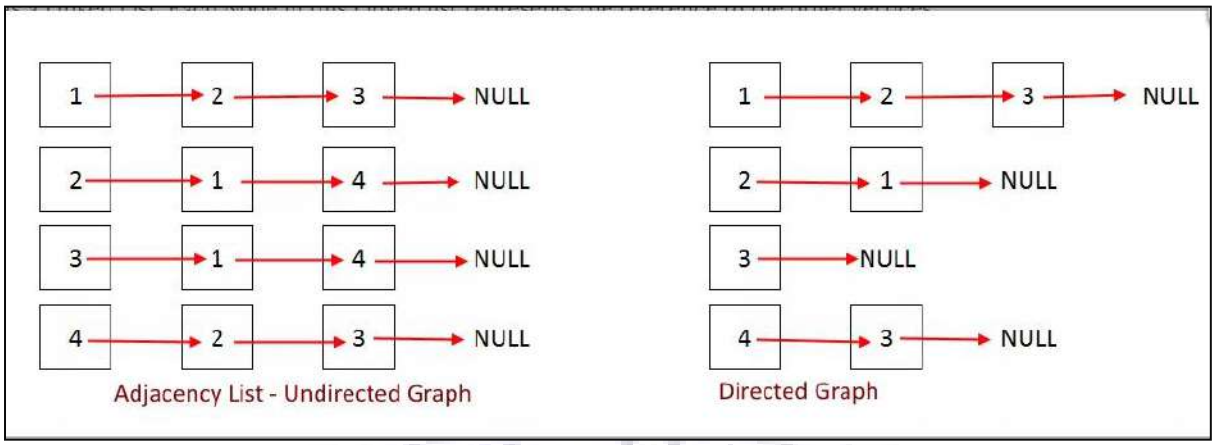
```
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // array of vertices
private int adjMat[][];      // adjacency matrix
private int nVerts;          // current number of vertices
// ----------------------------------------------------------public
Graph()              // constructor
```

```
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
                                adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++)      // set adjacency
for(int k=0; k<MAX_VERTS; k++)   //    matrix to 0
adjMat[j][k] = 0;
}  // end constructor
// --------------------------------------------------------------
public void addVertex(char lab)    // argument is label
{
vertexList[nVerts++] = new Vertex(lab);
}
// --------------------------------------------------------------public
void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// --------------------------------------------------------------public
void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
// --------------------------------------------------------------}
                                // end class Graph
```

## 2. Adjacency List

- An array of linked lists is used.
- Size of the array is equal to number of vertices.
- Let the array be array[].
- An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex.
- This representation can also be used to represent a weighted graph.
- The weights of edges can be stored in nodes of linked lists.

Following is adjacency list representation of the above graph.




## Adjacency List vs. Adjacency Matrix:

| Adjacency Matrix | Adjacency List |
|---|---|
| Connectivity between two vertices can be check quickly. | Difficult to check the connectivity between two vertices. |
| It's not space efficient. | Space efficient |
| It's easier to implement. | Vertices adjacent to another can be found easily |
| It's easier to follow. | |

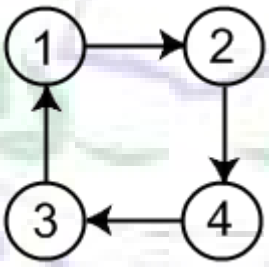
## Types of Graphs:

All graphs are divided into two groups or types.

## Directed Graphs:

- When the edges in a graph have a direction, the graph is called directed (or digraph).
- A graph is directed if each line (which is drawn to represent the edge) has an arrow.

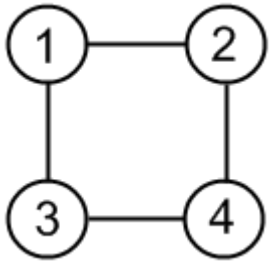
Example of a Directed Graph




## Undirected Graphs:

- When the edges in a graph have no direction, the graph is called undirected.
- Edge can be drawn as a line. A graph is undirected if each line doesn't have an arrow.

Example of an Undirected Graph




# Difference between Graphs and Trees:

**Tree:**

- Trees have a strict hierarchical structure.
- Trees are not flexible.
- Trees don't have loop.
- Each node must be directly or indirectly connected to the root of the tree.

**Graph:**

- Graphs don't have a strict hierarchical structure.
- Graph is flexible but Graphs have loops.
- Graphs may contain node (known as vertex) that is maybe disconnected.

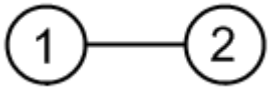
# Graph operations:

- Graphs are pretty generic data structures in that they can be used to represent lots of things.
- Thus, exactly what operations we'll want for a graph will depend on what we want to do with it.
- Suppose we want the following operations:
    1. Adding a new vertex to the graph.
    2. Adding an edge to the graph.
    3. Removing an edge
    4. Removing a vertex

# Graph Terminology:

## Adjacent nodes:

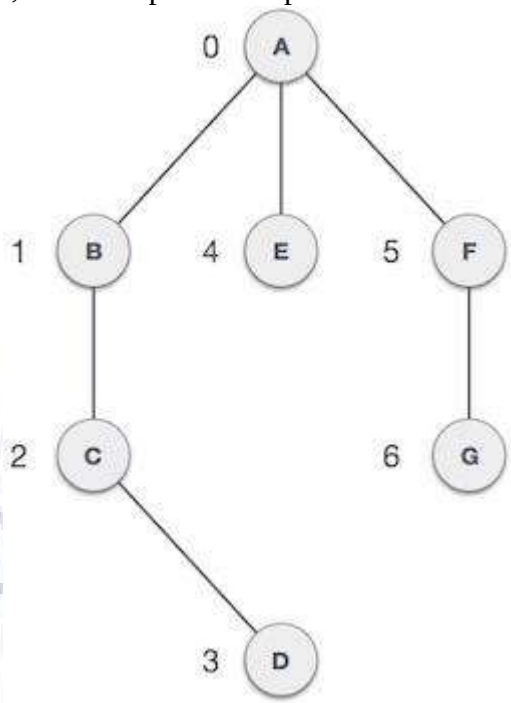
Two nodes are adjacent (parallel) if they are connected by an edge.




1 is adjacent to 2 or 2 is adjacent to 1

## Path:

Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



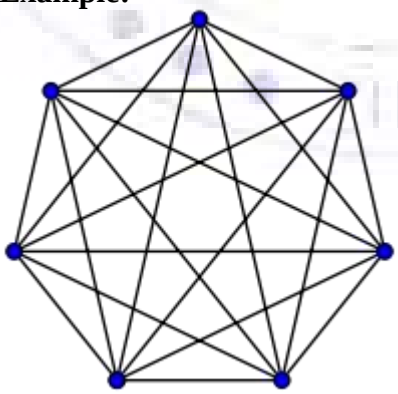

# Difference Between a path and an Edge

- An edge is a direct connection like in the above example there is an edge between A to E.
- A Path can be a direct or an indirect connection like in the above example there is a pth between A to d which is an indirect connection and there is also a path between A to E which is a direct connection.Hence,we can say that "**All edges are path but all paths are not edges**".

## Complete Graph:

A graph in which every vertex is directly connected to every other vertex.

**Example:**

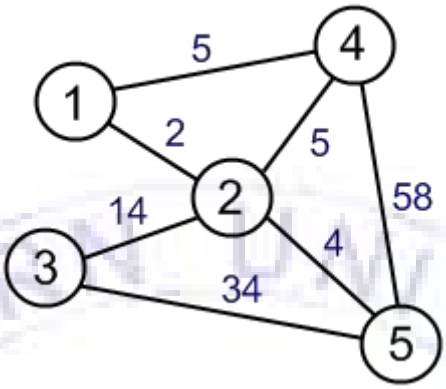


## Weighted Graph:

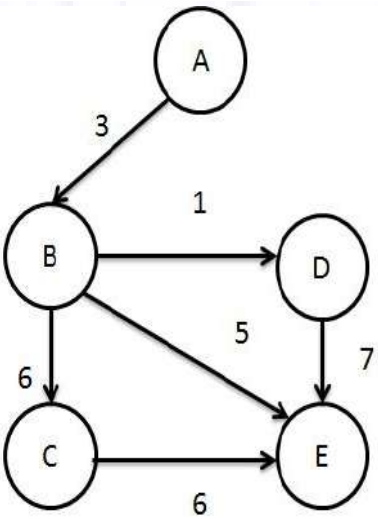
A graph in which every edge carries a value. Value can be anything – distance, time, cost, etc.

For instance, in the road network example, weight of each road may be its length or minimal time needed to drive along.

**Weighted UnDirected Graph**




**Weighted Directed Graph**

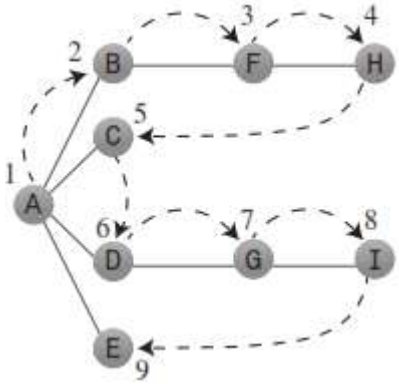



## Depth-First Search

The depth-first search uses a stack to remember where it should go when it reaches a dead end. We'll show an example, encourage you to try similar examples with the GraphN Workshop applet, and then finally show some code that carries out the search.

An Example

Consider an example



To carry out the depth-first search, you pick a starting point—in this case, vertex A You then do three things: visit this vertex, push it onto a stack so you can remember it, and mark it so you won't visit it again.

Next, you go to any vertex adjacent to A that hasn't yet been visited. We'll assume the vertices are selected in alphabetical order, so that brings up B. You visit B, mark it, and push it on the stack.

Now what? You're at B, and you do the same thing as before: go to an adjacent vertex that hasn't been visited. This leads you to F. We can call this process Rule 1.

RULE 1

*If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.*

Applying Rule 1 again leads you to H. At this point, however, you need to do something else because there are no unvisited vertices adjacent to H. Here's where Rule 2 comes in.

RULE 2

*If you can't follow Rule 1, then, if possible, pop a vertex off the stack.*

Following this rule, you pop H off the stack, which brings you back to F. F has no unvisited adjacent vertices, so you pop it. Ditto B. Now only A is left on the stack. A, however, does have unvisited adjacent vertices, so you visit the next one, C. But C is the end of the line again, so you pop it and you're back to A. You visit D, G, and I, and then pop them all when you reach the dead end at I. Now you're back to A. You visit E, and again you're back to A. This time, however, A has no unvisited neighbors, so we pop it off the stack. But now there's nothing left to pop, which brings up Rule 3.

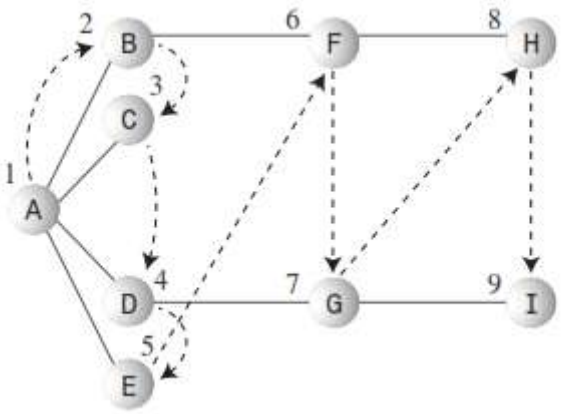
RULE 3

*If you can't follow Rule 1 or Rule 2, you're done.*

| Event | Stack |
|---|---|
| Visit A | A |
| Visit B | AB |
| Visit F | ABF |
| Visit H | ABFH |
| Pop H | ABF |
| Pop F | AB |
| Pop B | A |
| Visit C | AC |
| Pop C | A |
| Visit D | AD |
| Visit G | ADG |
| Visit I | ADGI |
| Pop I | ADG |
| Pop G | AD |
| Pop D | A |
| Visit E | AE |
| Pop E | A |
| Pop A | |
| Done | |

## Breadth-First Search

As we saw in the depth-first search, the algorithm acts as though it wants to get as far away from the starting point as quickly as possible. In the breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of search is implemented using a queue instead of a stack. An Example




A is the starting vertex, so you visit it and make it the current vertex. Then you follow these rules:

RULE 1

*Visit the next unvisited vertex (if there is one) that's adjacent to the current vertex, mark it, and insert it into the queue.*

RULE 2

*If you can't carry out Rule 1 because there are no more unvisited vertices, remove a vertex from the queue (if possible) and make it the current vertex.*

RULE 3

*If you can't carry out Rule 2 because the queue is empty, you're done.*

Thus, you first visit all the vertices adjacent to A, inserting each one into the queue as you visit it. Now you've visited A, B, C, D, and E. At this point the queue (from front to rear) contains BCDE.

There are no more unvisited vertices adjacent to A, so you remove B from the queue and look for vertices adjacent to it. You find F, so you insert it in the queue. There are no more unvisited vertices adjacent to B, so you remove C from the queue. It has no adjacent unvisited vertices, so you remove D and visit G. D has no more adjacent unvisited vertices, so you remove E. Now the queue is FG. You remove F and visit H, and then you remove G and visit I.

Now the queue is HI, but when you've removed each of these and found no adjacent unvisited vertices, the queue is empty, so you're done. Table shows this sequence.

| Event | Queue (Front to Rear) |
|---|---|
| Visit A | |
| Visit B | B |
| Visit C | BC |
| Visit D | BCD |
| Visit E | BCDE |
| Remove B | CDE |
| Visit F | CDEF |
| Remove C | DEF |
| Remove D | EF |
| Visit G | EFG |
| Remove E | FG |
| Remove F | G |
| Visit H | GH |
| Remove G | H |
| Visit I | HI |
| Remove H | I |
| Remove I | |
| Done | |

## Minimum Spanning Trees

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees. Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

- Cluster Analysis
- Handwriting recognition
- Image segmentation

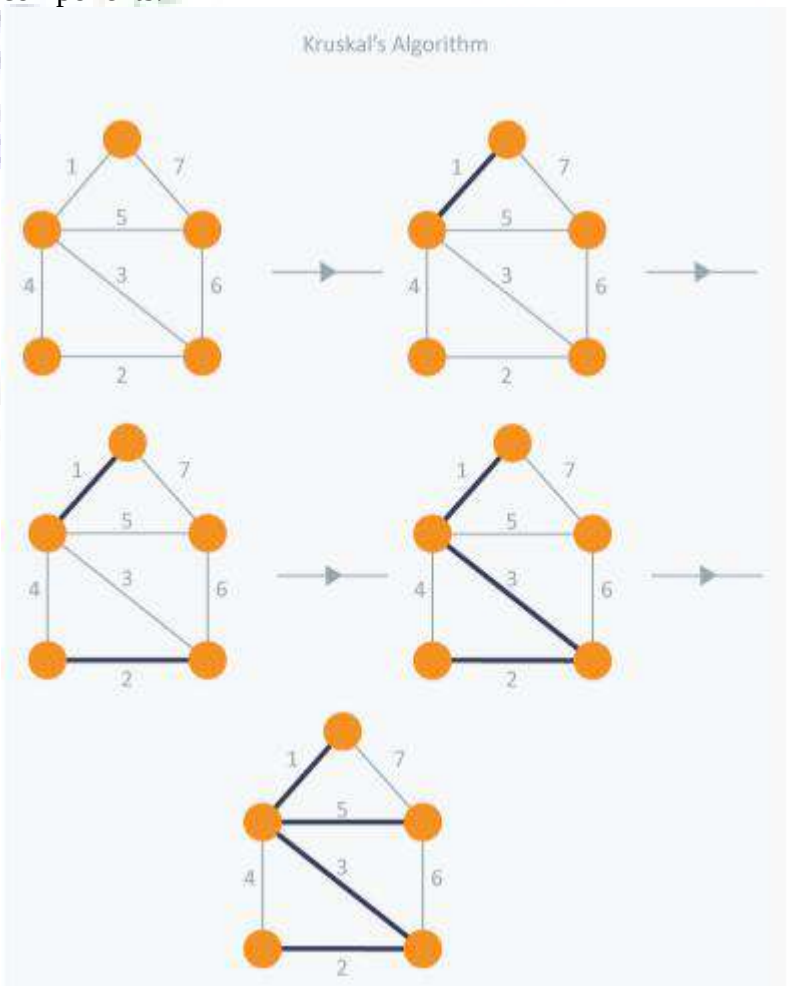
There are two famous algorithms for finding the Minimum Spanning Tree:

## Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.



## Prim’s Algorithm

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.